
returns

Release 0.22.0

dry-python team

Aug 26, 2023

CONTENTS

1	Installation	3
2	Contents	5
3	Maybe container	7
4	RequiresContext container	9
5	Result container	11
5.1	Straight-forward approach	11
5.2	Hidden problems	11
5.3	Pipe example	12
6	IO container	13
6.1	Troublesome IO	13
6.2	Explicit IO	14
7	Future container	15
7.1	Mixing sync and async code	15
7.2	Async code without exceptions	16
7.3	Better async composition	16
8	More!	19
9	Contents	21
9.1	Quickstart	21
9.2	Container: the concept	22
9.3	Railway oriented programming	29
9.4	Higher Kinded Types	31
9.5	Interfaces	39
9.6	Maybe	82
9.7	Result	93
9.8	IO	106
9.9	Future	126
9.10	Context	153
9.11	Create your own container	213
9.12	Pipelines	227
9.13	Converters	234
9.14	Pointfree	236
9.15	Methods	256
9.16	Do Notation	258

9.17	Helper functions	261
9.18	Curry	266
9.19	Trampolines	272
9.20	Primitive types	274
9.21	Development tooling	279
9.22	mypy plugin	282
9.23	pytest plugin	287
9.24	hypothesis plugin	291
9.25	Version history	297
10	Indices and tables	311
	Python Module Index	313
	Index	315



Make your functions return something meaningful, typed, and safe!

- Brings functional programming to Python land
- Provides a bunch of primitives to write declarative business logic
- Enforces better architecture
- Fully typed with annotations and checked with mypy, [PEP561 compatible](#)
- Adds emulated Higher Kinded Types support
- Provides type-safe interfaces to create your own data-types with enforced laws
- Has a bunch of helpers for better composition
- Pythonic and pleasant to write and to read
- Support functions and coroutines, framework agnostic
- Easy to start: has lots of docs, tests, and tutorials

[Quickstart](#) right now!

INSTALLATION

```
pip install returns
```

You can also install `returns` with the latest supported `mypy` version:

```
pip install returns[compatible-mypy]
```

You would also need to configure our ``mypy`` plugin https://returns.readthedocs.io/en/latest/pages/contrib/mypy_plugins.html>`_:

```
# In setup.cfg or mypy.ini:  
[mypy]  
plugins =  
    returns.contrib.mypy.returns_plugin
```

or:

```
[tool.mypy]  
plugins = ["returns.contrib.mypy.returns_plugin"]
```

We also recommend to use the same `mypy` settings we use.

Make sure you know how to get started, check out our docs! Try our demo.

CONTENTS

- *Maybe container* that allows you to write None-free code
- *RequiresContext container* that allows you to use typed functional dependency injection
- *Result container* that let's you to get rid of exceptions
- *IO container* and *IOResult* that marks all impure operations and structures them
- *Future container* and *FutureResult* to work with async code
- **Write your own container!** You would still have all the features for your own types (including full existing code reuse and type-safety)

MAYBE CONTAINER

None is called the [worst mistake in the history of Computer Science](#).

So, what can we do to check for None in our programs? You can use builtin [Optional](#) type and write a lot of `if some is not None:` conditions. But, **having ``null`` checks here and there makes your code unreadable**.

```
user: Optional[User]
discount_program: Optional['DiscountProgram'] = None

if user is not None:
    balance = user.get_balance()
    if balance is not None:
        credit = balance.credit_amount()
        if credit is not None and credit > 0:
            discount_program = choose_discount(credit)
```

Or you can use [Maybe](#) container! It consists of `Some` and `Nothing` types, representing existing state and empty (instead of `None`) state respectively.

```
from typing import Optional
from returns.maybe import Maybe, maybe

@maybe # decorator to convert existing Optional[int] to Maybe[int]
def bad_function() -> Optional[int]:
    ...

maybe_number: Maybe[float] = bad_function().bind_optional(
    lambda number: number / 2,
)
# => Maybe will return Some[float] only if there's a non-None value
#    Otherwise, will return Nothing
```

You can be sure that `.bind_optional()` method won't be called for `Nothing`. Forget about `None`-related errors forever!

We can also bind a `Optional`-returning function over a container. To achieve this, we are going to use `.bind_optional` method.

And here's how your initial refactored code will look:

```
user: Optional[User]

# Type hint here is optional, it only helps the reader here:
```

(continues on next page)

(continued from previous page)

```
discount_program: Maybe['DiscountProgram'] = Maybe.from_optional(  
    user,  
).bind_optional( # This won't be called if `user is None`  
    lambda real_user: real_user.get_balance(),  
).bind_optional( # This won't be called if `real_user.get_balance()` is None  
    lambda balance: balance.credit_amount(),  
).bind_optional( # And so on!  
    lambda credit: choose_discount(credit) if credit > 0 else None,  
)
```

Much better, isn't it?

REQUIRESCONTEXT CONTAINER

Many developers do use some kind of dependency injection in Python. And usually it is based on the idea that there's some kind of a container and assembly process.

Functional approach is much simpler!

Imagine that you have a django based game, where you award users with points for each guessed letter in a word (unguessed letters are marked as '. '):

```
from django.http import HttpRequest, HttpResponse
from words_app.logic import calculate_points

def view(request: HttpRequest) -> HttpResponse:
    user_word: str = request.POST['word'] # just an example
    points = calculate_points(user_word)
    ... # later you show the result to user somehow

# Somewhere in your `words_app/logic.py`:

def calculate_points(word: str) -> int:
    guessed_letters_count = len([letter for letter in word if letter != '.'])
    return _award_points_for_letters(guessed_letters_count)

def _award_points_for_letters(guessed: int) -> int:
    return 0 if guessed < 5 else guessed # minimum 6 points possible!
```

Awesome! It works, users are happy, your logic is pure and awesome. But, later you decide to make the game more fun: let's make the minimal accountable letters threshold configurable for an extra challenge.

You can just do it directly:

```
def _award_points_for_letters(guessed: int, threshold: int) -> int:
    return 0 if guessed < threshold else guessed
```

The problem is that `_award_points_for_letters` is deeply nested. And then you have to pass `threshold` through the whole callstack, including `calculate_points` and all other functions that might be on the way. All of them will have to accept `threshold` as a parameter! This is not useful at all! Large code bases will struggle a lot from this change.

Ok, you can directly use `django.settings` (or similar) in your `_award_points_for_letters` function. And **ruin your pure logic with framework specific details**. That's ugly!

Or you can use `RequiresContext` container. Let's see how our code changes:

```
from django.conf import settings
from django.http import HttpRequest, HttpResponse
from words_app.logic import calculate_points

def view(request: HttpRequest) -> HttpResponse:
    user_word: str = request.POST['word'] # just an example
    points = calculate_points(user_word)(settings) # passing the dependencies
    ... # later you show the result to user somehow

# Somewhere in your `words_app/logic.py`:

from typing import Protocol
from returns.context import RequiresContext

class _Deps(Protocol): # we rely on abstractions, not direct values or types
    WORD_THRESHOLD: int

def calculate_points(word: str) -> RequiresContext[int, _Deps]:
    guessed_letters_count = len([letter for letter in word if letter != '.'])
    return _award_points_for_letters(guessed_letters_count)

def _award_points_for_letters(guessed: int) -> RequiresContext[int, _Deps]:
    return RequiresContext(
        lambda deps: 0 if guessed < deps.WORD_THRESHOLD else guessed,
    )
```

And now you can pass your dependencies in a really direct and explicit way. And have the type-safety to check what you pass to cover your back. Check out [RequiresContext](#) docs for more. There you will learn how to make '.' also configurable.

We also have [RequiresContextResult](#) for context-related operations that might fail. And also [RequiresContextIOResult](#) and [RequiresContextFutureResult](#).

RESULT CONTAINER

Please, make sure that you are also aware of Railway Oriented Programming.

5.1 Straight-forward approach

Consider this code that you can find in **any** python project.

```
import requests

def fetch_user_profile(user_id: int) -> 'UserProfile':
    """Fetches UserProfile dict from foreign API."""
    response = requests.get('/api/users/{0}'.format(user_id))
    response.raise_for_status()
    return response.json()
```

Seems legit, does it not? It also seems like a pretty straightforward code to test. All you need is to mock `requests.get` to return the structure you need.

But, there are hidden problems in this tiny code sample that are almost impossible to spot at the first glance.

5.2 Hidden problems

Let's have a look at the exact same code, but with the all hidden problems explained.

```
import requests

def fetch_user_profile(user_id: int) -> 'UserProfile':
    """Fetches UserProfile dict from foreign API."""
    response = requests.get('/api/users/{0}'.format(user_id))

    # What if we try to find user that does not exist?
    # Or network will go down? Or the server will return 500?
    # In this case the next line will fail with an exception.
    # We need to handle all possible errors in this function
    # and do not return corrupt data to consumers.
    response.raise_for_status()

    # What if we have received invalid JSON?
```

(continues on next page)

```
# Next line will raise an exception!
return response.json()
```

Now, all (probably all?) problems are clear. How can we be sure that this function will be safe to use inside our complex business logic?

We really cannot be sure! We will have to create **lots** of `try` and `except` cases just to catch the expected exceptions. Our code will become complex and unreadable with all this mess!

Or we can go with the top level `except Exception:` case to catch literally everything. And this way we would end up with catching unwanted ones. This approach can hide serious problems from us for a long time.

5.3 Pipe example

```
import requests
from returns.result import Result, safe
from returns.pipeline import flow
from returns.pointfree import bind

def fetch_user_profile(user_id: int) -> Result['UserProfile', Exception]:
    """Fetches `UserProfile` TypedDict from foreign API."""
    return flow(
        user_id,
        _make_request,
        bind(_parse_json),
    )

@safe
def _make_request(user_id: int) -> requests.Response:
    # TODO: we are not yet done with this example, read more about `IO`:
    response = requests.get('/api/users/{0}'.format(user_id))
    response.raise_for_status()
    return response

@safe
def _parse_json(response: requests.Response) -> 'UserProfile':
    return response.json()
```

Now we have a clean and a safe and declarative way to express our business needs:

- We start from making a request, that might fail at any moment,
- Then parsing the response if the request was successful,
- And then return the result.

Now, instead of returning regular values we return values wrapped inside a special container thanks to the `@safe` decorator. It will return `Success[YourType]` or `Failure[Exception]`. And will never throw exception at us!

We also use `flow` and `bind` functions for handy and declarative composition.

This way we can be sure that our code won't break in random places due to some implicit exception. Now we control all parts and are prepared for the explicit errors.

We are not yet done with this example, let's continue to improve it in the next chapter.

IO CONTAINER

Let's look at our example from another angle. All its functions look like regular ones: it is impossible to tell whether they are *pure* or *impure* from the first sight.

It leads to a very important consequence: *we start to mix pure and impure code together*. We should not do that!

When these two concepts are mixed we suffer really bad when testing or reusing it. Almost everything should be *pure* by default. And we should explicitly mark *impure* parts of the program.

That's why we have created IO container to mark *impure* functions that never fail.

These *impure* functions use *random*, *current datetime*, *environment*, or *console*:

```
import random
import datetime as dt

from returns.io import IO

def get_random_number() -> IO[int]: # or use `@impure` decorator
    return IO(random.randint(1, 10)) # isn't pure, because random

now: Callable[[], IO[dt.datetime]] = impure(dt.datetime.now)

@impure
def return_and_show_next_number(previous: int) -> int:
    next_number = previous + 1
    print(next_number) # isn't pure, because does IO
    return next_number
```

Now we can clearly see which functions are *pure* and which ones are *impure*. This helps us a lot in building large applications, unit testing you code, and composing business logic together.

6.1 Troublesome IO

As it was already said, we use IO when we handle functions that do not fail.

What if our function can fail and is *impure*? Like `requests.get()` we had earlier in our example.

Then we have to use a special `IOResult` type instead of a regular `Result`. Let's find the difference:

- Our `_parse_json` function always returns the same result (hopefully) for the same input: you can either parse valid json or fail on invalid one. That's why we return *pure* `Result`, there's no IO inside

- Our `_make_request` function is impure and can fail. Try to send two similar requests with and without internet connection. The result will be different for the same input. That's why we must use `IOResult` here: it can fail and has IO

So, in order to fulfill our requirement and separate pure code from impure one, we have to refactor our example.

6.2 Explicit IO

Let's make our IO explicit!

```
import requests
from returns.io import IOResult, impure_safe
from returns.result import safe
from returns.pipeline import flow
from returns.pointfree import bind_result

def fetch_user_profile(user_id: int) -> IOResult['UserProfile', Exception]:
    """Fetches `UserProfile` TypedDict from foreign API."""
    return flow(
        user_id,
        _make_request,
        # before: def (Response) -> UserProfile
        # after safe: def (Response) -> ResultE[UserProfile]
        # after bind_result: def (IOResultE[Response]) -> IOResultE[UserProfile]
        bind_result(_parse_json),
    )

@impure_safe
def _make_request(user_id: int) -> requests.Response:
    response = requests.get('/api/users/{0}'.format(user_id))
    response.raise_for_status()
    return response

@safe
def _parse_json(response: requests.Response) -> 'UserProfile':
    return response.json()
```

And later we can use `unsafe_perform_io` somewhere at the top level of our program to get the pure (or “real”) value.

As a result of this refactoring session, we know everything about our code:

- Which parts can fail,
- Which parts are impure,
- How to compose them in a smart, readable, and typesafe manner.

FUTURE CONTAINER

There are several issues with `async` code in Python:

1. You cannot call `async` function from a `sync` one
2. Any unexpectedly thrown exception can ruin your whole event loop
3. Ugly composition with lots of `await` statements

`Future` and `FutureResult` containers solve these issues!

7.1 Mixing `sync` and `async` code

The main feature of `Future` is that it allows to run `async` code while maintaining `sync` context. Let's see an example.

Let's say we have two functions, the `first` one returns a number and the `second` one increments it:

```
async def first() -> int:
    return 1

def second(): # How can we call `first()` from here?
    return first() + 1 # Boom! Don't do this. We illustrate a problem here.
```

If we try to just run `first()`, we will just create an unawaited coroutine. It won't return the value we want.

But, if we would try to run `await first()`, then we would need to change `second` to be `async`. And sometimes it is not possible for various reasons.

However, with `Future` we can “pretend” to call `async` code from `sync` code:

```
from returns.future import Future

def second() -> Future[int]:
    return Future(first()).map(lambda num: num + 1)
```

Without touching our `first` `async` function or making `second` `async` we have achieved our goal. Now, our `async` value is incremented inside a `sync` function.

However, `Future` still requires to be executed inside a proper eventloop:

```
import anyio # or asyncio, or any other lib

# We can then pass our `Future` to any library: asyncio, trio, curio.
# And use any event loop: regular, uvloop, even a custom one, etc
assert anyio.run(second().awaitable) == 2
```

As you can see Future allows you to work with async functions from a sync context. And to mix these two realms together. Use raw Future for operations that cannot fail or raise exceptions. Pretty much the same logic we had with our IO container.

7.2 Async code without exceptions

We have already covered how `Result` <#result-container>`_` works for both pure and impure code. The main idea is: we don't raise exceptions, we return them. It is **especially** critical in async code, because a single exception can ruin all our coroutines running in a single eventloop.

We have a handy combination of Future and Result containers: FutureResult. Again, this is exactly like IOResult, but for impure async code. Use it when your Future might have problems: like HTTP requests or filesystem operations.

You can easily turn any wild throwing coroutine into a calm FutureResult:

```
import anyio
from returns.future import future_safe
from returns.io import IOFailure

@future_safe
async def raising():
    raise ValueError('Not so fast!')

ioresult = anyio.run(raising.awaitable) # all `Future`s return IO containers
assert ioresult == IOFailure(ValueError('Not so fast!')) # True
```

Using FutureResult will keep your code safe from exceptions. You can always `await` or `execute` inside an eventloop any FutureResult to get sync IOResult instance to work with it in a sync manner.

7.3 Better async composition

Previously, you had to do quite a lot of awaiting while writing async code:

```
async def fetch_user(user_id: int) -> 'User':
    ...

async def get_user_permissions(user: 'User') -> 'Permissions':
    ...

async def ensure_allowed(permissions: 'Permissions') -> bool:
    ...

async def main(user_id: int) -> bool:
    # Also, don't forget to handle all possible errors with `try / except`!
    user = await fetch_user(user_id) # We will await each time we use a coro!
    permissions = await get_user_permissions(user)
    return await ensure_allowed(permissions)
```

Some people are ok with it, but some people don't like this imperative style. The problem is that there was no choice. But now, you can do the same thing in functional style! With the help of Future and FutureResult containers:

```

import anyio
from returns.future import FutureResultE, future_safe
from returns.io import IOSuccess, IOFailure

@future_safe
async def fetch_user(user_id: int) -> 'User':
    ...

@future_safe
async def get_user_permissions(user: 'User') -> 'Permissions':
    ...

@future_safe
async def ensure_allowed(permissions: 'Permissions') -> bool:
    ...

def main(user_id: int) -> FutureResultE[bool]:
    # We can now turn `main` into a sync function, it does not `await` at all.
    # We also don't care about exceptions anymore, they are already handled.
    return fetch_user(user_id).bind(get_user_permissions).bind(ensure_allowed)

correct_user_id: int # has required permissions
banned_user_id: int # does not have required permissions
wrong_user_id: int # does not exist

# We can have correct business results:
assert anyio.run(main(correct_user_id).awaitable) == IOSuccess(True)
assert anyio.run(main(banned_user_id).awaitable) == IOSuccess(False)

# Or we can have errors along the way:
assert anyio.run(main(wrong_user_id).awaitable) == IOFailure(
    UserDoesNotExistError(...),
)

```

Or even something really fancy:

```

from returns.pointfree import bind
from returns.pipeline import flow

def main(user_id: int) -> FutureResultE[bool]:
    return flow(
        fetch_user(user_id),
        bind(get_user_permissions),
        bind(ensure_allowed),
    )

```

Later we can also refactor our logical functions to be sync and to return `FutureResult`.

Lovely, isn't it?

MORE!

Want more? Go to the docs! Or read these articles:

- [Python exceptions considered an anti-pattern](#)
- [Enforcing Single Responsibility Principle in Python](#)
- [Typed functional Dependency Injection in Python](#)
- [How Async Should Have Been](#)
- [Higher Kinded Types in Python](#)
- [Make Tests a Part of Your App](#)

Do you have an article to submit? Feel free to open a pull request!

CONTENTS

9.1 Quickstart

Starting is really fast! You can integrate `returns` into any project at any stage. You can use it fully or partially. With or without types.

`returns` is a very flexible library!

You can even just start using it without any deep theory around this project. But, you can always address our learning materials which will unravel all parts of functional programming with useful examples and simple terms.

9.1.1 Why

One of the most frequent questions Python developers ask: why would we need this?

Basically, the answer is that `returns` provides useful abstractions that solve some problems every developer has:

1. *Maybe* helps to work with `None` in a type-safe way
2. *Result* helps to work with exceptions in a type-safe way
3. *IO* helps to separate pure code from impure code to make your architecture better
4. *Future* helps to write `await` free code
5. *RequiresContext* helps to inject dependencies in a very readable, explicit, type-safe, and clean way
6. *Pipelines* can be used independently or together with the types above to create complex, declarative, and type-safe data pipelines

On top of that we provide useful interfaces that allows you to switch implementation on the fly. For example, you can write code that works the same way for sync and async execution flows. While being fully type-safe at the same time.

And you can write your own primitives that will solve any other problem you can possible have based on our existing or your custom interfaces.

In other words, `returns` unlocks insane powers of typed-functional programming to a regular Python developer.

9.1.2 Installation

returns is a pure Python library. Install it as usual:

```
pip install returns # or better use poetry
```

9.1.3 Typechecking and other integrations

This step is optional. If you use `mypy` for type-checking, than you will need to configure it. We really recommend using `mypy` with this project though. Because we have put a lot of efforts into the typing part.

Check out our docs on [mypy](#).

We also have built-in integrations with [pytest](#) and [hypothesis](#). Also, there is [developer tooling](#) you might enjoy.

9.1.4 Theory

Do you want to learn new awesome concepts? Then, start reading our “Userguide”!

It has everything you need! Reading order matters.

However, this is optional. You can still use `returns` without a deep-dive into theory.

9.1.5 Building your own stuff

You can extend `returns` and build your own stuff! Particularly, you can add new interfaces, new containers, and new integrations. See [this guide](#).

Join our chat to get help or advice.

9.2 Container: the concept

Container is a concept that allows you to write code around the existing wrapped values while maintaining the execution context.

List of supported containers:

- [Maybe](#) to handle `None` cases
- [Result](#) to handle possible exceptions
- [IO](#) to mark explicit IO actions
- [Future](#) to work with `async` code
- [RequiresContext](#) to pass context to your functions (DI and similar)

There are also some combinations like [IOResult](#), [FutureResult](#), [RequiresContextResult](#), [RequiresContextIOResult](#) and [RequiresContextFutureResult](#).

We will show you container’s simple API of one attribute and several simple methods.

9.2.1 Basics

The main idea behind a container is that it wraps some internal state. That's what `._inner_value` is used for.

And we have several functions to create new containers based on the previous state. And we can see how this state is evolving during the execution.

Fig. 1: State evolution.

9.2.2 Working with a container

We use two methods to create a new container from the previous one. `bind` and `map`.

The difference is simple:

- `map` works with functions that return regular value
- `bind` works with functions that return new container of the same type

We have `returns.interfaces.mappable.MappableN.map()` to compose containers with regular functions.

Here's how it looks:

Fig. 2: Illustration of map method.

```
>>> from typing import Any
>>> from returns.result import Success, Result

>>> def double(state: int) -> int:
...     return state * 2

>>> result: Result[int, Any] = Success(1).map(double)
>>> assert str(result) == '<Success: 2>'

>>> result: Result[int, Any] = result.map(lambda state: state + 1)
>>> assert str(result) == '<Success: 3>'
```

The same works with built-in functions as well:

```
>>> from returns.io import IO

>>> io = IO('bytes').map(list)
>>> str(io)
"<IO: ['b', 'y', 't', 'e', 's']>"
```

The second method is `bind`. It is a bit different. We pass a function that returns another container to it. `returns.interfaces.bindable.BindableN.bind()` is used to literally bind two different containers together.

Here's how it looks:

Fig. 3: Illustration of bind method.

```
from returns.result import Result, Success

def may_fail(user_id: int) -> Result[float, str]:
    ...

value: Result[int, str] = Success(1)
# Can be assumed as either Success[float] or Failure[str]:
result: Result[float, str] = value.bind(may_fail)
```

Note: All containers support these methods. Because all containers implement `returns.interfaces.mappable.MappableN` and `returns.interfaces.bindable.BindableN`.

You can read more about methods that some other containers support and *interfaces* behind them.

9.2.3 Instantiating a container

All `returns.interfaces applicative.ApplicativeN` containers support special `.from_value` method to construct a new container from a raw value.

```
>>> from returns.result import Result
>>> assert str(Result.from_value(1)) == '<Success: 1>'
```

There are also other methods in other interfaces. For example, here are some of them:

- `returns.interfaces.specific.maybe.MaybeLikeN.from_optional()` creates a value from `Optional` value

```
>>> from returns.maybe import Maybe, Some, Nothing
>>> assert Maybe.from_optional(1) == Some(1)
>>> assert Maybe.from_optional(None) == Nothing
```

- `returns.interfaces.failable.DiverseFailableN.from_failure()` creates a failing container from a value

```
>>> from returns.result import Result, Failure
>>> assert Result.from_failure(1) == Failure(1)
```

There are many other constructors! Check out concrete types and their interfaces.

9.2.4 Working with multiple containers

Multiple container arguments

We have already seen how we can work with one container and functions that receive a single argument.

Let's say you have a function of two arguments and two containers:

```
>>> def sum_two_numbers(first: int, second: int) -> int:
...     return first + second
```

And here are our two containers:

```
>>> from returns.io import IO

>>> one = IO(1)
>>> two = IO(2)
```

The naive approach to compose two IO containers and a function would be too hard to show here. Luckily, we support partial application and the `.apply()` method.

Here are the required steps:

0. We make `sum_two_numbers` to receive *partial arguments*
1. We create a new container that wraps `sum_two_numbers` function as a value
2. We then call `.apply()` twice to pass each value

It can be done like so:

```
>>> from returns.curry import curry
>>> from returns.io import IO

>>> @curry
... def sum_two_numbers(first: int, second: int) -> int:
...     return first + second

>>> one = IO(1)
>>> two = IO(2)
>>> assert two.apply(one.apply(IO(sum_two_numbers))) == IO(3)
```

But, there are other ways to make `sum_two_numbers` partial. One can use `partial` as well:

```
>>> from returns.curry import partial

>>> one = IO(1)
>>> two = IO(2)
>>> assert two.apply(one.apply(
...     IO(lambda x: partial(sum_two_numbers, x)),
... )) == IO(3)
```

Or even native lambda functions:

```
>>> one = IO(1)
>>> two = IO(2)
>>> assert two.apply(one.apply(
...     IO(lambda x: lambda y: sum_two_numbers(x, y)),
... )) == IO(3)
```

It would be faster, but not as elegant (and type-safe).

Working with iterable of containers

Imagine that you have to take 10 random numbers and then sum them to get the final result.

So, here's how your code will look like:

```
>>> import random
>>> from returns.io import IO

>>> def random_number() -> IO[int]:
...     return IO(2) # Example, basically alias of `random.randint(1, 5)`

>>> numbers = [random_number() for _ in range(10)]
>>> assert len(numbers) == 10
>>> assert all(isinstance(number, IO) for number in numbers)
```

So, how to sum these random values into a single IO[int] value? That's where *Fold.loop* really helps!

```
>>> from typing import Callable
>>> from returns.iterables import Fold

>>> def sum_two_numbers(first: int) -> Callable[[int], int]:
...     return lambda second: first + second

>>> assert Fold.loop(
...     numbers, # let's loop on our `IO` values
...     IO(0), # starting from `0` value
...     sum_two_numbers, # and getting the sum of each two numbers in a loop
... ) == IO(20)
```

We can also change the initial element to some other value:

```
>>> assert Fold.loop(
...     numbers,
...     IO(5), # now we will start from `5`, not `0`
...     sum_two_numbers,
... ) == IO(25)
```

Fold.loop is eager. It will be executed for all items in your iterable.

Collecting an iterable of containers into a single container

You might end up with an iterable of containers:

```
>>> from typing import List
>>> from returns.maybe import Maybe, Some, Nothing, maybe

>>> source = {'a': 1, 'b': 2}
>>> fetched_values: List[Maybe[int]] = [
...     maybe(source.get)(key)
...     for key in ('a', 'b')
... ]
```

To work with iterable of containers, it is recommended to cast it into a container with the iterable inside using the *Fold.collect* method:

```
>>> from returns.iterables import Fold
>>> assert Fold.collect(fetched_values, Some(())) == Some((1, 2))
```

Any falsy values will result in a falsy result (pun intended):

```
>>> fetched_values: List[Maybe[int]] = [
...     maybe(source.get)(key)
...     for key in ('a', 'c') # 'c' is missing!
... ]
>>> assert Fold.collect(fetched_values, Some(())) == Nothing
```

You can also use a different strategy to fetch values you need, to do just that we have `Fold.collect_all` method:

```
>>> fetched_values: Maybe[int] = [
...     maybe(source.get)(key)
...     for key in ('a', 'c') # 'c' is missing!
... ]
>>> assert Fold.collect_all(fetched_values, Some(())) == Some((1,))
```

We support any `Iterable[T]` input type and return a `Container[Sequence[T]]`.

You can subclass `Fold` type to change how any of these methods work.

9.2.5 Immutability

We like to think of `returns` as *immutable* structures. You cannot mutate the inner state of the created container, because we redefine `__setattr__` and `__delattr__` magic methods.

You cannot also set new attributes to container instances, since we are using `__slots__` for better performance and strictness.

Well, nothing is **really** immutable in python, but you were warned.

We also provide `returns.primitives.types.Immutable` mixin that users can use to quickly make their classes immutable.

9.2.6 Type safety

We try to make our containers optionally type safe.

What does it mean?

1. It is still good old python, do whatever you want without mypy
2. If you are using mypy you will be notified about type violations

We also ship PEP561 compatible `.pyi` files together with the source code. In this case these types will be available to users when they install our application.

We also ship custom mypy plugins to overcome some existing problems, please make sure to use them, since they increase your developer experience and type-safety level:

Check out our docs on using our *mypy plugins*.

9.2.7 Further reading

- *Railway oriented programming*

9.2.8 API Reference

`BaseContainer` is a base class for all other containers. It defines some basic things like representation, hashing, pickling, etc.

class `BaseContainer`(*inner_value*)

Bases: `Immutable`

Utility class to provide all needed magic methods to the context.

`__slots__` = ('_inner_value',)

`__init__`(*inner_value*)

Wraps the given value in the Container.

'value' is any arbitrary value of any type including functions.

`__repr__`()

Used to display details of object.

Return type

`str`

`__eq__`(*other*)

Used to compare two 'Container' objects.

Parameters

other (Any) –

Return type

`bool`

`__hash__`()

Used to use this value as a key.

Return type

`int`

`__getstate__`()

That's how this object will be pickled.

Return type

`_PickleState`

`__setstate__`(*state*)

Loading state from pickled data.

Parameters

state (Union[_PickleState, Any]) –

Return type

`None`

`__abstractmethods__` = `frozenset({})`


```
__annotations__ = {'_inner_value': typing.Any}
__module__ = 'returns.primitives.container'
```

container_equality(*self*, *other*)

Function to compare similar containers.

Compares both their types and their inner values.

Parameters

- **self** (*KindN*[*TypeVar*(*_EqualType*, bound= *Equable*), Any, Any, Any]) –
- **other** (*KindN*[*TypeVar*(*_EqualType*, bound= *Equable*), Any, Any, Any]) –

Return type

bool

9.3 Railway oriented programming

Containers can serve many different purposes (while still serving the main one: composition) for example, some of them (*Result* and *Maybe*) are used to work with different types of errors starting with *NullPointerException* to arbitrary user-defined ones.

9.3.1 Error handling

When talking about error handling we use a concept of *Railway oriented programming*. It means that flow of our program has two tracks:

1. Successful one: where everything goes perfectly: HTTP requests work, database is always serving us data, parsing values does not fail
2. Failed one: where something went wrong

We can switch from track to track: we can fail something or we can fix the situation.

Fig. 4: Railway oriented programming.

Returning execution to the right track

We also support two special methods to work with “failed” values:

- *returns.interfaces.altable.AtableN.alt()* transforms error to another error that works only when container is in failed state, is the opposite of *returns.interfaces.mappable.MappableN.map()* method
- *returns.interfaces.lashable.LashableN.lash()* is the opposite of *returns.interfaces.bindable.BindableN.bind()* method that works only when container is in failed state

Let’s start from the first one: *alt* method allows to change your error type.

Fig. 5: Illustration of *alt* method.

```
>>> from returns.result import Failure
>>> assert Failure(1).alt(str) == Failure('1')
```

The second method is `lash`. It is a bit different. We pass a function that returns another container to it. `returns.interfaces.lashable.LashableN.lash()` is used to literally bind two different containers together. It can also lash your flow and get on the successful track again:

Fig. 6: Illustration of `lash` method.

```
>>> from returns.result import Result, Failure, Success

>>> def tolerate_exception(state: Exception) -> Result[int, Exception]:
...     if isinstance(state, ZeroDivisionError):
...         return Success(0)
...     return Failure(state)

>>> value: Result[int, Exception] = Failure(ZeroDivisionError())
>>> result: Result[int, Exception] = value.lash(tolerate_exception)
>>> assert result == Success(0)

>>> value2: Result[int, Exception] = Failure(ValueError())
>>> result2: Result[int, Exception] = value2.lash(tolerate_exception)
>>> # => Failure(ValueError())
```

From typing perspective `.alt` and `.lash` are exactly the same as `.map` and `.bind` but only work with the second type argument instead of the first one:

```
from returns.result import Result

first: Result[int, int]
second: Result[int, int]

reveal_type(first.map(str))
# => Result[str, int]

reveal_type(second.alt(str))
# => Result[int, str]
```

Note: Not all containers support these methods, only containers that implement `returns.interfaces.lashable.LashableN` and `returns.interfaces.alttable.AlttableN`. For example, `IO` based containers and `RequiresContext` cannot be `alted` or `lashed`.

9.3.2 Unwrapping values

And we have two more functions to unwrap inner state of containers into a regular types:

- `.unwrap` returns a value if it is possible, raises `returns.primitives.exceptions.UnwrapFailedError` otherwise

```
>>> from returns.result import Failure, Success
>>> from returns.maybe import Some, Nothing

>>> assert Success(1).value_or(None) == 1
>>> assert Some(0).unwrap() == 0
```

```
>>> Failure(1).unwrap()
Traceback (most recent call last):
...
returns.primitives.exceptions.UnwrapFailedError

>>> Nothing.unwrap()
Traceback (most recent call last):
...
returns.primitives.exceptions.UnwrapFailedError
```

For failing containers you can use `returns.interfaces.unwrapable.Unwrapable.failure()` to unwrap the failed state:

```
>>> assert Failure(1).failure() == 1
>>> Success(1).failure()
Traceback (most recent call last):
...
returns.primitives.exceptions.UnwrapFailedError
```

Be careful, since this method will raise an exception when you try to `.failure()` a successful container.

Note: Not all containers support these methods, only containers that implement `returns.interfaces.unwrapable.Unwrapable`. For example, *IO* based containers and *RequiresContext* cannot be unwrapped.

Note: Some containers also have `.value_or()` helper method. Example:

```
>>> from returns.result import Success, Failure
>>> assert Success(1).value_or(None) == 1
>>> assert Failure(1).value_or(None) is None
```

9.3.3 Further reading

- Railway oriented programming in F#
- Against Railway-Oriented Programming

9.4 Higher Kinded Types

Higher Kinded Types is a new concept for Python developers. But, it is totally not new in general!

So, let's start with the detailed explanation: what Higher Kinded Types are?

9.4.1 Regular types

We can start with the very basic example. Let's say we have a function that transforms type A into a type B. These types A and B can be some specific ones, for example:

```
>>> def from_a_to_b(arg: int) -> str:
...     return str(arg)

>>> assert from_a_to_b(1) == '1'
```

That's what we already know and use. Let's scale things up!

9.4.2 Generics

The next thing we can do with types is to write generic types. What are generic types? Basically, they are some types that contain other types inside. Like `List[int]` is a list of integers: `[1, 2, 3]`. We know that `List[int]` has a shape of a list and contents of `int`.

We can also write functions that work with generics:

```
>>> from typing import List

>>> def all_to_str(arg: List[int]) -> List[str]:
...     return [str(item) for item in arg]

>>> assert all_to_str([1, 2]) == ['1', '2']
```

There's one more thing about generics we want to notice at this point. Different generics do have different numbers of type arguments:

- `List` has a single type argument: `List[Value]` or `Maybe[Value]`
- `Dict` has two type arguments: `Dict[Key, Value]` or `Result[Value, Error]`
- `Generator` has three type arguments: `Generator[Yield, Send, Return]` or `RequiresContextResult[Value, Error, Env]`

That's what we call a kind. So, `List` and `Maybe` have a kind of 1, `Dict` and `Result` have kind of 2, `Generator` and `RequiresContextResult` have a kind of 3.

So, let's go one level further.

9.4.3 Operations on generics

Let's say you have a function that copies all values of a passed argument. We can define this function as:

```
>>> from typing import TypeVar
>>> ValueType = TypeVar('ValueType')

>>> def copy(arg: ValueType) -> ValueType:
...     ...
```

This function can work with any type. It receives something and then returns the same value back. That's the whole point of copying!

But, there are different functions, that do different things with types. For example, we can write a function that converts a value inside any *Container1* (a base class for all our containers) from `int` to `str`:

We can also write functions that work with generics:

```
>>> from returns.interfaces.container import Container1
>>> def to_str(container: Container1[int]) -> Container1[str]:
...     return container.map(str)
```

And here's how it can be used:

```
>>> from returns.maybe import Maybe
>>> from returns.io import IO

>>> assert to_str(Maybe.from_value(1)) == Maybe.from_value('1')
>>> assert to_str(IO.from_value(1)) == IO.from_value('1')
```

It works just fine! But! It has a very important thing inside. All calls to `to_str` will return `Container1` type, not something specific:

```
reveal_type(to_str(Maybe.from_value(1))) # Container1[str]
reveal_type(to_str(IO.from_value(1)))   # Container1[str]
```

But, we know that this is not true. When we pass a `Maybe` in - we get the `Maybe` back. When we pass a `IO` in - we get the `IO` back.

How can we fix this problem? With `@overload`!

```
>>> from typing import overload
>>> from returns.maybe import Maybe
>>> from returns.io import IO

>>> @overload
... def to_str(arg: Maybe[int]) -> Maybe[str]:
...     ...

>>> @overload
... def to_str(arg: IO[int]) -> IO[str]:
...     ...
```

We kinda fixed it! Now, our calls will reveal the correct types for these three examples:

```
reveal_type(to_str(Maybe.from_value(1))) # Maybe[str]
reveal_type(to_str(IO.from_value(1)))   # IO[str]
```

But, there's an important limitation with this solution: no other types are allowed in this function anymore. So, you will try to use it with any other type, it won't be possible.

9.4.4 Current limitations

To overcome current `@overload` decorators limitations, we can imagine a syntax like this:

```
from typing import TypeVar
from returns.interfaces.container import Container1

T = TypeVar('T', bound=Container1)

def all_to_str(arg: T[int]) -> T[str]:
    ...
```

Sadly, this does not work. Because `TypeVar` cannot be used with `[]`. We have to find some other way.

9.4.5 Higher Kinded Types

So, that's where `returns` saves the day!

Note: Technical note: this feature requires *mypy plugin*.

The main idea is that we can rewrite `T[int]` as `Kind1[T, int]`. Let's see how it works:

```
>>> from returns.primitives.hkt import Kind1
>>> from returns.interfaces.container import ContainerN
>>> from typing import TypeVar

>>> T = TypeVar('T', bound=ContainerN)

>>> def to_str(container: Kind1[T, int]) -> Kind1[T, str]:
...     return container.map(str)
```

Now, this will work almost correctly! Why almost? Because the revealed type will be `Kind1`.

```
reveal_type(to_str(Maybe.from_value(1))) # Kind1[Maybe, str]
reveal_type(to_str(IO.from_value(1)))   # Kind1[IO, str]
```

That's not something we want. We don't need `Kind1`, we need real `Maybe` or `IO` values.

The final solution is to decorate `to_str` with `@kinded`:

```
>>> from returns.primitives.hkt import kinded

>>> @kinded
... def to_str(container: Kind1[T, int]) -> Kind1[T, str]:
...     return container.map(str)
```

Now, it will be fully working:

```
reveal_type(to_str(Maybe.from_value(1))) # Maybe[str]
reveal_type(to_str(IO.from_value(1)))   # IO[str]
```

And the thing about this approach is that it will be:

1. Fully type-safe. It works with correct interface `ContainerN`, returns the correct type, has correct type transformation
2. Is opened for further extension and even custom types

9.4.6 Kinds

As it was said `Maybe[int]`, `Result[str, int]`, and `RequiresContextResult[str, int, bool]` are different in terms of a number of type arguments. We support different kinds:

- `Kind1[Maybe, int]` is similar to `Maybe[int]`
- `Kind2[Result, str, int]` is similar to `Result[str, int]`
- `Kind3[RequiresContextResult, str, int, bool]` is similar to `RequiresContextResult[str, int, bool]`

You can use any of them freely.

Later you will learn how to *create your own types* that support kinds!

9.4.7 Further reading

- [Higher Kinded Types in Python](#)

9.4.8 FAQ

Which types you can use with KindN?

The first position in all `KindN` types can be occupied by either `Instance` type or `TypeVar` with `bound=`.

Let's see an example:

```
>>> from typing import TypeVar
>>> from returns.primitives.hkt import KindN, kinded
>>> from returns.interfaces.mappable import MappableN

>>> _FirstType = TypeVar('_FirstType')
>>> _SecondType = TypeVar('_SecondType')
>>> _ThirdType = TypeVar('_ThirdType')
>>> _MappableKind = TypeVar('_MappableKind', bound=MappableN)

>>> @kinded
... def works_with_interface(
...     container: KindN[_MappableKind, _FirstType, _SecondType, _ThirdType],
... ) -> KindN[_MappableKind, str, _SecondType, _ThirdType]:
...     return container.map(str)
```

This version of `works_with_interface` will work with any subtype of `MappableN`. Because we use `_MappableKind` in its definition. And `_MappableKind` is a `TypeVar` bound to `MappableN`. Arguments of non `MappableN` subtypes will be rejected by a type-checker:

```
>>> from returns.maybe import Maybe
>>> from returns.io import IO
>>> from returns.result import Success

>>> assert works_with_interface(Maybe.from_value(1)) == Maybe.from_value('1')
>>> assert works_with_interface(IO.from_value(1)) == IO.from_value('1')
>>> assert works_with_interface(Success(1)) == Success('1')
```

In contrast, we can work directly with some specific type, let's say Maybe container:

```
>>> from returns.maybe import Maybe

>>> @kinded
... def works_with_maybe(
...     container: KindN[Maybe, _FirstType, _SecondType, _ThirdType],
... ) -> KindN[Maybe, str, _SecondType, _ThirdType]:
...     return container.map(str)

>>> assert works_with_maybe(Maybe.from_value(1)) == Maybe.from_value('1')
```

Function `works_with_maybe` will work correctly with Maybe instance. Other types will be rejected.

So, choose wisely which mechanism you need.

9.4.9 API Reference

class KindN

Bases: `Generic[_InstanceType, _TypeArgType1, _TypeArgType2, _TypeArgType3]`

Emulation support for Higher Kinded Types.

Consider KindN to be an alias of Generic type. But with some extra goodies.

KindN is the top-most type for other Kind types like Kind1, Kind2, Kind3, etc.

The only difference between them is how many type arguments they can hold. Kind1 can hold just two type arguments: `Kind1[IO, int]` which is almost equals to `IO[int]`. Kind2 can hold just two type arguments: `Kind2[IOResult, int, str]` which is almost equals to `IOResult[int, str]`. And so on.

The idea behind KindN is that one cannot write this code:

```
from typing import TypeVar

T = TypeVar('T')
V = TypeVar('V')

def impossible(generic: T, value: V) -> T[V]:
    return generic(value)
```

But, with KindN this becomes possible in a form of `Kind1[T, V]`.

Note: To make sure it works correctly, your type has to be a subtype of KindN.

We use a custom mypy plugin to make sure types are correct. Otherwise, it is currently impossible to properly type this.

We use “emulated Higher Kinded Types” concept. Read the whitepaper: <https://bit.ly/2ABACx2>

`KindN` does not exist in runtime. It is used just for typing. There are (and must be) no instances of this type directly.

Implementation details

We didn’t use `ABCMeta` to disallow its creation, because we don’t want to have a possible metaclass conflict with other metaclasses. Current API allows you to mix `KindN` anywhere.

We allow `_InstanceType` of `KindN` to be `Instance` type or `TypeVarType` with `bound=...`

See also:

- <https://arrow-kt.io/docs/0.10/patterns/glossary/#higher-kinds>
- <https://github.com/gcanti/fp-ts/blob/master/docs/guides/HKT.md>
- <https://bow-swift.io/docs/fp-concepts/higher-kinded-types>
- <https://github.com/pelotom/hkts>

Kind1

Type alias for kinds with one type argument.

alias of `KindN[_InstanceType, _TypeArgType1, Any, Any]`

Kind2

Type alias for kinds with two type arguments.

alias of `KindN[_InstanceType, _TypeArgType1, _TypeArgType2, Any]`

Kind3

Type alias for kinds with three type arguments.

alias of `KindN[_InstanceType, _TypeArgType1, _TypeArgType2, _TypeArgType3]`

class SupportsKindN

Bases: `KindN[_InstanceType, _TypeArgType1, _TypeArgType2, _TypeArgType3]`

Base class for your containers.

Notice, that we use `KindN / Kind1` to annotate values, but we use `SupportsKindN / SupportsKind1` to inherit from.

Implementation details

The only thing this class does is: making sure that the resulting classes won’t have `__getattr__` available during the typechecking phase.

Needless to say, that `__getattr__` during runtime - never exists at all.

SupportsKind1

Type alias used for inheritance with one type argument.

alias of `SupportsKindN[_InstanceType, _TypeArgType1, NoReturn, NoReturn]`

SupportsKind2

Type alias used for inheritance with two type arguments.

alias of `SupportsKindN[_InstanceType, _TypeArgType1, _TypeArgType2, NoReturn]`

SupportsKind3

Type alias used for inheritance with three type arguments.

alias of `SupportsKindN[_InstanceType, _TypeArgType1, _TypeArgType2, _TypeArgType3]`

dekind(*kind*)

Turns `Kind1[IO, int]` type into real `IO[int]` type.

Should be used when you are left with accidental `KindN` instance when you really want to have the real type.

Works with type arguments of any length.

We use a custom mypy plugin to make sure types are correct. Otherwise, it is currently impossible to properly type this.

In runtime it just returns the passed argument, nothing really happens:

```
>>> from returns.io import IO
>>> from returns.primitives.hkt import Kind1

>>> container: Kind1[IO, int] = IO(1)
>>> assert dekind(container) is container
```

However, please, do not use this function unless you know exactly what you are doing and why do you need it.

Parameters

kind (`KindN[TypeVar(_InstanceType, covariant=True), TypeVar(_TypeArgType1, covariant=True), TypeVar(_TypeArgType2, covariant=True), TypeVar(_TypeArgType3, covariant=True)]`) –

Return type

`TypeVar(_InstanceType, covariant=True)`

class Kinded(*args, **kwargs)

Bases: `Protocol[_FunctionDefType]`

Protocol that tracks kinded functions calls.

We use a custom mypy plugin to make sure types are correct. Otherwise, it is currently impossible to properly type this.

kinded(*function*)

Decorator to be used when you want to dekind the function's return type.

Does nothing in runtime, just returns its argument.

We use a custom mypy plugin to make sure types are correct. Otherwise, it is currently impossible to properly type this.

Here's an example of how it should be used:

```
>>> from typing import TypeVar
>>> from returns.primitives.hkt import KindN, kinded
>>> from returns.interfaces.bindable import BindableN

>>> _Binds = TypeVar('_Binds', bound=BindableN) # just an example
```

(continues on next page)

(continued from previous page)

```

>>> _Type1 = TypeVar('_Type1')
>>> _Type2 = TypeVar('_Type2')
>>> _Type3 = TypeVar('_Type3')

>>> @kinded
... def bindable_identity(
...     container: KindN[_Binds, _Type1, _Type2, _Type3],
... ) -> KindN[_Binds, _Type1, _Type2, _Type3]:
...     return container # just do nothing

```

As you can see, here we annotate our return type as `-> KindN[_Binds, _Type1, _Type2, _Type3]`, it would be true without `@kinded` decorator.

But, `@kinded` decorator dekindes the return type and infers the real type behind it:

```

>>> from returns.io import IO, IOResult

>>> assert bindable_identity(IO(1)) == IO(1)
>>> # => Revealed type: 'IO[int]'

>>> iores: IOResult[int, str] = IOResult.from_value(1)
>>> assert bindable_identity(iores) == iores
>>> # => Revealed type: 'IOResult[int, str]'

```

The difference is very clear in methods modules, like:

- Raw `returns.methods.bind.internal_bind()` that returns `KindN` instance
- User-facing `returns.methods.bind.bind()` that returns the container type

You must use this decorator for your own kinded functions as well.

Parameters

`function` (`TypeVar(_FunctionType, bound= Callable)`) –

Return type

`Kinded`[`TypeVar(_FunctionType, bound= Callable)`]

9.5 Interfaces

We provide a lot of generic interfaces to write our bundled and your own custom types.

These interfaces are designed:

1. To be subclassed
2. To provide abstract methods to implement in your own types
3. To enforce correctness on final types
4. To attach critical laws to be checked

We use *Higher Kinded Types* to define abstract methods.

Reading about interfaces will be the most useful if you plan to *create your own containers*.

9.5.1 General information

All the non-specific interfaces (e.g. `MappableN`, `BindableN`, `ApplicativeN`) can have **Nth** types, at the maximum of three possible types. What does this mean?

`MappableN` interface, for example, can have one, two or three possible types. See the example below:

```
>>> from typing import NoReturn

>>> from returns.interfaces.mappable import (
...     MappableN, Mappable1, Mappable2, Mappable3,
... )

>>> one_type: MappableN[int, NoReturn, NoReturn]
>>> two_types: MappableN[int, str, NoReturn]
>>> three_types: MappableN[int, str, bool]

>>> # We have a shortcut for each amount of arguments to reduce the boilerplate
>>> one_type: Mappable1[int]
>>> two_types: Mappable2[int, str]
>>> three_type: Mappable3[int, str, bool]
```

Note: Useful links before you start here:

- [Functors, Applicatives, And Monads In Pictures](#)
 - [Understanding Functor and Monad With a Bag of Peanuts](#)
 - [Variance of generic types](#)
 - [If you know map, I will teach you monads](#)
-

Naming convention

We follow a very specific naming convention in our interface names.

If interface does not depend on the number of types it works with and is always the same, we name it as is. For example, `Equable` is always the same and does not depend on the number of type arguments. We use adjectives to name these interfaces.

Secondly, if interface depends on the number of type arguments, it is named with `N` suffix in the end. It would always have numeric aliases for each number of arguments supported. For example, `MappableN`, `Mappable1`, `Mappable2`, and `Mappable3`.

The last criteria we have to decided on naming is “whether this interface always the same or it can have slight variations”? That’s why we have `ResultLikeN` and `ResultBasedN` interfaces. Because `ResultBasedN` has two extra methods compared to `ResultLikeN`. We use `Like` suffix for interfaces that describes some similar types. We use `Based` suffix for interfaces that descire almost concrete types.

Laws

Some interfaces define its laws as values. These laws can be viewed as tests that are attached to the specific interface.

We are able to check them of any type that implements a given interfaces with laws by our own *check_all_laws hypothesis plugin*.

In this docs we are going to describe each general interface and its laws.

9.5.2 Mappable

Something is considered mappable if we can map it using a function, generally map is a method that accepts a function.

An example in this library is *Maybe*, that implements the Mappable interface:

```
>>> from returns.maybe import Maybe, Some

>>> def can_be_mapped(string: str) -> str:
...     return string + '!'

>>> maybe_str: Maybe[str] = Some('example')
>>> assert maybe_str.map(can_be_mapped) == Some('example!')
```

MappableN interface helps us to create our own mappable container like *Maybe*.

```
>>> from typing import Callable, TypeVar

>>> from returns.interfaces.mappable import Mappable1
>>> from returns.primitives.hkt import SupportsKind1
>>> from returns.primitives.container import BaseContainer

>>> _NumberType = TypeVar('_NumberType')
>>> _NewNumberType = TypeVar('_NewNumberType')

>>> class Number(
...     BaseContainer,
...     SupportsKind1['Number', _NumberType],
...     Mappable1[_NumberType],
... ):
...     def __init__(self, inner_value: _NumberType) -> None:
...         super().__init__(inner_value)
...
...     def map( # This method is required by Mappable
...         self,
...         function: Callable[[_NumberType], _NewNumberType]
...     ) -> 'Number[_NewNumberType]':
...         return Number(function(self._inner_value))
```

With our *Number* mappable class we can compose easily math functions with it.

```
>>> def my_math_function(number: int) -> int:
...     return number - 1

>>> number: Number[int] = Number(-41)
>>> assert number.map(my_math_function).map(abs) == Number(42)
```

Laws

To make sure your Mappable implementation is right, you can apply the Mappable laws on it to test.

1. *Identity Law*: When we pass the identity function to the map method, the Mappable instance has to be the same, unchanged.

```
>>> from returns.functions import identity
>>> mappable_number: Number[int] = Number(1)
>>> assert mappable_number.map(identity) == Number(1)
```

2. *Associative Law*: Given two functions, x and y, calling the map method with x function and after that calling with y function must have the same result if we compose them together.

```
>>> from returns.functions import compose
>>> def add_one(number: int) -> int:
...     return number + 1
>>> def multiply_by_ten(number: int) -> int:
...     return number * 10
>>> mappable_number: Number[int] = Number(9)
>>> assert mappable_number.map(
...     add_one,
... ).map(
...     multiply_by_ten,
... ) == mappable_number.map(
...     compose(add_one, multiply_by_ten),
... )
```

9.5.3 Bindable

Bindable is something that we can bind with a function. Like *Maybe*, so *BindableN* interface will help us to create our custom bindable.

```
>>> from typing import Callable, TypeVar
>>> from returns.interfaces.bindable import Bindable1
>>> from returns.primitives.hkt import SupportsKind1, Kind1, dekind
>>> from returns.primitives.container import BaseContainer
>>> _NumberType = TypeVar('_NumberType')
>>> _NewNumberType = TypeVar('_NewNumberType')
>>> class Number(
...     BaseContainer,
...     SupportsKind1['Number', _NumberType],
...     Bindable1[_NumberType],
... ):
...     def __init__(self, inner_value: _NumberType) -> None:
...         super().__init__(inner_value)
```

(continues on next page)

(continued from previous page)

```

...
...     def bind( # This method is required by Bindable
...         self,
...         function: Kind1[
...             'Number',
...             Callable[[_NumberType], 'Number[_NewNumberType]'],
...         ],
...     ) -> 'Number[_NewNumberType]':
...         return dekind(function(self._inner_value))

```

And here's how we can use it:

```

>>> def double(arg: int) -> Number[int]:
...     return Number(arg * 2)

>>> number = Number(5)
>>> assert number.bind(double) == Number(10)

```

9.5.4 Applicative

Something is considered applicative if it is a functor already and, moreover, we can apply another container to it and construct a new value with `.from_value` method.

An example in this library is *Maybe*, that implements the *Mappable* and *Applicative* interfaces:

```

>>> from returns.maybe import Maybe, Some

>>> maybe_str = Maybe.from_value('example')
>>> maybe_func = Maybe.from_value(len) # we use function as a value!

>>> assert maybe_str.apply(maybe_func) == Some(7)

```

As you see, `apply` takes a container with a function inside and applies it to the current value inside the container.

This way we really execute `Maybe.from_value(len('example'))`.

ApplicativeN which is a subtype of *MappableN* interface helps us to create our own applicative container like *Maybe*.

```

>>> from typing import Callable, TypeVar

>>> from returns.interfaces.applicative import Applicative1
>>> from returns.primitives.hkt import SupportsKind1, Kind1, dekind
>>> from returns.primitives.container import BaseContainer

>>> _NumberType = TypeVar('_NumberType')
>>> _NewNumberType = TypeVar('_NewNumberType')

>>> class Number(
...     BaseContainer,
...     SupportsKind1['Number', _NumberType],
...     Applicative1[_NumberType],
... ):
...     def __init__(self, inner_value: _NumberType) -> None:

```

(continues on next page)

```

...     super().__init__(inner_value)
...
...     def map( # This method is required by Mappable
...             self,
...             function: Callable[[_NumberType], _NewNumberType]
...         ) -> 'Number[_NewNumberType]':
...             return Number(function(self._inner_value))
...
...     def apply( # This method is required by Applicative
...                self,
...                container: Kind1[
...                    'Number',
...                    Callable[[_NumberType], _NewNumberType],
...                ],
...            ) -> 'Number[_NewNumberType]':
...                return Number.from_value(
...                    dekind(container._inner_value(self._inner_value)),
...                )
...
...     @classmethod
...     def from_value( # This method is required by Applicative
...                     cls,
...                     inner_value: _NewNumberType,
...                 ) -> 'Number[_NewNumberType]':
...                     return Number(inner_value)

```

With our `Number` mappable class we can compose easily math functions with it.

```

>>> def my_math_function(number: int) -> int:
...     return number - 1

>>> number = Number(3)
>>> number_function = Number.from_value(my_math_function)

>>> assert number.apply(number_function) == Number(2)

```

Laws

To make sure your `Applicative` implementation is right, you can apply the `Applicative` laws on it to test.

1. *Identity Law*: When we pass an applicative instance with wrapped identity function to the `apply` method, the `Applicative` has to be the same, unchanged.

```

>>> from returns.functions import identity

>>> applicative_number: Number[int] = Number(1)
>>> assert applicative_number.apply(
...     applicative_number.from_value(identity),
... ) == Number(1)

```

2. *Interchange Law*: We can start our composition with both raw value and a function.


```

>>> def function(arg: int) -> int:
...     return arg + 1

>>> raw_value = 5

>>> assert Number.from_value(raw_value).apply(
...     Number.from_value(function),
... ) == Number.from_value(function).apply(
...     Number.from_value(lambda inner: inner(raw_value)),
... )

```

3. *Homomorphism Law*: The homomorphism law says that applying a wrapped function to a wrapped value is the same as applying the function to the value in the normal way and then using `.from_value` on the result.

```

>>> def function(arg: int) -> int:
...     return arg + 1

>>> raw_value = 5

>>> assert Number.from_value(
...     function(raw_value),
... ) == Number.from_value(raw_value).apply(
...     Number.from_value(function),
... )

```

4. *Composition Law*: Applying two functions twice is the same as applying their composition once.

```

>>> from returns.functions import compose

>>> def first(arg: int) -> int:
...     return arg * 2

>>> def second(arg: int) -> int:
...     return arg + 1

>>> instance = Number(5)
>>> assert instance.apply(
...     Number.from_value(compose(first, second)),
... ) == instance.apply(
...     Number.from_value(first),
... ).apply(
...     Number.from_value(second),
... )

```

Plus all laws from `MappableN` interface.

9.5.5 Container

ContainerN is a central piece of our library. It is an interface that combines *ApplicativeN* and *BindableN* together.

So, in other words: Container is an *Applicative* that you can bind!

```
>>> from typing import Callable, TypeVar

>>> from returns.interfaces.container import Container1
>>> from returns.primitives.hkt import SupportsKind1, Kind1, dekind
>>> from returns.primitives.container import BaseContainer

>>> _NumberType = TypeVar('_NumberType')
>>> _NewNumberType = TypeVar('_NewNumberType')

>>> class Number(
...     BaseContainer,
...     SupportsKind1['Number', _NumberType],
...     Container1[_NumberType],
... ):
...     def __init__(self, inner_value: _NumberType) -> None:
...         super().__init__(inner_value)
...
...     def map( # This method is required by Mappable
...         self,
...         function: Callable[[_NumberType], _NewNumberType]
...     ) -> 'Number[_NewNumberType]':
...         return Number(function(self._inner_value))
...
...     def bind( # This method is required by Bindable
...         self,
...         function: Kind1[
...             'Number',
...             Callable[[_NumberType], 'Number[_NewNumberType]'],
...         ],
...     ) -> 'Number[_NewNumberType]':
...         return dekind(function(self._inner_value))
...
...     def apply( # This method is required by Applicative
...         self,
...         container: Kind1[
...             'Number',
...             Callable[[_NumberType], _NewNumberType],
...         ],
...     ) -> 'Number[_NewNumberType]':
...         return Number.from_value(
...             container._inner_value(self._inner_value),
...         )
...
...     @classmethod
...     def from_value( # This method is required by Applicative
...         cls,
...         inner_value: _NewNumberType,
...     ) -> 'Number[_NewNumberType]':
```

(continues on next page)

(continued from previous page)

```
...     return Number(inner_value)
```

This code gives us an opportunity to use `Number` with `map`, `apply`, and `bind` as we already did in the examples above.

Laws

To make sure other people will be able to use your implementation, it should respect three new laws.

1. *Left Identity*: If we bind a function to our bindable must have to be the same result as passing the value directly to the function.

```
>>> def can_be_bound(value: int) -> Number[int]:
...     return Number(value)
>>> assert Number.from_value(5).bind(can_be_bound) == can_be_bound(5)
```

2. *Right Identity*: If we pass the bindable constructor through `bind` must have to be the same result as instantiating the bindable on our own.

```
>>> number = Number(2)
>>> assert number.bind(Number) == Number(2)
```

3. *Associative Law*: Given two functions, `x` and `y`, calling the `bind` method with `x` function and after that calling with `y` function must have the same result if we bind with a function that passes the value to `x` and then bind the result with `y`.

```
>>> def minus_one(arg: int) -> Number[int]:
...     return Number(arg - 1)
>>> def half(arg: int) -> Number[int]:
...     return Number(arg // 2)
>>> number = Number(9)
>>> assert number.bind(minus_one).bind(half) == number.bind(
...     lambda value: minus_one(value).bind(half),
... )
```

Plus all laws from `MappableN` and `ApplicativeN` interfaces.

9.5.6 More!

We have way more interfaces with different features! We have covered all of them in the technical docs.

So, use them to enforce type-safety of your own containers.

Specific interfaces

We also have a whole package of different specific interfaces that will help you to create containers based on our internal types, like `Result`.

9.5.7 FAQ

Why do you have general and specific interfaces?

We have `.interfaces.*` types that can be applied to any possible type. There's nothing they know about other types or `returns` package.

We also have a special `.interfaces.specific` package where we have types that know about other types in `returns`. For example, `MappableN` from `.interfaces` only knows about `.map` method. It does not require anything else.

But, `ResultLikeN` from `.interfaces.specific.result` does require to have `.bind_result` method which relies on our `Result` type.

That's the only difference. Build your own types with any of those interfaces.

Why some interfaces do not have type alias for 1 or 2 type arguments?

Some types like `ResultLikeN` do not have type aliases for one type argument in a form of `ResultLike1`.

Why does `Mappable1` exists and `ResultLike1` does not?

Because `Mappable1` does make sense. But, `ResultLike1` requires at least two (value and error) types to exist. The same applies for `ReaderLike1` and `ReaderResultLike1` and `ReaderResultLike2`.

We don't support type aliases for types that won't make sense.

What's the difference between `MappableN` and `BindableN`?

While `MappableN` you have to pass a pure function, like:

```
>>> def can_be_mapped(string: str) -> str:
...   return string
```

with `Bindable` we have to pass a function that returns another container:

```
>>> from returns.maybe import Maybe
>>> def can_be_bound(string: str) -> Maybe[str]:
...   return Some(string + '!')
```

The main difference is the return type. The consequence of this is big! `BindableN` allows to change the container type. While `MappableN` cannot do that.

So, `Some.bind(function)` can be evaluated to both `Some` and `Nothing`. While `Some.map(function)` will always stay as `Some`.

What is the difference between ResultLikeN and ResultBasedN?

ResultLikeN is just an intention of having a result (e.g. `FutureResult`), it's not the result yet. While ResultBasedN is a concrete result (e.g. `IOResult`), it has the desired result value.

Because of this difference between them is why we can't unwrap a ResultLikeN container, it does not have the real result yet.

See the example below using `FutureResult` to get a `IOResult`:

```
>>> import anyio
>>> from returns.future import FutureResult
>>> from returns.interfaces.specific.future_result import FutureResultBasedN
>>> from returns.interfaces.specific.ioresult import (
...     IOResultBasedN,
...     IOResultLikeN,
... )
>>> from returns.interfaces.specific.result import ResultLikeN, ResultBasedN
>>> from returns.io import IOSuccess, IOResult
>>> from returns.result import Success, Result

>>> async def coro(arg: int) -> Result[int, str]:
...     return Success(arg + 1)

>>> # `result_like` does not have the result we want (Result[int, str])
>>> # it's just the intention of having one,
>>> # we have to await it to get the real result
>>> result_like: FutureResult[int, str] = FutureResult(coro(1))
>>> assert isinstance(result_like, FutureResultBasedN)
>>> assert isinstance(result_like, IOResultLikeN)
>>> assert isinstance(result_like, ResultLikeN)

>>> # `anyio.run(...)` will await our coroutine and give the real result to us
>>> result: IOResult[int, str] = anyio.run(result_like.awaitable)
>>> assert isinstance(result, IOResultBasedN)
>>> assert isinstance(result, ResultLikeN)

>>> # Compare it with the real result:
>>> assert isinstance(Success(1), ResultBasedN)
```

Note: The same difference applies to all `*ResultLikeN` vs `*ResultBasedN` (e.g. `IOResultLikeN` and `IOResultBasedN`)

9.5.8 API Reference

Overview

Here's a full overview of all our interfaces:

Let's review it one by one.

Equable

final class `_LawSpec`

Bases: `LawSpecDef`

Equality laws.

Description: <https://bit.ly/34D40iT>

static `reflexive_law`(*first*)

Value should be equal to itself.

Parameters

first (TypeVar(_EqualType, bound= Equable)) –

Return type

None

static `symmetry_law`(*first*, *second*)

If $A == B$ then $B == A$.

Parameters

- **first** (TypeVar(_EqualType, bound= Equable)) –
- **second** (TypeVar(_EqualType, bound= Equable)) –

Return type

None

static `transitivity_law`(*first*, *second*, *third*)

If $A == B$ and $B == C$ then $A == C$.

Parameters

- **first** (TypeVar(_EqualType, bound= Equable)) –
- **second** (TypeVar(_EqualType, bound= Equable)) –
- **third** (TypeVar(_EqualType, bound= Equable)) –

Return type

None

class `Equable`

Bases: `Lawful[Equable]`

Interface for types that can be compared with real values.

Not all types can, because some don't have the value at a time: - Future has to be awaited to get the value - Reader has to be called to get the value

```
_laws: ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law1 object>,
<returns.primitives.laws.Law2 object>, <returns.primitives.laws.Law3 object>)
```

Some classes and interfaces might have laws, some might not have any.

abstract equals(*other*)

Type-safe equality check for values of the same type.

Parameters

- **self** (TypeVar(_EqualType, bound= Equable)) –
- **other** (TypeVar(_EqualType, bound= Equable)) –

Return type

bool

Mappable

final class _LawSpec

Bases: *LawSpecDef*

Mappable or functor laws.

https://en.wikibooks.org/wiki/Haskell/The_Functor_class#The_functor_laws

static identity_law(*mappable*)

Mapping identity over a value must return the value unchanged.

Parameters

mappable (*MappableN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –

Return type

None

static associative_law(*mappable*, *first*, *second*)

Mapping twice or mapping a composition is the same thing.

Parameters

- **mappable** (*MappableN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **first** (Callable[[TypeVar(_FirstType)], TypeVar(_NewType1)]) –
- **second** (Callable[[TypeVar(_NewType1)], TypeVar(_NewType2)]) –

Return type

None

class MappableN

Bases: Generic[_FirstType, _SecondType, _ThirdType], *Lawful*[MappableN[_FirstType, _SecondType, _ThirdType]]

Allows to chain wrapped values in containers with regular functions.

Behaves like a functor.

See also:

- <https://en.wikipedia.org/wiki/Functor>

```
_laws: ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law1 object>,
<returns.primitives.laws.Law3 object>)
```

Some classes and interfaces might have laws, some might not have any.

abstract map(*function*)

Allows to run a pure function over a container.

Parameters

- **self** (TypeVar(_MappableType, bound= MappableN)) –
- **function** (Callable[[TypeVar(_FirstType)], TypeVar(_UpdatedType)]) –

Return type

KindN[TypeVar(_MappableType, bound= MappableN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]

Mappable1

Type alias for kinds with one type argument.

alias of *MappableN*[_FirstType, NoReturn, NoReturn]

Mappable2

Type alias for kinds with two type arguments.

alias of *MappableN*[_FirstType, _SecondType, NoReturn]

Mappable3

Type alias for kinds with three type arguments.

alias of *MappableN*[_FirstType, _SecondType, _ThirdType]

Bindable

class BindableN

Bases: Generic[_FirstType, _SecondType, _ThirdType]

Represents a “context” in which calculations can be executed.

Bindable allows you to bind together a series of calculations while maintaining the context of that specific container.

In contrast to *returns.interfaces.lashable.LashableN*, works with the first type argument.

abstract bind(*function*)

Applies ‘function’ to the result of a previous calculation.

And returns a new container.

Parameters

- **self** (TypeVar(_BindableType, bound= BindableN)) –
- **function** (Callable[[TypeVar(_FirstType)], *KindN*[TypeVar(_BindableType, bound= BindableN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]) –

Return type

KindN[TypeVar(_BindableType, bound= BindableN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]

Bindable1

Type alias for kinds with one type argument.

alias of `BindableN[_FirstType, NoReturn, NoReturn]`

Bindable2

Type alias for kinds with two type arguments.

alias of `BindableN[_FirstType, _SecondType, NoReturn]`

Bindable3

Type alias for kinds with three type arguments.

alias of `BindableN[_FirstType, _SecondType, _ThirdType]`

Applicative**final class _LawSpec**

Bases: `LawSpecDef`

Applicative mappable laws.

Definition: <https://bit.ly/3hC8F8E> Discussion: <https://bit.ly/3jffz3L>

static identity_law(container)

Identity law.

If we apply wrapped identity function to a container, nothing happens.

Parameters

container (`ApplicativeN`[`TypeVar`(`_FirstType`), `TypeVar`(`_SecondType`), `TypeVar`(`_ThirdType`)])

Return type

None

static interchange_law(raw_value, container, function)

Interchange law.

Basically we check that we can start our composition with both `raw_value` and `function`.

Great explanation: <https://stackoverflow.com/q/27285918/4842742>

Parameters

- **raw_value** (`TypeVar`(`_FirstType`))
- **container** (`ApplicativeN`[`TypeVar`(`_FirstType`), `TypeVar`(`_SecondType`), `TypeVar`(`_ThirdType`)])
- **function** (`Callable`[[`TypeVar`(`_FirstType`), `TypeVar`(`_NewType1`)])

Return type

None

static homomorphism_law(raw_value, container, function)

Homomorphism law.

The homomorphism law says that applying a wrapped function to a wrapped value is the same as applying the function to the value in the normal way and then using `.from_value` on the result.

Parameters

- **raw_value** (TypeVar(_FirstType)) –
- **container** (*ApplicativeN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **function** (Callable[[TypeVar(_FirstType)], TypeVar(_NewType1)]) –

Return type

None

static composition_law(*container*, *first*, *second*)

Composition law.

Applying two functions twice is the same as applying their composition once.

Parameters

- **container** (*ApplicativeN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **first** (Callable[[TypeVar(_FirstType)], TypeVar(_NewType1)]) –
- **second** (Callable[[TypeVar(_NewType1)], TypeVar(_NewType2)]) –

Return type

None

class ApplicativeNBases: *MappableN*[_FirstType, _SecondType, _ThirdType], *Lawful*[*ApplicativeN*[_FirstType, _SecondType, _ThirdType]]

Allows to create unit containers from raw values and to apply wrapped funcs.

See also:

- https://en.wikipedia.org/wiki/Applicative_functor
- <http://learnyouahaskell.com/functors-applicative-functors-and-monoids>

```
_laws: ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law1 object>,
<returns.primitives.laws.Law3 object>, <returns.primitives.laws.Law3 object>,
<returns.primitives.laws.Law3 object>)
```

Some classes and interfaces might have laws, some might not have any.

abstract apply(*container*)

Allows to apply a wrapped function over a container.

Parameters

- **self** (TypeVar(_ApplicativeType, bound= *ApplicativeN*)) –
- **container** (*KindN*[TypeVar(_ApplicativeType, bound= *ApplicativeN*), Callable[[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]]) –

Return type*KindN*[TypeVar(_ApplicativeType, bound= *ApplicativeN*), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]**abstract classmethod from_value**(*inner_value*)

Unit method to create new containers from any raw value.

Parameters**inner_value** (TypeVar(_UpdatedType)) –**Return type***KindN*[TypeVar(_ApplicativeType, bound= ApplicativeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]**Applicative1**

Type alias for kinds with one type argument.

alias of *ApplicativeN*[_FirstType, NoReturn, NoReturn]**Applicative2**

Type alias for kinds with two type arguments.

alias of *ApplicativeN*[_FirstType, _SecondType, NoReturn]**Applicative3**

Type alias for kinds with three type arguments.

alias of *ApplicativeN*[_FirstType, _SecondType, _ThirdType]**Altable****final class _LawSpec**Bases: *LawSpecDef*

Mappable or functor laws.

https://en.wikibooks.org/wiki/Haskell/The_Functor_class#The_functor_laws**static identity_law**(*altable*)

Mapping identity over a value must return the value unchanged.

Parameters**altable** (*AltableN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –**Return type**

None

static associative_law(*altable*, *first*, *second*)

Mapping twice or mapping a composition is the same thing.

Parameters

- **altable** (*AltableN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **first** (Callable[[TypeVar(_SecondType)], TypeVar(_NewType1)]) –
- **second** (Callable[[TypeVar(_NewType1)], TypeVar(_NewType2)]) –

Return type

None

class AltableNBases: Generic[_FirstType, _SecondType, _ThirdType], *Lawful*[*AltableN*[_FirstType, _SecondType, _ThirdType]]

Modifies the second type argument with a pure function.

```
_laws: ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law1 object>,
<returns.primitives.laws.Law3 object>)
```

Some classes and interfaces might have laws, some might not have any.

abstract alt(*function*)

Allows to run a pure function over a container.

Parameters

- **self** (TypeVar(_AltableType, bound= AltableN)) –
- **function** (Callable[[TypeVar(_SecondType), TypeVar(_UpdatedType)]) –

Return type

KindN[TypeVar(_AltableType, bound= AltableN), TypeVar(_FirstType), TypeVar(_UpdatedType), TypeVar(_ThirdType)]

Altable2

Type alias for kinds with two type arguments.

alias of *AltableN*[_FirstType, _SecondType, NoReturn]

Altable3

Type alias for kinds with three type arguments.

alias of *AltableN*[_FirstType, _SecondType, _ThirdType]

BiMappable

class BiMappableN

Bases: *MappableN*[_FirstType, _SecondType, _ThirdType], *AltableN*[_FirstType, _SecondType, _ThirdType]

Allows to change both types of a container at the same time.

Uses `.map` to change first type and `.alt` to change second type.

See also:

- <https://typelevel.org/cats/typeclasses/bifunctor.html>

BiMappable2

Type alias for kinds with two type arguments.

alias of *BiMappableN*[_FirstType, _SecondType, NoReturn]

BiMappable3

Type alias for kinds with three type arguments.

alias of *BiMappableN*[_FirstType, _SecondType, _ThirdType]

Swappable

final class _LawSpec

Bases: *LawSpecDef*

Laws for *SwappableN* type.

static double_swap_law(container)

Swapping container twice.

It ensure that we get the initial value back. In other words, swapping twice does nothing.

Parameters

container (*SwappableN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –

Return type

None

class SwappableN

Bases: *BiMappableN*[_FirstType, _SecondType, _ThirdType], *Lawful*[*SwappableN*[_FirstType, _SecondType, _ThirdType]]

Interface that allows swapping first and second type values.

_laws: ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law1 object>,)

Some classes and interfaces might have laws, some might not have any.

abstract swap()

Swaps first and second types in *SwappableN*.

Parameters

self (TypeVar(_SwappableType, bound= *SwappableN*)) –

Return type

KindN[TypeVar(_SwappableType, bound= *SwappableN*), TypeVar(_SecondType), TypeVar(_FirstType), TypeVar(_ThirdType)]

Swappable2

Type alias for kinds with two type arguments.

alias of *SwappableN*[_FirstType, _SecondType, NoReturn]

Swappable3

Type alias for kinds with three type arguments.

alias of *SwappableN*[_FirstType, _SecondType, _ThirdType]

Lashable

class LashableN

Bases: Generic[_FirstType, _SecondType, _ThirdType]

Represents a “context” in which calculations can be executed.

Rescueable allows you to bind together a series of calculations while maintaining the context of that specific container.

In contrast to `returns.interfaces.bindable.BindableN`, works with the second type value.

abstract lash(*function*)

Applies ‘function’ to the result of a previous calculation.

And returns a new container.

Parameters

- **self** (TypeVar(_LashableType, bound= LashableN)) –
- **function** (Callable[[TypeVar(_SecondType)], KindN[TypeVar(_LashableType, bound= LashableN), TypeVar(_FirstType), TypeVar(_UpdatedType), TypeVar(_ThirdType)]])) –

Return type

KindN[TypeVar(_LashableType, bound= LashableN), TypeVar(_FirstType), TypeVar(_UpdatedType), TypeVar(_ThirdType)]

Lashable2

Type alias for kinds with two type arguments.

alias of *LashableN*[_FirstType, _SecondType, NoReturn]

Lashable3

Type alias for kinds with three type arguments.

alias of *LashableN*[_FirstType, _SecondType, _ThirdType]

Unwrappable**class Unwrappable**

Bases: Generic[_FirstType, _SecondType]

Represents containers that can unwrap and return its wrapped value.

There are no aliases or UnwrappableN for Unwrappable interface. Because it always uses two and just two types.

Not all types can be Unwrappable because we do require to raise UnwrapFailedError if unwrap is not possible.

abstract unwrap()

Custom magic method to unwrap inner value from container.

Should be redefined for ones that actually have values. And for ones that raise an exception for no values.

Note: As a part of the contract, failed unwrap calls must raise *returns.primitives.exceptions.UnwrapFailedError* exception.

This method is the opposite of `failure()`.

Parameters

self (TypeVar(_UnwrappableType, bound= Unwrappable)) –

Return type

TypeVar(_FirstType)

abstract failure()

Custom magic method to unwrap inner value from the failed container.

Note: As a part of the contract, failed `failure` calls must raise `returns.primitives.exceptions.UnwrapFailedError` exception.

This method is the opposite of `unwrap()`.

Parameters

self (TypeVar(_UnwrappableType, bound= Unwrappable)) –

Return type

TypeVar(_SecondType)

Container**final class _LawSpec**

Bases: `LawSpecDef`

Container laws.

Definition: https://wiki.haskell.org/Monad_laws Good explanation: <https://bit.ly/2Qsi5re>

static left_identity_law(raw_value, container, function)

Left identity.

The first law states that if we take a value, put it in a default context with `return` and then feed it to a function by using `bind`, it's the same as just taking the value and applying the function to it.

Parameters

- **raw_value** (TypeVar(_FirstType)) –
- **container** (`ContainerN`[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **function** (Callable[[TypeVar(_FirstType)], `KindN`[`ContainerN`, TypeVar(_NewType1), TypeVar(_SecondType), TypeVar(_ThirdType)]]]) –

Return type

None

static right_identity_law(container)

Right identity.

The second law states that if we have a container value and we use `bind` to feed it to `.from_value`, the result is our original container value.

Parameters

container (`ContainerN`[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –

Return type

None

static associative_law(container, first, second)

Associativity law.

The final monad law says that when we have a chain of container functions applications with `bind`, it shouldn't matter how they're nested.

Parameters

- **container** (*ContainerN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **first** (Callable[[TypeVar(_FirstType), *KindN*[*ContainerN*, TypeVar(_NewType1), TypeVar(_SecondType), TypeVar(_ThirdType)]]]) –
- **second** (Callable[[TypeVar(_NewType1), *KindN*[*ContainerN*, TypeVar(_NewType2), TypeVar(_SecondType), TypeVar(_ThirdType)]]]) –

Return type

None

class ContainerN

Bases: *ApplicativeN*[_FirstType, _SecondType, _ThirdType], *BindableN*[_FirstType, _SecondType, _ThirdType], *Lawful*[*ContainerN*[_FirstType, _SecondType, _ThirdType]]

Handy alias for types with `.bind`, `.map`, and `.apply` methods.

Should be a base class for almost any containers you write.

See also:

- <https://bit.ly/2CTEVov>

```
_laws: ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law3 object>, <returns.primitives.laws.Law1 object>, <returns.primitives.laws.Law3 object>)
```

Some classes and interfaces might have laws, some might not have any.

Container1

Type alias for kinds with one type argument.

alias of *ContainerN*[_FirstType, NoReturn, NoReturn]

Container2

Type alias for kinds with two type arguments.

alias of *ContainerN*[_FirstType, _SecondType, NoReturn]

Container3

Type alias for kinds with three type arguments.

alias of *ContainerN*[_FirstType, _SecondType, _ThirdType]

Failable**final class _FailableLawSpec**

Bases: *LawSpecDef*

Failable laws.

We need to be sure that `.lash` won't lash success types.

```
static lash_short_circuit_law(raw_value, container, function)
```

Ensures that you cannot lash a success.

Parameters

- **raw_value** (TypeVar(_FirstType)) –

- **container** (*FailableN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **function** (Callable[[TypeVar(_SecondType)], *KindN*[*FailableN*, TypeVar(_FirstType), TypeVar(_NewFirstType), TypeVar(_ThirdType)]] –

Return type

None

class FailableN

Bases: *ContainerN*[_FirstType, _SecondType, _ThirdType], *LashableN*[_FirstType, _SecondType, _ThirdType], *Lawful*[*FailableN*[_FirstType, _SecondType, _ThirdType]]

Base type for types that can fail.

It is a raw type and should not be used directly. Use *SingleFailableN* and *DiverseFailableN* instead.

_laws: *ClassVar*[*Sequence*[*Law*]] = (<returns.primitives.laws.Law3 object>,)

Some classes and interfaces might have laws, some might not have any.

Failable2

Type alias for kinds with two type arguments.

alias of *FailableN*[_FirstType, _SecondType, NoReturn]

Failable3

Type alias for kinds with three type arguments.

alias of *FailableN*[_FirstType, _SecondType, _ThirdType]

final class _SingleFailableLawSpec

Bases: *LawSpecDef*

Single Failable laws.

We need to be sure that `.map` and `.bind` works correctly for `empty` property.

static map_short_circuit_law(*container*, *function*)

Ensures that you cannot map from the *empty* property.

Parameters

- **container** (*SingleFailableN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **function** (Callable[[TypeVar(_FirstType)], TypeVar(_NewFirstType)]) –

Return type

None

static bind_short_circuit_law(*container*, *function*)

Ensures that you cannot bind from the *empty* property.

Parameters

- **container** (*SingleFailableN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **function** (Callable[[TypeVar(_FirstType)], *KindN*[*SingleFailableN*, TypeVar(_NewFirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]] –

Return type

None

static apply_short_circuit_law(*container*, *function*)

Ensures that you cannot apply from the *empty* property.

Parameters

- **container** (*SingleFailableN*[*TypeVar*(_FirstType), *TypeVar*(_SecondType), *TypeVar*(_ThirdType)]) –
- **function** (*Callable*[[*TypeVar*(_FirstType)], *TypeVar*(_NewFirstType)]) –

Return type

None

class SingleFailableN

Bases: *FailableN*[_FirstType, _SecondType, _ThirdType]

Base type for types that have just only one failed value.

Like Maybe types where the only failed value is *Nothing*.

_laws: *ClassVar*[*Sequence*[*Law*]] = (<*returns.primitives.laws.Law2* object>, <*returns.primitives.laws.Law2* object>, <*returns.primitives.laws.Law2* object>)

Some classes and interfaces might have laws, some might not have any.

abstract property empty: *SingleFailableN*[_FirstType, _SecondType, _ThirdType]

This property represents the failed value. :param self: :type self: *TypeVar*(_SingleFailableType, bound= *SingleFailableN*)

SingleFailable2

Type alias for kinds with two types arguments.

alias of *SingleFailableN*[_FirstType, _SecondType, *NoReturn*]

SingleFailable3

Type alias for kinds with three type arguments.

alias of *SingleFailableN*[_FirstType, _SecondType, _ThirdType]

final class _DiverseFailableLawSpec

Bases: *LawSpecDef*

Diverse Failable laws.

We need to be sure that *.map*, *.bind*, *.apply* and *.alt* works correctly for both success and failure types.

static map_short_circuit_law(*raw_value*, *container*, *function*)

Ensures that you cannot map a failure.

Parameters

- **raw_value** (*TypeVar*(_SecondType)) –
- **container** (*DiverseFailableN*[*TypeVar*(_FirstType), *TypeVar*(_SecondType), *TypeVar*(_ThirdType)]) –
- **function** (*Callable*[[*TypeVar*(_FirstType)], *TypeVar*(_NewFirstType)]) –

Return type

None

static bind_short_circuit_law(*raw_value*, *container*, *function*)

Ensures that you cannot bind a failure.

See: <https://wiki.haskell.org/Typeclassopedia#MonadFail>

Parameters

- **raw_value** (TypeVar(_SecondType)) –
- **container** (*DiverseFailableN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **function** (Callable[[TypeVar(_FirstType)], *KindN*[*DiverseFailableN*, TypeVar(_NewFirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]]) –

Return type

None

static apply_short_circuit_law(raw_value, container, function)

Ensures that you cannot apply a failure.

Parameters

- **raw_value** (TypeVar(_SecondType)) –
- **container** (*DiverseFailableN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **function** (Callable[[TypeVar(_FirstType)], TypeVar(_NewFirstType)]) –

Return type

None

static alt_short_circuit_law(raw_value, container, function)

Ensures that you cannot alt a success.

Parameters

- **raw_value** (TypeVar(_SecondType)) –
- **container** (*DiverseFailableN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **function** (Callable[[TypeVar(_SecondType)], TypeVar(_NewFirstType)]) –

Return type

None

class DiverseFailableN

Bases: *FailableN*[_FirstType, _SecondType, _ThirdType], *SwappableN*[_FirstType, _SecondType, _ThirdType], *Lawful*[*DiverseFailableN*[_FirstType, _SecondType, _ThirdType]]

Base type for types that have any failed value.

Like Result types.

```
_laws: ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law3 object>,
<returns.primitives.laws.Law3 object>, <returns.primitives.laws.Law3 object>,
<returns.primitives.laws.Law3 object>)
```

Some classes and interfaces might have laws, some might not have any.

abstract classmethod from_failure(inner_value)

Unit method to create new containers from any raw value.

Parameters

- **inner_value** (TypeVar(_UpdatedType)) –

Return type

`KindN`[`TypeVar`(`_DiverseFailableType`, `bound=` `DiverseFailableN`),
`TypeVar`(`_FirstType`), `TypeVar`(`_UpdatedType`), `TypeVar`(`_ThirdType`)]

DiverseFailable2

Type alias for kinds with two type arguments.

alias of `DiverseFailableN`[`_FirstType`, `_SecondType`, `NoReturn`]

DiverseFailable3

Type alias for kinds with three type arguments.

alias of `DiverseFailableN`[`_FirstType`, `_SecondType`, `_ThirdType`]

Maybe specific**final class** `_LawSpec`

Bases: `LawSpecDef`

Maybe laws.

We need to be sure that `.map`, `.bind`, `.bind_optional`, and `.lash` works correctly for both successful and failed types.

static `map_short_circuit_law`(*container*, *function*)

Ensures that you cannot map from failures.

Parameters

- **container** (`MaybeLikeN`[`TypeVar`(`_FirstType`), `TypeVar`(`_SecondType`), `TypeVar`(`_ThirdType`)]) –
- **function** (`Callable`[[`TypeVar`(`_FirstType`)], `TypeVar`(`_NewType1`)]) –

Return type

`None`

static `bind_short_circuit_law`(*container*, *function*)

Ensures that you cannot bind from failures.

Parameters

- **container** (`MaybeLikeN`[`TypeVar`(`_FirstType`), `TypeVar`(`_SecondType`), `TypeVar`(`_ThirdType`)]) –
- **function** (`Callable`[[`TypeVar`(`_FirstType`)], `KindN`[`MaybeLikeN`, `TypeVar`(`_NewType1`), `TypeVar`(`_SecondType`), `TypeVar`(`_ThirdType`)]]) –

Return type

`None`

static `bind_optional_short_circuit_law`(*container*, *function*)

Ensures that you cannot bind from failures.

Parameters

- **container** (`MaybeLikeN`[`TypeVar`(`_FirstType`), `TypeVar`(`_SecondType`), `TypeVar`(`_ThirdType`)]) –
- **function** (`Callable`[[`TypeVar`(`_FirstType`)], `Optional`[`TypeVar`(`_NewType1`)]]) –

Return type

None

static `lash_short_circuit_law(raw_value, container, function)`

Ensures that you cannot lash a success.

Parameters

- **raw_value** (TypeVar(_FirstType)) –
- **container** (*MaybeLikeN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **function** (Callable[[TypeVar(_SecondType), *KindN*[*MaybeLikeN*, TypeVar(_FirstType), TypeVar(_NewType1), TypeVar(_ThirdType)]]]) –

Return type

None

static `unit_structure_law(container, function)`

Ensures None is treated specially.

Parameters

- **container** (*MaybeLikeN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **function** (Callable[[TypeVar(_FirstType), None]) –

Return type

None

class `MaybeLikeN`Bases: *SingleFailableN*[_FirstType, _SecondType, _ThirdType], *Lawful*[*MaybeLikeN*[_FirstType, _SecondType, _ThirdType]]

Type for values that do look like a Maybe.

For example, `RequiresContextMaybe` should be created from this interface. Cannot be unwrapped or compared.

```
_laws: ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law2 object>,
<returns.primitives.laws.Law2 object>, <returns.primitives.laws.Law2 object>,
<returns.primitives.laws.Law3 object>, <returns.primitives.laws.Law2 object>)
```

Some classes and interfaces might have laws, some might not have any.

abstract `bind_optional(function)`Binds a function that returns `Optional` values.**Parameters**

- **self** (TypeVar(_MaybeLikeType, bound= *MaybeLikeN*)) –
- **function** (Callable[[TypeVar(_FirstType), `Optional`[TypeVar(_UpdatedType)]]]) –

Return type*KindN*[TypeVar(_MaybeLikeType, bound= *MaybeLikeN*), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]**abstract classmethod** `from_optional(inner_value)`Unit method to create containers from `Optional` value.

Parameters**inner_value** (Optional[TypeVar(_ValueType)]) –**Return type***KindN*[TypeVar(_MaybeLikeType, bound= MaybeLikeN), TypeVar(_ValueType), TypeVar(_SecondType), TypeVar(_ThirdType)]**MaybeLike2**

Type alias for kinds with two type arguments.

alias of *MaybeLikeN*[_FirstType, _SecondType, NoReturn]**MaybeLike3**

Type alias for kinds with three type arguments.

alias of *MaybeLikeN*[_FirstType, _SecondType, _ThirdType]**class MaybeBasedN**Bases: *MaybeLikeN*[_FirstType, _SecondType, _ThirdType], *Unwrappable*[_FirstType, None], *Equable*

Concrete interface for Maybe type.

Can be unwrapped and compared.

abstract or_else_call(*function*)

Calls a function in case there nothing to unwrap.

Parameters**function** (Callable[[], TypeVar(_ValueType)]) –**Return type**

Union[TypeVar(_FirstType), TypeVar(_ValueType)]

MaybeBased2

Type alias for kinds with two type arguments.

alias of *MaybeBasedN*[_FirstType, _SecondType, NoReturn]**MaybeBased3**

Type alias for kinds with three type arguments.

alias of *MaybeBasedN*[_FirstType, _SecondType, _ThirdType]**Result specific**

An interface that represents a pure computation result.

For impure result see *returns.interfaces.specific.ioresult.IOResultLikeN* type.**class ResultLikeN**Bases: *DiverseFailableN*[_FirstType, _SecondType, _ThirdType]

Base types for types that looks like Result but cannot be unwrapped.

Like RequiresContextResult or FutureResult.

abstract bind_result(*function*)

Runs Result returning function over a container.

Parameters

- **self** (TypeVar(_ResultLikeType, bound= ResultLikeN)) –
- **function** (Callable[[TypeVar(_FirstType)], *Result*[TypeVar(_UpdatedType), TypeVar(_SecondType)]] –

Return type

KindN[TypeVar(_ResultLikeType, bound= ResultLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]

abstract classmethod from_result(*inner_value*)

Unit method to create new containers from any raw value.

Parameters

inner_value (*Result*[TypeVar(_ValueType), TypeVar(_ErrorType)]) –

Return type

KindN[TypeVar(_ResultLikeType, bound= ResultLikeN), TypeVar(_ValueType), TypeVar(_ErrorType), TypeVar(_ThirdType)]

ResultLike2

Type alias for kinds with two type arguments.

alias of *ResultLikeN*[_FirstType, _SecondType, NoReturn]

ResultLike3

Type alias for kinds with three type arguments.

alias of *ResultLikeN*[_FirstType, _SecondType, _ThirdType]

class UnwrappableResult

Bases: *ResultLikeN*[_FirstType, _SecondType, _ThirdType], *Unwrappable*[_FirstUnwrappableType, _SecondUnwrappableType], *Equable*

Intermediate type with 5 type arguments that represents unwrappable result.

It is a raw type and should not be used directly. Use *ResultBasedN* and *IOResultBasedN* instead.

class ResultBasedN

Bases: *UnwrappableResult*[_FirstType, _SecondType, _ThirdType, _FirstType, _SecondType]

Base type for real *Result* types.

Can be unwrapped.

ResultBased2

Type alias for kinds with two type arguments.

alias of *ResultBasedN*[_FirstType, _SecondType, NoReturn]

ResultBased3

Type alias for kinds with three type arguments.

alias of *ResultBasedN*[_FirstType, _SecondType, _ThirdType]

IO specific

class `IOLikeN`

Bases: `ContainerN[_FirstType, _SecondType, _ThirdType]`

Represents interface for types that looks like fearless IO.

This type means that IO cannot fail. Like random numbers, date, etc. Don't use this type for IO that can. Instead, use `returns.interfaces.specific.ioresult.IOResultBasedN` type.

abstract `bind_io`(*function*)

Allows to apply a wrapped function over a container.

Parameters

- **self** (`TypeVar(_IOLikeType, bound= IOLikeN)`) –
- **function** (`Callable[[TypeVar(_FirstType)], IO[TypeVar(_UpdatedType)]]`) –

Return type

`KindN[TypeVar(_IOLikeType, bound= IOLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]`

abstract classmethod `from_io`(*inner_value*)

Unit method to create new containers from successful IO.

Parameters

inner_value (`IO[TypeVar(_UpdatedType)]`) –

Return type

`KindN[TypeVar(_IOLikeType, bound= IOLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]`

`IOLike1`

Type alias for kinds with one type argument.

alias of `IOLikeN[_FirstType, NoReturn, NoReturn]`

`IOLike2`

Type alias for kinds with two type arguments.

alias of `IOLikeN[_FirstType, _SecondType, NoReturn]`

`IOLike3`

Type alias for kinds with three type arguments.

alias of `IOLikeN[_FirstType, _SecondType, _ThirdType]`

class `IOBasedN`

Bases: `IOLikeN[_FirstType, _SecondType, _ThirdType]`, `Equable`

Represents the base interface for types that do fearless IO.

This type means that IO cannot fail. Like random numbers, date, etc. Don't use this type for IO that can. Instead, use `returns.interfaces.specific.ioresult.IOResultBasedN` type.

This interface also supports direct comparison of two values. While `IOLikeN` is different. It can be lazy and cannot be compared.

`IOBased1`

Type alias for kinds with one type argument.

alias of `IOBasedN[_FirstType, NoReturn, NoReturn]`

IOBased2

Type alias for kinds with two type arguments.

alias of `IOBasedN[_FirstType, _SecondType, NoReturn]`

IOBased3

Type alias for kinds with three type arguments.

alias of `IOBasedN[_FirstType, _SecondType, _ThirdType]`

IOResult specific

An interface for types that do IO and can fail.

It is a base interface for both sync and async IO stacks.

class IOResultLikeN

Bases: `IOLikeN[_FirstType, _SecondType, _ThirdType]`, `ResultLikeN[_FirstType, _SecondType, _ThirdType]`

Base type for types that look like `IOResult` but cannot be unwrapped.

Like `FutureResult` or `RequiresContextIOResult`.

abstract bind_ioresult(*function*)

Runs `IOResult` returning function over a container.

Parameters

- **self** (`TypeVar(_IOResultLikeType, bound= IOResultLikeN)`) –
- **function** (`Callable[[TypeVar(_FirstType)], IOResult[TypeVar(_UpdatedType), TypeVar(_SecondType)]]`) –

Return type

`KindN[TypeVar(_IOResultLikeType, bound= IOResultLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]`

abstract compose_result(*function*)

Allows to compose the underlying `Result` with a function.

Parameters

- **self** (`TypeVar(_IOResultLikeType, bound= IOResultLikeN)`) –
- **function** (`Callable[[Result[TypeVar(_FirstType), TypeVar(_SecondType)]], KindN[TypeVar(_IOResultLikeType, bound= IOResultLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]`) –

Return type

`KindN[TypeVar(_IOResultLikeType, bound= IOResultLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]`

abstract classmethod from_ioresult(*inner_value*)

Unit method to create new containers from `IOResult` type.

Parameters

inner_value (`IOResult[TypeVar(_ValueType), TypeVar(_ErrorType)]`) –

Return type

`KindN[TypeVar(_IOResultLikeType, bound= IOResultLikeN), TypeVar(_ValueType), TypeVar(_ErrorType), TypeVar(_ThirdType)]`

abstract classmethod `from_failed_io(inner_value)`

Unit method to create new containers from failed IO.

Parameters

inner_value (*IO*[TypeVar(_ErrorType)]) –

Return type

KindN[TypeVar(_IOResultLikeType, bound= IOResultLikeN), TypeVar(_FirstType), TypeVar(_ErrorType), TypeVar(_ThirdType)]

IOResultLike2

Type alias for kinds with two type arguments.

alias of *IOResultLikeN*[_FirstType, _SecondType, NoReturn]

IOResultLike3

Type alias for kinds with three type arguments.

alias of *IOResultLikeN*[_FirstType, _SecondType, _ThirdType]

class IOResultBasedN

Bases: *IOResultLikeN*[_FirstType, _SecondType, _ThirdType], *IOBasedN*[_FirstType, _SecondType, _ThirdType], *UnwrappableResult*[_FirstType, _SecondType, _ThirdType, IO[_FirstType], IO[_SecondType]]

Base type for real IOResult types.

Can be unwrapped.

IOResultBased2

Type alias for kinds with two type arguments.

alias of *IOResultBasedN*[_FirstType, _SecondType, NoReturn]

IOResultBased3

Type alias for kinds with three type arguments.

alias of *IOResultBasedN*[_FirstType, _SecondType, _ThirdType]

Future specific

Represents the base interfaces for types that do fearless async operations.

This type means that `Future` cannot fail. Don't use this type for async that can. Instead, use `returns.interfaces.specific.future_result.FutureResultBasedN` type.

class FutureLikeN

Bases: *IOLikeN*[_FirstType, _SecondType, _ThirdType]

Base type for ones that does look like `Future`.

But at the time this is not a real `Future` and cannot be awaited.

abstract `bind_future(function)`

Allows to bind `Future` returning function over a container.

Parameters

- **self** (TypeVar(_FutureLikeType, bound= FutureLikeN)) –
- **function** (Callable[[TypeVar(_FirstType)], *Future*[TypeVar(_UpdatedType)]]) –

Return type

KindN[TypeVar(_FutureLikeType, bound= FutureLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]

abstract bind_async_future(*function*)

Allows to bind async Future returning function over container.

Parameters

- **self** (TypeVar(_FutureLikeType, bound= FutureLikeN)) –
- **function** (Callable[[TypeVar(_FirstType)], Awaitable[*Future*[TypeVar(_UpdatedType)]]]) –

Return type

KindN[TypeVar(_FutureLikeType, bound= FutureLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]

abstract bind_async(*function*)

Binds async function returning the same type of container.

Parameters

- **self** (TypeVar(_FutureLikeType, bound= FutureLikeN)) –
- **function** (Callable[[TypeVar(_FirstType)], Awaitable[*KindN*[TypeVar(_FutureLikeType, bound= FutureLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]) –

Return type

KindN[TypeVar(_FutureLikeType, bound= FutureLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]

abstract bind_awaitable(*function*)

Allows to bind async function over container.

Parameters

- **self** (TypeVar(_FutureLikeType, bound= FutureLikeN)) –
- **function** (Callable[[TypeVar(_FirstType)], Awaitable[TypeVar(_UpdatedType)]]]) –

Return type

KindN[TypeVar(_FutureLikeType, bound= FutureLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]

abstract classmethod from_future(*inner_value*)

Unit method to create new containers from successful Future.

Parameters

inner_value (*Future*[TypeVar(_UpdatedType)]) –

Return type

KindN[TypeVar(_FutureLikeType, bound= FutureLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]

FutureLike1

Type alias for kinds with one type argument.

alias of *FutureLikeN*[_FirstType, NoReturn, NoReturn]

FutureLike2

Type alias for kinds with two type arguments.

alias of *FutureLikeN*[_FirstType, _SecondType, NoReturn]

FutureLike3

Type alias for kinds with three type arguments.

alias of *FutureLikeN*[_FirstType, _SecondType, _ThirdType]

class AwaitableFutureN

Bases: *Generic*[_FirstType, _SecondType, _ThirdType]

Type that provides the required API for Future to be async.

Should not be used directly. Use FutureBasedN instead.

abstract async awaitable()

Underling logic under await expression.

Parameters

self (TypeVar[_AsyncFutureType, bound= AwaitableFutureN]) –

Return type

IOLikeN[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]

AsyncFuture1

Type alias for kinds with one type argument.

alias of *AwaitableFutureN*[_FirstType, NoReturn, NoReturn]

AsyncFuture2

Type alias for kinds with two type arguments.

alias of *AwaitableFutureN*[_FirstType, _SecondType, NoReturn]

AsyncFuture3

Type alias for kinds with three type arguments.

alias of *AwaitableFutureN*[_FirstType, _SecondType, _ThirdType]

class FutureBasedN

Bases: *FutureLikeN*[_FirstType, _SecondType, _ThirdType], *AwaitableFutureN*[_FirstType, _SecondType, _ThirdType]

Base type for real Future objects.

They can be awaited.

FutureBased1

Type alias for kinds with one type argument.

alias of *FutureBasedN*[_FirstType, NoReturn, NoReturn]

FutureBased2

Type alias for kinds with two type arguments.

alias of *FutureBasedN*[_FirstType, _SecondType, NoReturn]

FutureBased3

Type alias for kinds with three type arguments.

alias of *FutureBasedN*[_FirstType, _SecondType, _ThirdType]

FutureResult specific

Represents the base interfaces for types that do fear-some async operations.

This type means that FutureResult can (and will!) fail with exceptions.

Use this type to mark that this specific async opetaion can fail.

class FutureResultLikeN

Bases: `FutureLikeN[_FirstType, _SecondType, _ThirdType]`, `IOResultLikeN[_FirstType, _SecondType, _ThirdType]`

Base type for ones that does look like FutureResult.

But at the time this is not a real Future and cannot be awaited. It is also cannot be unwrapped, because it is not a real IOResult.

abstract bind_future_result(function)

Allows to bind FutureResult functions over a container.

Parameters

- **self** (TypeVar(_FutureResultLikeType, bound= FutureResultLikeN)) –
- **function** (Callable[[TypeVar(_FirstType)], FutureResult[TypeVar(_UpdatedType), TypeVar(_SecondType)]] –

Return type

`KindN[TypeVar(_FutureResultLikeType, bound= FutureResultLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]`

abstract bind_async_future_result(function)

Allows to bind async FutureResult functions over container.

Parameters

- **self** (TypeVar(_FutureResultLikeType, bound= FutureResultLikeN)) –
- **function** (Callable[[TypeVar(_FirstType)], Awaitable[FutureResult[TypeVar(_UpdatedType), TypeVar(_SecondType)]]]) –

Return type

`KindN[TypeVar(_FutureResultLikeType, bound= FutureResultLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]`

abstract classmethod from_failed_future(inner_value)

Creates new container from a failed Future.

Parameters

inner_value (Future[TypeVar(_ErrorType)]) –

Return type

`KindN[TypeVar(_FutureResultLikeType, bound= FutureResultLikeN), TypeVar(_FirstType), TypeVar(_ErrorType), TypeVar(_ThirdType)]`

classmethod from_future_result(inner_value)

Creates container from FutureResult instance.

Parameters

inner_value (FutureResult[TypeVar(_ValueType), TypeVar(_ErrorType)]) –

Return type

`KindN`[`TypeVar`(`_FutureResultLikeType`, `bound=FutureResultLikeN`),
`TypeVar`(`_ValueType`), `TypeVar`(`_ErrorType`), `TypeVar`(`_ThirdType`)]

FutureResultLike2

Type alias for kinds with two type arguments.

alias of `FutureResultLikeN`[`_FirstType`, `_SecondType`, `NoReturn`]

FutureResultLike3

Type alias for kinds with three type arguments.

alias of `FutureResultLikeN`[`_FirstType`, `_SecondType`, `_ThirdType`]

class FutureResultBasedN

Bases: `FutureBasedN`[`_FirstType`, `_SecondType`, `_ThirdType`], `FutureResultLikeN`[`_FirstType`,
`_SecondType`, `_ThirdType`]

Base type for real `FutureResult` objects.

They can be awaited. Still cannot be unwrapped.

FutureResultBased2

Type alias for kinds with two type arguments.

alias of `FutureResultBasedN`[`_FirstType`, `_SecondType`, `NoReturn`]

FutureResultBased3

Type alias for kinds with three type arguments.

alias of `FutureResultBasedN`[`_FirstType`, `_SecondType`, `_ThirdType`]

Reader specific

This module is special.

`Reader` does not produce `ReaderLikeN` interface as other containers.

Because `Reader` can be used with two or three type arguments: - `RequiresContext`[`value`, `env`] -
`RequiresContextResult`[`value`, `error`, `env`]

Because the second type argument changes its meaning based on the used `KindN` instance, we need to have two separate interfaces for two separate use-cases: - `ReaderLike2` is used for types where the second type argument is `env` -
`ReaderLike3` is used for types where the third type argument is `env`

We also have two methods and two poinfree helpers for `bind_context` composition: one for each interface.

Furthermore, `Reader` cannot have `ReaderLike1` type, because we need both `value` and `env` types at all cases.

See also:

- <https://github.com/dry-python/returns/issues/485>

class Contextable

Bases: `Generic`[`_ValueType`, `_EnvType`]

Special type we use as a base one for all callable `Reader` instances.

It only has a single method. And is a base type for every single one of them.

But, each `Reader` defines the return type differently. For example:

- `Reader` has just `_ReturnType`

- ReaderResult has Result[_FirstType, _SecondType]
- ReaderIOResult has IOResult[_FirstType, _SecondType]

And so on.

class ReaderLike2

Bases: *ContainerN*[_FirstType, _SecondType, NoReturn]

Reader interface for Kind2 based types.

It has two type arguments and treats the second type argument as env type.

abstract property no_args: NoDeps

Is required to call Reader with no explicit arguments. :param self: :type self:
TypeVar(_ReaderLike2Type, bound= ReaderLike2)

abstract bind_context(function)

Allows to apply a wrapped function over a Reader container.

Parameters

- **self** (TypeVar(_ReaderLike2Type, bound= ReaderLike2)) –
- **function** (Callable[[TypeVar(_FirstType)], *RequiresContext*[TypeVar(_UpdatedType), TypeVar(_SecondType)]]) –

Return type

KindN[TypeVar(_ReaderLike2Type, bound= ReaderLike2), TypeVar(_UpdatedType), TypeVar(_SecondType), Any]

abstract modify_env(function)

Transforms the environment before calling the container.

Parameters

- **self** (TypeVar(_ReaderLike2Type, bound= ReaderLike2)) –
- **function** (Callable[[TypeVar(_UpdatedType)], TypeVar(_SecondType)]) –

Return type

KindN[TypeVar(_ReaderLike2Type, bound= ReaderLike2), TypeVar(_FirstType), TypeVar(_UpdatedType), Any]

abstract classmethod ask()

Returns the dependencies inside the container.

Return type

KindN[TypeVar(_ReaderLike2Type, bound= ReaderLike2), TypeVar(_SecondType), TypeVar(_SecondType), Any]

abstract classmethod from_context(inner_value)

Unit method to create new containers from successful Reader.

Parameters

inner_value (*RequiresContext*[TypeVar(_ValueType), TypeVar(_EnvType)]) –

Return type

KindN[TypeVar(_ReaderLike2Type, bound= ReaderLike2), TypeVar(_ValueType), TypeVar(_EnvType), Any]

class CallableReader2

Bases: *ReaderLike2*[_FirstType, _SecondType], *Contextable*[_ValueType, _EnvType]

Intermediate interface for ReaderLike2 + `__call__` method.

Has 4 type variables to type Reader and `__call__` independently. Since, we don't have any other fancy ways of doing it.

Should not be used directly other than defining your own Reader interfaces.

class ReaderLike3

Bases: *ContainerN*[_FirstType, _SecondType, _ThirdType]

Reader interface for Kind3 based types.

It has three type arguments and treats the third type argument as env type. The second type argument is not used here.

abstract property no_args: Any

Is required to call Reader with no explicit arguments. `:param self: :type self: TypeVar(_ReaderLike3Type, bound= ReaderLike3)`

abstract bind_context(function)

Allows to apply a wrapped function over a Reader container.

Parameters

- **self** (TypeVar(_ReaderLike3Type, bound= ReaderLike3)) –
- **function** (Callable[[TypeVar(_FirstType)], *RequiresContext*[TypeVar(_UpdatedType), TypeVar(_ThirdType)]]) –

Return type

KindN[TypeVar(_ReaderLike3Type, bound= ReaderLike3), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]

abstract modify_env(function)

Transforms the environment before calling the container.

Parameters

- **self** (TypeVar(_ReaderLike3Type, bound= ReaderLike3)) –
- **function** (Callable[[TypeVar(_UpdatedType)], TypeVar(_ThirdType)]) –

Return type

KindN[TypeVar(_ReaderLike3Type, bound= ReaderLike3), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_UpdatedType)]

abstract classmethod ask()

Returns the dependencies inside the container.

Return type

KindN[TypeVar(_ReaderLike3Type, bound= ReaderLike3), TypeVar(_ThirdType), TypeVar(_SecondType), TypeVar(_ThirdType)]

abstract classmethod from_context(inner_value)

Unit method to create new containers from successful Reader.

Parameters

inner_value (*RequiresContext*[TypeVar(_ValueType), TypeVar(_EnvType)]) –

Return type

KindN[*TypeVar*(*_ReaderLike3Type*, bound= *ReaderLike3*), *TypeVar*(*_ValueType*), *TypeVar*(*_SecondType*), *TypeVar*(*_EnvType*)]

class CallableReader3

Bases: *ReaderLike3*[*_FirstType*, *_SecondType*, *_ThirdType*], *Contextable*[*_ValueType*, *_EnvType*]

Intermediate interface for *ReaderLike3* + `__call__` method.

Has 5 type variables to type *Reader* and `__call__` independently. Since, we don't have any other fancy ways of doing it.

Should not be used directly other than defining your own *Reader* interfaces.

final class _LawSpec

Bases: *LawSpecDef*

Concrete laws for *ReaderBased2*.

See: <https://github.com/haskell/mtl/pull/61/files>

static purity_law(*container*, *env*)

Calling a *Reader* twice has the same result with the same *env*.

Parameters

- **container** (*ReaderBased2*[*TypeVar*(*_FirstType*), *TypeVar*(*_SecondType*)]) –
- **env** (*TypeVar*(*_SecondType*)) –

Return type

None

static asking_law(*container*, *env*)

Asking for an *env*, always returns the *env*.

Parameters

- **container** (*ReaderBased2*[*TypeVar*(*_FirstType*), *TypeVar*(*_SecondType*)]) –
- **env** (*TypeVar*(*_SecondType*)) –

Return type

None

class ReaderBased2

Bases: *CallableReader2*[*_FirstType*, *_SecondType*, *_FirstType*, *_SecondType*], *Lawful*[*ReaderBased2*[*_FirstType*, *_SecondType*]]

This interface is very specific to our *Reader* type.

The only thing that differs from *ReaderLike2* is that we know the specific types for its `__call__` method.

`_laws`: `ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law2 object>, <returns.primitives.laws.Law2 object>)`

Some classes and interfaces might have laws, some might not have any.

ReaderResult specific

class ReaderResultLikeN

Bases: `ReaderLike3[_FirstType, _SecondType, _ThirdType]`, `ResultLikeN[_FirstType, _SecondType, _ThirdType]`

Base interface for all types that do look like `ReaderResult` instance.

Cannot be called.

abstract bind_context_result(function)

Binds a `ReaderResult` returning function over a container.

Parameters

- **self** (`TypeVar[_ReaderResultLikeType, bound= ReaderResultLikeN]`) –
- **function** (`Callable[[TypeVar(_FirstType)], RequiresContextResult[TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]`) –

Return type

`KindN[TypeVar(_ReaderResultLikeType, bound= ReaderResultLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]`

abstract classmethod from_failed_context(inner_value)

Unit method to create new containers from failed `Reader`.

Parameters

inner_value (`RequiresContext[TypeVar(_ErrorType), TypeVar(_EnvType)]`) –

Return type

`KindN[TypeVar(_ReaderResultLikeType, bound= ReaderResultLikeN), TypeVar(_FirstType), TypeVar(_ErrorType), TypeVar(_EnvType)]`

abstract classmethod from_result_context(inner_value)

Unit method to create new containers from `ReaderResult`.

Parameters

inner_value (`RequiresContextResult[TypeVar(_ValueType), TypeVar(_ErrorType), TypeVar(_EnvType)]`) –

Return type

`KindN[TypeVar(_ReaderResultLikeType, bound= ReaderResultLikeN), TypeVar(_ValueType), TypeVar(_ErrorType), TypeVar(_EnvType)]`

ReaderResultLike3

Type alias for kinds with three type arguments.

alias of `ReaderResultLikeN[_FirstType, _SecondType, _ThirdType]`

final class _LawSpec

Bases: `LawSpecDef`

Concrete laws for `ReaderResultBasedN`.

See: <https://github.com/haskell/mtl/pull/61/files>

static purity_law(container, env)

Calling a `Reader` twice has the same result with the same `env`.

Parameters

- **container** (*ReaderResultBasedN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **env** (TypeVar(_ThirdType)) –

Return type

None

static asking_law(*container*, *env*)

Asking for an env, always returns the env.

Parameters

- **container** (*ReaderResultBasedN*[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **env** (TypeVar(_ThirdType)) –

Return type

None

class ReaderResultBasedN

Bases: *ReaderResultLikeN*[_FirstType, _SecondType, _ThirdType], *CallableReader3*[_FirstType, _SecondType, _ThirdType, Result[_FirstType, _SecondType], _ThirdType], *Lawful*[*ReaderResultBasedN*[_FirstType, _SecondType, _ThirdType]]

This interface is very specific to our *ReaderResult* type.

The only thing that differs from *ReaderResultLikeN* is that we know the specific types for its `__call__` method.

In this case the return type of `__call__` is *Result*.

`_laws: ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law2 object>, <returns.primitives.laws.Law2 object>)`

Some classes and interfaces might have laws, some might not have any.

ReaderResultBased3

Type alias for kinds with three type arguments.

alias of *ReaderResultBasedN*[_FirstType, _SecondType, _ThirdType]

ReaderIOResult specific**class ReaderIOResultLikeN**

Bases: *ReaderResultLikeN*[_FirstType, _SecondType, _ThirdType], *IOResultLikeN*[_FirstType, _SecondType, _ThirdType]

Base interface for all types that do look like *ReaderIOResult* instance.

Cannot be called.

abstract bind_context_ioresult(*function*)

Binds a *ReaderIOResult* returning function over a container.

Parameters

- **self** (TypeVar(_ReaderIOResultLikeType, bound= ReaderIOResultLikeN)) –
- **function** (Callable[[TypeVar(_FirstType)], *RequiresContextIOResult*[TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]) –

Return type

KindN[TypeVar(_ReaderIOResultLikeType, bound= ReaderIOResultLikeN),
TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]

abstract classmethod `from_ioresult_context`(*inner_value*)

Unit method to create new containers from ReaderIOResult.

Parameters

inner_value (*RequiresContextIOResult*[TypeVar(_ValueType),
TypeVar(_ErrorType), TypeVar(_EnvType)]) –

Return type

KindN[TypeVar(_ReaderIOResultLikeType, bound= ReaderIOResultLikeN),
TypeVar(_ValueType), TypeVar(_ErrorType), TypeVar(_EnvType)]

ReaderIOResultLike3

Type alias for kinds with three type arguments.

alias of *ReaderIOResultLikeN*[_FirstType, _SecondType, _ThirdType]

final class `_LawSpec`

Bases: *LawSpecDef*

Concrete laws for ReaderIOResultBasedN.

See: <https://github.com/haskell/mtl/pull/61/files>

static asking_law(*container*, *env*)

Asking for an env, always returns the env.

Parameters

- **container** (*ReaderIOResultBasedN*[TypeVar(_FirstType),
TypeVar(_SecondType), TypeVar(_ThirdType)]) –
- **env** (TypeVar(_ThirdType)) –

Return type

None

class `ReaderIOResultBasedN`

Bases: *ReaderIOResultLikeN*[_FirstType, _SecondType, _ThirdType],
CallableReader3[_FirstType, _SecondType, _ThirdType, IOResult[_FirstType, _SecondType],
_ThirdType], *Lawful*[ReaderIOResultBasedN[_FirstType, _SecondType, _ThirdType]]

This interface is very specific to our ReaderIOResult type.

The only thing that differs from ReaderIOResultLikeN is that we know the specific types for its `__call__` method.

In this case the return type of `__call__` is IOResult.

`_laws: ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law2 object>,)`

Some classes and interfaces might have laws, some might not have any.

ReaderIOResultBased3

Type alias for kinds with three type arguments.

alias of *ReaderIOResultBasedN*[_FirstType, _SecondType, _ThirdType]

ReaderFutureResult specific

class ReaderFutureResultLikeN

Bases: `ReaderIOResultLikeN[_FirstType, _SecondType, _ThirdType]`,
`FutureResultLikeN[_FirstType, _SecondType, _ThirdType]`

Interface for all types that do look like `ReaderFutureResult` instance.

Cannot be called.

abstract bind_context_future_result(function)

Bind a `ReaderFutureResult` returning function over a container.

Parameters

- **self** (`TypeVar[_ReaderFutureResultLikeType, bound= ReaderFutureResultLikeN]`) –
- **function** (`Callable[[TypeVar(_FirstType)], RequiresContextFutureResult[TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]`) –

Return type

`KindN[TypeVar(_ReaderFutureResultLikeType, bound= ReaderFutureResultLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]`

abstract bind_async_context_future_result(function)

Bind async `ReaderFutureResult` function.

Parameters

- **self** (`TypeVar[_ReaderFutureResultLikeType, bound= ReaderFutureResultLikeN]`) –
- **function** (`Callable[[TypeVar(_FirstType)], Awaitable[RequiresContextFutureResult[TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]`) –

Return type

`KindN[TypeVar(_ReaderFutureResultLikeType, bound= ReaderFutureResultLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]`

abstract classmethod from_future_result_context(inner_value)

Unit method to create new containers from `ReaderFutureResult`.

Parameters

inner_value (`RequiresContextFutureResult[TypeVar(_ValueType), TypeVar(_ErrorType), TypeVar(_EnvType)]`) –

Return type

`KindN[TypeVar(_ReaderFutureResultLikeType, bound= ReaderFutureResultLikeN), TypeVar(_ValueType), TypeVar(_ErrorType), TypeVar(_EnvType)]`

ReaderFutureResultLike3

Type alias for kinds with three type arguments.

alias of `ReaderFutureResultLikeN[_FirstType, _SecondType, _ThirdType]`

final class _LawSpec

Bases: `LawSpecDef`

Concrete laws for `ReaderFutureResultBasedN`.

See: <https://github.com/haskell/mtl/pull/61/files>

```
static asking_law(container, env)
```

Asking for an env, always returns the env.

Parameters

- **container** (*ReaderFutureResultBasedN*[*TypeVar*(*_FirstType*), *TypeVar*(*_SecondType*), *TypeVar*(*_ThirdType*)])
- **env** (*TypeVar*(*_ThirdType*))

Return type

None

class ReaderFutureResultBasedN

Bases: *ReaderFutureResultLikeN*[*_FirstType*, *_SecondType*, *_ThirdType*], *CallableReader3*[*_FirstType*, *_SecondType*, *_ThirdType*, *FutureResult*[*_FirstType*, *_SecondType*], *_ThirdType*], *Lawful*[*ReaderFutureResultBasedN*[*_FirstType*, *_SecondType*, *_ThirdType*]]

This interface is very specific to our *ReaderFutureResult* type.

The only thing that differs from *ReaderFutureResultLikeN* is that we know the specific types for its `__call__` method.

In this case the return type of `__call__` is *FutureResult*.

```
_laws: ClassVar[Sequence[Law]] = (<returns.primitives.laws.Law2 object>,) )
```

Some classes and interfaces might have laws, some might not have any.

ReaderFutureResultBased3

Type alias for kinds with three type arguments.

alias of *ReaderFutureResultBasedN*[*_FirstType*, *_SecondType*, *_ThirdType*]

9.6 Maybe

The *Maybe* container is used when a series of computations could return *None* at any point.

9.6.1 Maybe container

Maybe consist of two types: *Some* and *Nothing*. We have a convenient method to create different *Maybe* types based on just a single value:

```
>>> from returns.maybe import Maybe
>>> assert str(Maybe.from_optional(1)) == '<Some: 1>'
>>> assert str(Maybe.from_optional(None)) == '<Nothing>'
```

We also have another method called `.from_value` that behaves a bit differently:

```
>>> from returns.maybe import Maybe
>>> assert str(Maybe.from_value(1)) == '<Some: 1>'
>>> assert str(Maybe.from_value(None)) == '<Some: None>'
```

Usage

It might be very useful for complex operations like the following one:

```
>>> from attr import dataclass
>>> from typing import Optional
>>> from returns.maybe import Maybe, Nothing

>>> @dataclass
... class Address(object):
...     street: Optional[str]

>>> @dataclass
... class User(object):
...     address: Optional[Address]

>>> @dataclass
... class Order(object):
...     user: Optional[User]

>>> def get_street_address(order: Order) -> Maybe[str]:
...     return Maybe.from_optional(order.user).bind_optional(
...         lambda user: user.address,
...     ).bind_optional(
...         lambda address: address.street,
...     )

>>> with_address = Order(User(Address('Some street')))
>>> empty_user = Order(None)
>>> empty_address = Order(User(None))
>>> empty_street = Order(User(Address(None)))

>>> str(get_street_address(with_address)) # all fields are not None
'<Some: Some street>'

>>> assert get_street_address(empty_user) == Nothing
>>> assert get_street_address(empty_address) == Nothing
>>> assert get_street_address(empty_street) == Nothing
```

Optional type

One may ask: “How is that different to the `Optional[]` type?” That’s a really good question!

Consider the same code to get the street name without `Maybe` and using raw `Optional` values:

```
order: Order # some existing Order instance
street: Optional[str] = None
if order.user is not None:
    if order.user.address is not None:
        street = order.user.address.street
```

It looks way uglier and can grow even more uglier and complex when new logic will be introduced.

9.6.2 Pattern Matching

Maybe values can be matched using the new feature of Python 3.10, [Structural Pattern Matching](#), see the example below:

```
from dataclasses import dataclass
from typing import Final

from returns.maybe import Maybe, Nothing, Some

@dataclass
class _Book(object):
    book_id: int
    name: str

_BOOK_LIST: Final = (
    _Book(book_id=1, name='Category Theory for Programmers'),
    _Book(book_id=2, name='Fluent Python'),
    _Book(book_id=3, name='Learn You Some Erlang for Great Good'),
    _Book(book_id=4, name='Learn You a Haskell for Great Good'),
)

def _find_book(book_id: int) -> Maybe[_Book]:
    for book in _BOOK_LIST:
        if book.book_id == book_id:
            return Some(book)
    return Nothing

if __name__ == '__main__':
    desired_book = _find_book(2)
    match desired_book:
        # Matches any `Some` instance that contains a book named `Fluent Python`
        case Some(_Book(name='Fluent Python')):
            print('"Fluent Python" was found')

        # Matches any `Some` instance and binds its value to the `book` variable
        case Some(book):
            print('Book found: {0}'.format(book.name))

        # Matches `Nothing` instance
        case Maybe.empty:
            print('Not found the desired book!')
```


9.6.3 Decorators

Limitations

Typing will only work correctly if *our mypy plugin* is used. This happens due to [mypy issue](#).

maybe

Sometimes we have to deal with functions that dears to return `Optional` values!

We have to work with it the carefully and write `if x is not None:` everywhere. Luckily, we have your back! `maybe` function decorates any other function that returns `Optional` and converts it to return `Maybe` instead:

```
>>> from typing import Optional
>>> from returns.maybe import Maybe, Some, maybe

>>> @maybe
... def number(num: int) -> Optional[int]:
...     if num > 0:
...         return num
...     return None

>>> result: Maybe[int] = number(1)
>>> assert result == Some(1)
```

9.6.4 FAQ

How can I turn `Maybe` into `Optional` again?

When working with regular Python, you might need regular `Optional[a]` values.

You can easily get one from your `Maybe` container at any point in time:

```
>>> from returns.maybe import Maybe
>>> assert Maybe.from_optional(1).value_or(None) == 1
>>> assert Maybe.from_optional(None).value_or(None) == None
```

As you can see, revealed type of `.value_or(None)` is `Optional[a]`. Use it a fallback.

How to model absence of value vs presence of `None` value?

Let's say you have this dict: `values = {'a': 1, 'b': None}` So, you can have two types of `None` here:

- `values.get('b')`
- `values.get('c')`

But, they are different! The first has explicit `None` value, the second one has no given key and `None` is used as a default. You might need to know exactly which case you are dealing with. For example, in validation.

So, the first thing to remember is that:

```
>>> assert Some(None) != Nothing
```

There's a special way to work with a type like this:

```
>>> values = {'a': 1, 'b': None}
>>> assert Maybe.from_value(values).map(lambda d: d.get('a')) == Some(1)
>>> assert Maybe.from_value(values).map(lambda d: d.get('b')) == Some(None)
```

In contrast, you can ignore both None values easily:

```
>>> assert Maybe.from_value(values).bind_optional(
...     lambda d: d.get('a'),
... ) == Some(1)
>>> assert Maybe.from_value(values).bind_optional(
...     lambda d: d.get('b'),
... ) == Nothing
```

So, how to write a complete check for a value: both present and missing?

```
>>> from typing import Optional, Dict, TypeVar
>>> from returns.maybe import Maybe, Some, Nothing

>>> _Key = TypeVar('_Key')
>>> _Value = TypeVar('_Value')

>>> def check_key(
...     haystack: Dict[_Key, _Value],
...     needle: _Key,
... ) -> Maybe[_Value]:
...     if needle not in haystack:
...         return Nothing
...     return Maybe.from_value(haystack[needle]) # try with `.from_optional`

>>> real_values = {'a': 1}
>>> opt_values = {'a': 1, 'b': None}

>>> assert check_key(real_values, 'a') == Some(1)
>>> assert check_key(real_values, 'b') == Nothing
>>> # Type revealed: returns.maybe.Maybe[builtins.int]

>>> assert check_key(opt_values, 'a') == Some(1)
>>> assert check_key(opt_values, 'b') == Some(None)
>>> assert check_key(opt_values, 'c') == Nothing
>>> # Type revealed: returns.maybe.Maybe[Union[builtins.int, None]]
```

Choose wisely between `.from_value` and `.map`, and `.from_optional` and `.bind_optional`. They are similar, but do different things.

Note that you can also use `returns.pipeline.is_successful()` to check if the value is present.

See the [original issue](#) about `Some(None)` for more details and the full history.

Why there's no IOMaybe?

We do have `IOResult`, but we don't have `IOMaybe`. Why? Because when dealing with IO there are a lot of possible errors. And `Maybe` represents just `None` and the value.

It is not useful for IO related tasks. So, use `Result` instead, which can represent what happened to your IO.

You can convert `Maybe` to `Result` and back again with special *Converters*.

Why Maybe does not have alt method?

Well, because `Maybe` only has a single failed value: `Nothing` and it cannot be altered.

But, `Maybe` has `returns.maybe.Maybe.or_else_call()` method to call a passed callback function with zero argument on failed container:

```
>>> from returns.maybe import Some, Nothing
>>> assert Some(1).or_else_call(lambda: 2) == 1
>>> assert Nothing.or_else_call(lambda: 2) == 2
```

This method is unique to `Maybe` container.

9.6.5 Further reading

- Option Monads in Rust
- Option overview in TypeScript
- Maybe not - Rich Hickey

9.6.6 API Reference

class `Maybe`(*inner_value*)

Bases: `BaseContainer`, `SupportsKindN[Maybe, _ValueType, NoReturn, NoReturn]`, `MaybeBasedN[_ValueType, None, NoReturn]`

Represents a result of a series of computations that can return `None`.

An alternative to using exceptions or constant `is None` checks. `Maybe` is an abstract type and should not be instantiated directly. Instead use `Some` and `Nothing`.

See also:

- <https://github.com/gcanti/fp-ts/blob/master/docs/modules/Option.ts.md>

empty: `ClassVar[Maybe[Any]] = <Nothing>`

Alias for `Nothing`

success_type

Success type that is used to represent the successful computation.

Parameters

inner_value (`TypeVar[_ValueType, covariant=True]`) –

alias of `Some`

failure_type

Failure type that is used to represent the failed computation.

Parameters

inner_value (None) –

alias of `_Nothing`

equals(*other*)

Typesafe equality comparison with other *Result* objects.

Parameters

- **self** (*KindN*[*TypeVar*(`_EqualType`, bound= *Equable*), Any, Any, Any]) –
- **other** (*KindN*[*TypeVar*(`_EqualType`, bound= *Equable*), Any, Any, Any]) –

Return type

`bool`

map(*function*)

Composes successful container with a pure function.

```
>>> from returns.maybe import Some, Nothing
>>> def mappable(string: str) -> str:
...     return string + 'b'

>>> assert Some('a').map(mappable) == Some('ab')
>>> assert Nothing.map(mappable) == Nothing
```

Parameters

function (*Callable*[[*TypeVar*(`_ValueType`, covariant=True), *TypeVar*(`_NewValueType`)]) –

Return type

Maybe[*TypeVar*(`_NewValueType`)]

apply(*function*)

Calls a wrapped function in a container on this container.

```
>>> from returns.maybe import Some, Nothing

>>> def applicable(string: str) -> str:
...     return string + 'b'

>>> assert Some('a').apply(Some(applicable)) == Some('ab')
>>> assert Some('a').apply(Nothing) == Nothing
>>> assert Nothing.apply(Some(applicable)) == Nothing
>>> assert Nothing.apply(Nothing) == Nothing
```

Parameters

function (*KindN*[*Maybe*, *Callable*[[*TypeVar*(`_ValueType`, covariant=True), *TypeVar*(`_NewValueType`)], Any, Any]) –

Return type

Maybe[*TypeVar*(`_NewValueType`)]

bind(*function*)

Composes successful container with a function that returns a container.

```
>>> from returns.maybe import Nothing, Maybe, Some
>>> def bindable(string: str) -> Maybe[str]:
...     return Some(string + 'b')

>>> assert Some('a').bind(bindable) == Some('ab')
>>> assert Nothing.bind(bindable) == Nothing
```

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)], *KindN[Maybe,*
TypeVar(_NewValueType), Any, Any]) –

Return type

Maybe[TypeVar(_NewValueType)]

bind_optional(*function*)

Binds a function returning an optional value over a container.

```
>>> from returns.maybe import Some, Nothing
>>> from typing import Optional

>>> def bindable(arg: str) -> Optional[int]:
...     return len(arg) if arg else None

>>> assert Some('a').bind_optional(bindable) == Some(1)
>>> assert Some('').bind_optional(bindable) == Nothing
```

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)],
Optional[TypeVar(_NewValueType)]) –

Return type

Maybe[TypeVar(_NewValueType)]

lash(*function*)

Composes failed container with a function that returns a container.

```
>>> from returns.maybe import Maybe, Some, Nothing

>>> def lashable(arg=None) -> Maybe[str]:
...     return Some('b')

>>> assert Some('a').lash(lashable) == Some('a')
>>> assert Nothing.lash(lashable) == Some('b')
```

We need this feature to make Maybe compatible with different Result like operations.

Parameters

function (Callable[[Any], *KindN[Maybe,* TypeVar(_ValueType), covariant=True), Any,
Any]) –

Return type

Maybe[TypeVar(_ValueType), covariant=True)]

classmethod `do(expr)`

Allows working with unwrapped values of containers in a safe way.

```
>>> from returns.maybe import Maybe, Some, Nothing

>>> assert Maybe.do(
...     first + second
...     for first in Some(2)
...     for second in Some(3)
... ) == Some(5)

>>> assert Maybe.do(
...     first + second
...     for first in Some(2)
...     for second in Nothing
... ) == Nothing
```

See *Do Notation* to learn more.

Parameters

expr (Generator[TypeVar(_NewValueType), None, None]) –

Return type

Maybe[TypeVar(_NewValueType)]

value_or(*default_value*)

Get value from successful container or default value from failed one.

```
>>> from returns.maybe import Nothing, Some
>>> assert Some(0).value_or(1) == 0
>>> assert Nothing.value_or(1) == 1
```

Parameters

default_value (TypeVar(_NewValueType)) –

Return type

Union[TypeVar(_ValueType, covariant=True), TypeVar(_NewValueType)]

or_else_call(*function*)

Get value from successful container or default value from failed one.

Really close to `value_or()` but works with lazy values. This method is unique to Maybe container, because other containers do have `.alt` method.

But, Maybe does not have this method. There's nothing to `alt` in Nothing.

Instead, it has this method to execute some function if called on a failed container:

```
>>> from returns.maybe import Some, Nothing
>>> assert Some(1).or_else_call(lambda: 2) == 1
>>> assert Nothing.or_else_call(lambda: 2) == 2
```

It might be useful to work with exceptions as well:

```
>>> def fallback() -> NoReturn:
...     raise ValueError('Nothing!')
```

(continues on next page)

(continued from previous page)

```
>>> Nothing.or_else_call(fallback)
Traceback (most recent call last):
...
ValueError: Nothing!
```

Parameters

function (Callable[[], TypeVar(_NewValueType)]) –

Return type

Union[TypeVar(_ValueType, covariant=True), TypeVar(_NewValueType)]

unwrap()

Get value from successful container or raise exception for failed one.

```
>>> from returns.maybe import Nothing, Some
>>> assert Some(1).unwrap() == 1

>>> Nothing.unwrap()
Traceback (most recent call last):
...
returns.primitives.exceptions.UnwrapFailedError
```

Return type

TypeVar(_ValueType, covariant=True)

failure()

Get failed value from failed container or raise exception from success.

```
>>> from returns.maybe import Nothing, Some
>>> assert Nothing.failure() is None

>>> Some(1).failure()
Traceback (most recent call last):
...
returns.primitives.exceptions.UnwrapFailedError
```

Return type

None

classmethod from_value(inner_value)

Creates new instance of Maybe container based on a value.

```
>>> from returns.maybe import Maybe, Some
>>> assert Maybe.from_value(1) == Some(1)
>>> assert Maybe.from_value(None) == Some(None)
```

Parameters

inner_value (TypeVar(_NewValueType)) –

Return type

Maybe[TypeVar(_NewValueType)]

classmethod `from_optional(inner_value)`

Creates new instance of Maybe container based on an optional value.

```
>>> from returns.maybe import Maybe, Some, Nothing
>>> assert Maybe.from_optional(1) == Some(1)
>>> assert Maybe.from_optional(None) == Nothing
```

Parameters

inner_value (Optional[TypeVar(_NewValueType)]) –

Return type

Maybe[TypeVar(_NewValueType)]

final class `Some(inner_value)`

Bases: *Maybe*[_ValueType]

Represents a calculation which has succeeded and contains the value.

Quite similar to Success type.

Parameters

inner_value (TypeVar(_ValueType, covariant=True)) –

bind(*function*)

Binds current container to a function that returns container.

bind_optional(*function*)

Binds a function returning an optional value over a container.

unwrap()

Returns inner value for successful container.

map(*function*)

Composes current container with a pure function.

apply(*container*)

Calls a wrapped function in a container on this container.

lash(*function*)

Does nothing for Some.

value_or(*default_value*)

Returns inner value for successful container.

or_else_call(*function*)

Returns inner value for successful container.

failure()

Raises exception for successful container.

Nothing: *Maybe*[NoReturn] = <Nothing>

Public unit value of protected _Nothing type.

maybe(*function*)

Decorator to convert None-returning function to Maybe container.

This decorator works with sync functions only. Example:


```

>>> from typing import Optional
>>> from returns.maybe import Nothing, Some, maybe

>>> @maybe
... def might_be_none(arg: int) -> Optional[int]:
...     if arg == 0:
...         return None
...     return 1 / arg

>>> assert might_be_none(0) == Nothing
>>> assert might_be_none(1) == Some(1.0)

```

Parameters

function (Callable[[ParamSpec(_FuncParams)], Optional[TypeVar(_ValueType, covariant=True)]]) –

Return type

Callable[[ParamSpec(_FuncParams)], *Maybe*[TypeVar(_ValueType, covariant=True)]]

9.7 Result

Make sure to get familiar with *Railway oriented programming*.

Result is obviously a result of some series of computations. It might succeed with some resulting value. Or it might return an error with some extra details.

Result consist of two types: Success and Failure. Success represents successful operation result and Failure indicates that something has failed.

```

from returns.result import Result, Success, Failure

def find_user(user_id: int) -> Result['User', str]:
    user = User.objects.filter(id=user_id)
    if user.exists():
        return Success(user[0])
    return Failure('User was not found')

user_search_result = find_user(1)
# => Success(User{id: 1, ...})

user_search_result = find_user(0) # id 0 does not exist!
# => Failure('User was not found')

```

When is it useful? When you do not want to use exceptions to break your execution scope. Or when you do not want to use None to represent empty values, since it will raise `TypeError` somewhere and other None exception-friends.

9.7.1 Composition

Make sure to check out how to compose container with `flow` or *pipe*! Read more about them if you want to compose your containers easily.

9.7.2 Pattern Matching

Result values can be matched using the new feature of Python 3.10, [Structural Pattern Matching](#), see the example below:

```
from returns.result import Failure, Success, safe

@safe
def div(first_number: int, second_number: int) -> int:
    return first_number // second_number

match div(1, 0):
    # Matches if the result stored inside `Success` is `10`
    case Success(10):
        print('Result is "10"')

    # Matches any `Success` instance and binds its value to the `value` variable
    case Success(value):
        print('Result is "{0}"'.format(value))

    # Matches if the result stored inside `Failure` is `ZeroDivisionError`
    case Failure(ZeroDivisionError()):
        print('"ZeroDivisionError" was raised')

    # Matches any `Failure` instance
    case Failure(_):
        print('The division was a failure')
```

9.7.3 Aliases

There are several useful aliases for `Result` type with some common values:

- `returns.result.ResultE` is an alias for `Result[... Exception]`, just use it when you want to work with `Result` containers that use exceptions as error type. It is named `ResultE` because it is `ResultException` and `ResultError` at the same time.

9.7.4 Decorators

Limitations

Typing will only work correctly if *our mypy plugin* is used. This happens due to *mypy issue*.

safe

safe is used to convert regular functions that can throw exceptions to functions that return *Result* type.

Supports only regular functions. If you need to mark async functions as *safe*, use *future_safe* instead.

```
>>> from returns.result import Success, safe

>>> @safe # Will convert type to: Callable[[int], Result[float, Exception]]
... def divide(number: int) -> float:
...     return number / number

>>> assert divide(1) == Success(1.0)
>>> str(divide(0))
'<Failure: division by zero>'
```

If you want to *safe* handle only a set of exceptions:

```
>>> @safe(exceptions=(ZeroDivisionError,)) # Other exceptions will be raised
... def divide(number: int) -> float:
...     if number > 10:
...         raise ValueError('Too big')
...     return number / number

>>> assert divide(5) == Success(1.0)
>>> assert divide(0).failure()
>>> divide(15)
Traceback (most recent call last):
...
ValueError: Too big
```

attempt

Similar to *safe* function but instead of wrapping the exception error in a *Failure* container it'll wrap the argument that lead to that exception.

```
>>> from returns.result import Failure, Success, attempt

>>> @attempt
... def divide_itself(number: int) -> float:
...     return number / number

>>> assert divide_itself(2) == Success(1.0)
>>> assert divide_itself(0) == Failure(0)
```

Warning: This decorator works only with functions that has just one argument.

9.7.5 FAQ

How to create unit objects?

Use `Success` or `Failure`. Alternatively `returns.result.Result.from_value()` or `returns.result.Result.from_failure()`.

It might be a good idea to use unit functions together with the explicit annotation. Python's type system does not allow us to do much, so this is required:

```
>>> from returns.result import Result, Success
>>> def callback(arg: int) -> Result[float, int]:
...     return Success(float(arg))
>>> first: Result[int, int] = Success(1)
>>> assert first.bind(callback) == Success(1.0)
```

Otherwise `first` will have `Result[int, Any]` type. Which is okay in some situations.

How to compose error types?

You might want to sometimes use `unify` *Pointfree* functions instead of `.bind` to compose error types together. While `.bind` enforces error type to stay the same, `unify` is designed to return a `Union` of a previous error type and a new one.

It gives an extra flexibility, but also provokes more thinking and can be problematic in some cases.

Like so:

```
>>> from returns.result import Result, Success, Failure
>>> from returns.pointfree import unify
>>> def div(number: int) -> Result[float, ZeroDivisionError]:
...     if number:
...         return Success(1 / number)
...     return Failure(ZeroDivisionError('division by zero'))
>>> container: Result[int, ValueError] = Success(1)
>>> assert unify(div)(container) == Success(1.0)
>>> # => Revealed type is:
>>> # Result[float, Union[ValueError, ZeroDivisionError]]
```

So, that's a way to go, if you need this composition.

map vs bind

We use the `map` method when we're working with pure functions, a function is pure if it doesn't produce any side-effect (e.g. Exceptions). On the other hand, we use the `bind` method if a function returns a `Result` instance which translates its potential side-effect into a raw value. See the example below:

```
>>> import json
>>> from typing import Dict

>>> from returns.result import Failure, Result, Success, safe

>>> # `cast_to_bool` doesn't produce any side-effect
>>> def cast_to_bool(arg: int) -> bool:
...     return bool(arg)

>>> # `parse_json` can produce Exceptions, so we use the `safe` decorator
>>> # to prevent any kind of exceptions
>>> @safe
... def parse_json(arg: str) -> Dict[str, str]:
...     return json.loads(arg)

>>> assert Success(1).map(cast_to_bool) == Success(True)
>>> assert Success({'example': 'example'}).bind(parse_json) == Success({'example':
↳ "example"})
>>> assert Success('').bind(parse_json).alt(str) == Failure('Expecting value: line 1,
↳ column 1 (char 0)')
```

How to check if your result is a success or failure?

`Result` is a container and you can use `returns.pipeline.is_successful()` like so:

```
>>> from returns.result import Success, Failure
>>> from returns.pipeline import is_successful

>>> assert is_successful(Success(1)) is True
>>> assert is_successful(Failure('text')) is False
```

9.7.6 Further reading

- Railway Oriented Programming
- Recoverable Errors with Result in Rust
- Either overview in TypeScript

9.7.7 API Reference

class `Result`(*inner_value*)

Bases: `BaseContainer`, `SupportsKindN[Result, _ValueType, _ErrorType, NoReturn]`, `ResultBasedN[_ValueType, _ErrorType, NoReturn]`

Base class for `Failure` and `Success`.

`Result` does not have a public constructor. Use `Success()` and `Failure()` to construct the needed values.

See also:

- <https://bit.ly/361qQhi>
- <https://hackernoon.com/the-throw-keyword-was-a-mistake-19e532di>

success_type

Success type that is used to represent the successful computation.

Parameters

inner_value (`TypeVar(_ValueType, covariant=True)`) –

alias of `Success`

failure_type

Failure type that is used to represent the failed computation.

Parameters

inner_value (`TypeVar(_ErrorType, covariant=True)`) –

alias of `Failure`

equals(*other*)

Typesafe equality comparison with other `Result` objects.

Parameters

- **self** (`KindN[TypeVar(_EqualType, bound=Equable), Any, Any, Any]`) –
- **other** (`KindN[TypeVar(_EqualType, bound=Equable), Any, Any, Any]`) –

Return type

`bool`

property trace: `Optional[List[FrameInfo]]`

Returns a list with stack trace when `Failure()` was called.

swap()

Swaps value and error types.

So, values become errors and errors become values. It is useful when you have to work with errors a lot. And since we have a lot of `.bind_` related methods and only a single `.lash` - it is easier to work with values.

```
>>> from returns.result import Success, Failure
>>> assert Success(1).swap() == Failure(1)
>>> assert Failure(1).swap() == Success(1)
```

Return type

`Result[TypeVar(_ErrorType, covariant=True), TypeVar(_ValueType, covariant=True)]`

map(*function*)

Composes successful container with a pure function.

```
>>> from returns.result import Failure, Success

>>> def mappable(string: str) -> str:
...     return string + 'b'

>>> assert Success('a').map(mappable) == Success('ab')
>>> assert Failure('a').map(mappable) == Failure('a')
```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)]], covariant=True) –

Return type

Result[TypeVar(_NewValueType), TypeVar(_ErrorType), covariant=True]

apply(*container*)

Calls a wrapped function in a container on this container.

```
>>> from returns.result import Failure, Success

>>> def applicable(string: str) -> str:
...     return string + 'b'

>>> assert Success('a').apply(Success(applicable)) == Success('ab')
>>> assert Failure('a').apply(Success(applicable)) == Failure('a')

>>> assert Success('a').apply(Failure(1)) == Failure(1)
>>> assert Failure(1).apply(Failure(2)) == Failure(1)
```

Parameters

container (*KindN*[*Result*, Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)], TypeVar(_ErrorType), Any]] –

Return type

Result[TypeVar(_NewValueType), TypeVar(_ErrorType), covariant=True]

bind(*function*)

Composes successful container with a function that returns a container.

```
>>> from returns.result import Result, Success, Failure

>>> def bindable(arg: str) -> Result[str, str]:
...     if len(arg) > 1:
...         return Success(arg + 'b')
...     return Failure(arg + 'c')

>>> assert Success('aa').bind(bindable) == Success('aab')
>>> assert Success('a').bind(bindable) == Failure('ac')
>>> assert Failure('a').bind(bindable) == Failure('a')
```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], KindN[Result, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), Any]]) –

Return type

Result[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_result(function)

Alias for `bind_result` method, it is the same as `bind` here.

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], KindN[Result, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), Any]]) –

Return type

Result[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

alt(function)

Composes failed container with a pure function to modify failure.

```
>>> from returns.result import Failure, Success
>>> def altable(arg: str) -> str:
...     return arg + 'b'
>>> assert Success('a').alt(altable) == Success('a')
>>> assert Failure('a').alt(altable) == Failure('ab')
```

Parameters

function (Callable[[TypeVar(_ErrorType, covariant=True)], TypeVar(_NewErrorType)]) –

Return type

Result[TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType)]

lash(function)

Composes failed container with a function that returns a container.

```
>>> from returns.result import Result, Success, Failure
>>> def lashable(arg: str) -> Result[str, str]:
...     if len(arg) > 1:
...         return Success(arg + 'b')
...         return Failure(arg + 'c')
>>> assert Success('a').lash(lashable) == Success('a')
>>> assert Failure('a').lash(lashable) == Failure('ac')
>>> assert Failure('aa').lash(lashable) == Success('aab')
```

Parameters

function (Callable[[TypeVar(_ErrorType, covariant=True)], KindN[Result, TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType), Any]]) –

Return type

Result[TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType)]

classmethod `do(expr)`

Allows working with unwrapped values of containers in a safe way.

```
>>> from returns.result import Result, Failure, Success

>>> assert Result.do(
...     first + second
...     for first in Success(2)
...     for second in Success(3)
... ) == Success(5)

>>> assert Result.do(
...     first + second
...     for first in Failure('a')
...     for second in Success(3)
... ) == Failure('a')
```

See *Do Notation* to learn more. This feature requires our *mypy plugin*.

Parameters

expr (Generator[TypeVar(_NewValueType), None, None]) –

Return type

`Result`[TypeVar(_NewValueType), TypeVar(_NewErrorType)]

value_or(*default_value*)

Get value or default value.

```
>>> from returns.result import Failure, Success
>>> assert Success(1).value_or(2) == 1
>>> assert Failure(1).value_or(2) == 2
```

Parameters

default_value (TypeVar(_NewValueType)) –

Return type

Union[TypeVar(_ValueType, covariant=True), TypeVar(_NewValueType)]

unwrap()

Get value or raise exception.

```
>>> from returns.result import Failure, Success
>>> assert Success(1).unwrap() == 1

>>> Failure(1).unwrap()
Traceback (most recent call last):
...
returns.primitives.exceptions.UnwrapFailedError
```

Return type

TypeVar(_ValueType, covariant=True)

failure()

Get failed value or raise exception.

```

>>> from returns.result import Failure, Success
>>> assert Failure(1).failure() == 1

>>> Success(1).failure()
Traceback (most recent call last):
...
returns.primitives.exceptions.UnwrapFailedError

```

Return type

`TypeVar(_ErrorType, covariant=True)`

classmethod `from_value`(*inner_value*)

One more value to create success unit values.

It is useful as a united way to create a new value from any container.

```

>>> from returns.result import Result, Success
>>> assert Result.from_value(1) == Success(1)

```

You can use this method or `Success()`, choose the most convenient for you.

Parameters

inner_value (`TypeVar(_NewValueType)`) –

Return type

`Result`[`TypeVar(_NewValueType)`, `Any`]

classmethod `from_failure`(*inner_value*)

One more value to create failure unit values.

It is useful as a united way to create a new value from any container.

```

>>> from returns.result import Result, Failure
>>> assert Result.from_failure(1) == Failure(1)

```

You can use this method or `Failure()`, choose the most convenient for you.

Parameters

inner_value (`TypeVar(_NewErrorType)`) –

Return type

`Result`[`Any`, `TypeVar(_NewErrorType)`]

classmethod `from_result`(*inner_value*)

Creates a new `Result` instance from existing `Result` instance.

```

>>> from returns.result import Result, Failure, Success
>>> assert Result.from_result(Success(1)) == Success(1)
>>> assert Result.from_result(Failure(1)) == Failure(1)

```

This is a part of `returns.interfaces.specific.result.ResultBasedN` interface.

Parameters

inner_value (`Result`[`TypeVar(_NewValueType)`, `TypeVar(_NewErrorType)`]) –

Return type

`Result`[`TypeVar(_NewValueType)`, `TypeVar(_NewErrorType)`]

final class Failure(*inner_value*)

Bases: *Result*[Any, _ErrorType]

Represents a calculation which has failed.

It should contain an error code or message.

Parameters

inner_value (TypeVar(_ErrorType, covariant=True)) –

alt(*function*)

Composes failed container with a pure function to modify failure.

map(*function*)

Does nothing for Failure.

bind(*function*)

Does nothing for Failure.

bind_result(*function*)

Alias for *bind* method. Part of the *ResultBasedN* interface.

lash(*function*)

Composes this container with a function returning container.

apply(*container*)

Does nothing for Failure.

value_or(*default_value*)

Returns default value for failed container.

swap()

Failures swap to *Success*.

unwrap()

Raises an exception, since it does not have a value inside.

Return type

NoReturn

failure()

Returns failed value.

Return type

TypeVar(_ErrorType, covariant=True)

final class Success(*inner_value*)

Bases: *Result*[_ValueType, Any]

Represents a calculation which has succeeded and contains the result.

Contains the computation value.

Parameters

inner_value (TypeVar(_ValueType, covariant=True)) –

alt(*function*)

Does nothing for Success.

map(*function*)

Composes current container with a pure function.

bind(*function*)

Binds current container to a function that returns container.

bind_result(*function*)

Alias for *bind* method. Part of the *ResultBasedN* interface.

lash(*function*)

Does nothing for Success.

apply(*container*)

Calls a wrapped function in a container on this container.

value_or(*default_value*)

Returns the value for successful container.

swap()

Successes swap to *Failure*.

unwrap()

Returns the unwrapped value from successful container.

Return type

TypeVar(_ValueType, covariant=True)

failure()

Raises an exception for successful container.

Return type

NoReturn

ResultE

Alias for a popular case when Result has Exception as error type.

alias of `Result[_ValueType, Exception]`

safe(*function=None, exceptions=None*)

Decorator to convert exception-throwing function to Result container.

Should be used with care, since it only catches Exception subclasses. It does not catch BaseException subclasses.

If you need to mark async function as safe, use `returns.future.future_safe()` instead. This decorator only works with sync functions. Example:

```
>>> from returns.result import Result, Success, safe
>>> @safe
... def might_raise(arg: int) -> float:
...     return 1 / arg
>>> assert might_raise(1) == Success(1.0)
>>> assert isinstance(might_raise(0), Result.failure_type)
```

You can also use it with explicit exception types as the first argument:

```
>>> from returns.result import Result, Success, safe
>>> @safe(exceptions=(ZeroDivisionError,))
```

(continues on next page)

(continued from previous page)

```

... def might_raise(arg: int) -> float:
...     return 1 / arg

>>> assert might_raise(1) == Success(1.0)
>>> assert isinstance(might_raise(0), Result.failure_type)

```

In this case, only exceptions that are explicitly listed are going to be caught.

Similar to `returns.io.impure_safe()` and `returns.future.future_safe()` decorators.

Parameters

- **function** (Optional[Callable[[ParamSpec(_FuncParams)], TypeVar(_ValueType, covariant=True)]] –
- **exceptions** (Optional[Tuple[Type[Exception], ...]] –

Return type

Union[Callable[[ParamSpec(_FuncParams)], `Result`[TypeVar(_ValueType, covariant=True), Exception]], Callable[[Callable[[ParamSpec(_FuncParams)], TypeVar(_ValueType, covariant=True)]]], Callable[[ParamSpec(_FuncParams)], `Result`[TypeVar(_ValueType, covariant=True), Exception]]]

`attempt(func)`

Decorator to convert exception-throwing function to `Result` container.

It's very similar with `returns.result.safe()`, the difference is when an exception is raised it won't wrap that given exception into a `Failure`, it'll wrap the argument that lead to the exception.

```

>>> import json
>>> from typing import Dict, Any

>>> from returns.result import Failure, Success, attempt

>>> @attempt
... def parse_json(string: str) -> Dict[str, Any]:
...     return json.loads(string)

>>> assert parse_json('{"key": "value"}') == Success({'key': 'value'})
>>> assert parse_json('incorrect input') == Failure('incorrect input')

```

Parameters

func (Callable[[TypeVar(_FirstType)], TypeVar(_NewValueType)]) –

Return type

Callable[[TypeVar(_FirstType)], `Result`[TypeVar(_NewValueType), TypeVar(_FirstType)]]

9.8 IO

Mathematicians dream in pure functions. Each of them only relies on its arguments and always produces the same result for the same input.

That's not how a useful program works. We need to rely on the environment and we need to do side effects.

Furthermore, there are several types of IO in our programs:

- Some IO never fails, like: getting current date and time, random number, or OS name
- Some IO might fail, like: sending network requests, accessing filesystem, or database

There's a solution.

9.8.1 IO container

Once you have an IO operation you can mark it appropriately. We can use a simple class `returns.io.IO` to mark impure parts of the program that do not fail.

```
>>> import random
>>> from returns.io import IO
>>> def get_random_number() -> IO[int]:
...     return IO(random.randint(1, 10))
...
>>> assert isinstance(get_random_number(), IO)
```

And later we can work inside this IO context and do not break into our pure part of the program:

```
>>> assert get_random_number().map(lambda number: number / number) == IO(1.0)
```

And it infects all other functions that call it.

```
>>> def modify_number(number: int) -> IO[float]:
...     return get_random_number().map(lambda rnd: number / rnd)
...
>>> assert isinstance(modify_number(1), IO)
```

It is good enough to indicate that you are aware of side effects of the function.

9.8.2 IOResult

On the other hand, we can have IO parts of the program that do fail.

Imagine we have this beautiful pure function:

```
def can_book_seats(
    number_of_seats: int,
    reservation: 'Reservation',
) -> bool:
    return reservation.capacity >= number_of_seats + reservation.booked
```

What's good about it? We can test it easily. Even without setting up any testing framework, simple `doctests` will be enough.

This code is **beautiful**, because it is **simple**.

We can later use its result to process the result of the booking request:

```
def process_booking_result(is_successful: bool) -> 'ProcessID':
    ...

process_booking_result(is_successful) # works just fine!
```

At this point we don't have IO in our program.

Impure functions

But, imagine that our requirements had changed. And now we have to grab the number of already booked tickets from some other microservice and fetch the maximum capacity from the database:

```
import requests
import db

def can_book_seats(
    number_of_seats: int,
    place_id: int,
) -> bool:
    capacity = db.get_place_capacity(place_id) # sql query
    booked = requests('https://partner.com/api').json()['booked'] # http req
    return capacity >= number_of_seats + booked
```

Now testing this code will become a nightmare! It will require to setup:

- real database and tables
- fixture data
- requests mocks for different outcomes
- and the whole Universe!

Our complexity has sky-rocketed! And the most annoying part is that all other functions that call `can_book_seats` now also have to do the same setup. It seems like IO is indelible mark (some people also call it “effect”).

And at some point it time we will start to mix pure and impure code together. Let's not forget that all of these operations can fail too!

Separating two worlds

Well, IO mark is indeed indelible and should be respected.

And then impurity becomes explicit:

```
import requests
import db
from returns.io import IOResultE

def can_book_seats(
    number_of_seats: int,
    place_id: int,
) -> IOResultE[bool]:
    ...
```

Now this function returns `IOResultE[bool]` instead of a regular `bool`. It means, that it cannot be used where regular `bool` can be:

```
def process_booking_result(is_successful: bool) -> 'ProcessID':
  ...

is_successful: IOResultE[bool] = can_book_seats(number_of_seats, place_id)
process_booking_result(is_successful) # Boom!
# => Argument 1 has incompatible type "IOResultE[bool]"; expected "bool"
```

See? It is now impossible for a pure function to use `IOResultE[bool]`. It is impossible to unwrap or get a raw value from this container. Once it is marked as IO it will never return to the pure state (well, there's a hack actually: [unsafe_perform_io](#)).

Now we have to work inside the IO context:

```
message_id: IOResultE['ProcessID'] = can_book_seats(
  number_of_seats,
  place_id,
).map(
  process_booking_result,
)
```

Or it can be annotated to work with impure results:

```
def process_booking_result(
  is_successful: IOResultE[bool],
) -> IOResultE['ProcessID']:
  ...

is_successful: IOResult[bool] = can_book_seats(number_of_seats, place_id)
process_booking_result(is_successful) # Works!
```

Now, all our impurity is explicit. We can track it, we can fight it, we can design it better. By saying that, it is assumed that you have a functional core and imperative shell.

Lifting

You can also lift regular functions into one that works with IO or `IOResult` on both ends. It really helps you with the composition!

```
>>> from returns.io import IO
>>> from returns.pointfree import map_

>>> def regular_function(arg: int) -> float:
...     return arg / 2 # not an `IO` operation

>>> container = IO(1)
>>> # When we need to compose `regular_function` with `IO`,
>>> # we have two ways of doing it:
>>> io = container.map(regular_function)
>>> assert io == IO(0.5)

>>> # or, it is the same as:
```

(continues on next page)

(continued from previous page)

```
>>> io = map_(regular_function)(container)
>>> assert io == IO(0.5)
```

`IOResult` can lift both regular functions and ones that return `Result`:

```
>>> from returns.io import IOResult, IOSuccess
>>> from returns.pointfree import map_

>>> def regular_function(arg: int) -> float:
...     return arg / 2 # not an `IO` operation

>>> container: IOResult[int, str] = IOSuccess(1)
>>> # When we need to compose `regular_function` with `IOResult`,
>>> # we have two ways of doing it:
>>> io = container.map(regular_function)
>>> assert io == IOSuccess(0.5)

>>> # or, it is the same as:
>>> io = map_(regular_function)(container)
>>> assert io == IOSuccess(0.5)
```

And `Result` based functions:

```
>>> from returns.io import IOResult, IOSuccess
>>> from returns.result import Result, Success, Failure
>>> from returns.pointfree import bind_result

>>> def regular_function(arg: int) -> Result[float, str]:
...     if arg > 0:
...         return Success(arg / 2)
...     return Failure('zero')

>>> assert bind_result(regular_function)(
...     IOSuccess(1),
... ) == IOResult.from_result(regular_function(1))
```

Lifting is useful when using `returns.pipeline.pipe()` and other different declarative tools.

9.8.3 Pattern Matching

`IOResult` values can be matched using the new feature of Python 3.10, [Structural Pattern Matching](#), see the example below:

```
from returns.io import IOFailure, IOResult, IOSuccess
from returns.result import Success

container: IOResult[int, str] = IOSuccess(42)
match container:
    # Matches if the result stored inside `IOSuccess` is `42`
    # We need to use `Success` until the custom matching protocol
    # is released. For more information, please visit:
    # https://www.python.org/dev/peps/pep-0622/#custom-matching-protocol
```

(continues on next page)

```
case IOSuccess(Success(42)):
    print('Result is "42"')

# Matches any `IOSuccess` instance
# and binds its value to the `value` variable
case IOSuccess(value):
    print('Result is "{0}"'.format(value))

# Matches any `IOFailure` instance
case IOFailure(_):
    print('A failure was occurred')
```

9.8.4 Aliases

There are several useful aliases for `IOResult` type with some common values:

- `returns.io.IOResultE` is an alias for `IOResult[... Exception]`, just use it when you want to work with `IOResult` containers that use exceptions as error type. It is named `IOResultE` because it is `IOResultException` and `IOResultError` at the same time.

9.8.5 Decorators

Limitations

Typing will only work correctly if *our mypy plugin* is used. This happens due to *mypy issue*.

impure

We also have this handy decorator to help you with the existing impure things in Python:

```
from returns.io import impure

name: IO[str] = impure(input)('What is your name?')
```

You can also decorate your own functions with `@impure` for better readability and clearness:

```
import random
from returns.io import impure

@impure
def get_user() -> 'User':
    return random.randint(1, 5)
```

impure_safe

Similar to `impure` and `safe` decorators. Once applied, it transforms the return type to be `IOResultE`:

```

from returns.io import IOResultE, impure_safe

@impure_safe
def http_get(path: str) -> 'Response':
    return requests.get(path)

container: IOResultE['Response'] = http_get('/home')

```

Use for impure operations that might fail.

9.8.6 Helpers

Don't forget to check out *Converters*.

unsafe_perform_io

Sometimes you really need to get the raw value from IO container. For example:

```

def index_view(request, user_id):
    user: IO[User] = get_user(user_id)
    return render('index.html', {'user': user}) # ???

```

In this case your web-framework will not render your user correctly. Since it does not expect it to be wrapped inside IO containers. And we obviously cannot map or bind this function.

What to do? Use *unsafe_perform_io*:

```

from returns.unsafe import unsafe_perform_io

def index_view(request, user_id):
    user: IO[User] = get_user(user_id)
    return render('index.html', {'user': unsafe_perform_io(user)}) # Ok

```

We need it as an escape and compatibility mechanism for our imperative shell.

In other words:

```

>>> from returns.unsafe import unsafe_perform_io
>>> from returns.io import IO

>>> assert unsafe_perform_io(IO('abc')) == 'abc'

```

It is recommended to use `import-linter` to restrict imports from `returns.unsafe` expect the top-level modules.

Inspired by Haskell's `unsafePerformIO`

9.8.7 FAQ

Why aren't IO lazy?

Please, note that our IO implementation is not lazy by design. This way when you mark something as `@impure` it will work as previously. The only thing that changes is the return type.

Instead we offer to use `unsafe_perform_io` to work with IO and simulate laziness.

But, you can always make your IO lazy:

```
>>> from returns.io import IO
>>> lazy = lambda: IO(1)
>>> assert lazy() == IO(1)
```

We have decided that it would be better and more familiar for Python devs.

What is the difference between IO[T] and T?

What kind of input parameter should my function accept IO[T] or simple T?

It really depends on your domain / context. If the value is pure, than use raw unwrapped values. If the value is fetched, input, received, selected, than use IO or IOResult container: first one for operations that never fail, second one for operations that might fail.

Most web applications are just fully covered with IO.

Why can't we use IO[Result] instead of IOResult?

We actually can! But, it is harder to write. And IOResult is actually the very same thing as IO[Result], but has nicer API:

```
x: IO[Result[int, str]]
x.map(lambda io: io.map(lambda number: number + 1))

# Is the same as:

y: IOResult[int, str]
y.map(lambda number: number + 1)
```

The second one looks better, doesn't it?

How to create unit objects for IOResult?

TLDR: you need to use `IOSuccess` and `IOFailure` functions or `IOResult.from_value` and `IOResult.from_failure` methods:

```
>>> from returns.io import IOResult, IOSuccess, IOFailure

>>> first: IOResult[int, str] = IOSuccess(1)
>>> second: IOResult[float, int] = IOFailure(1)

>>> assert IOResult.from_value(1) == IOSuccess(1)
>>> assert IOResult.from_failure(2) == IOFailure(2)
```

You can also annotate your variables properly. Otherwise, `mypy` will treat `IOSuccess(1)` as `IOSuccess[int, Any]`. You can narrow the type in advance.

See *How to create unit objects?* for more details.

9.8.8 Further reading

- Functional core, imperative shell
- Functional architecture is Ports and Adapters
- IO effect in Scala
- Getting started with fp-ts: IO
- IOEither
- Effect Tracking Is Commercially Worthless

9.8.9 API Reference

class `IO`(*inner_value*)

Bases: `BaseContainer`, `SupportsKindN[IO, _ValueType, NoReturn, NoReturn]`, `IOLikeN[_ValueType, NoReturn, NoReturn]`

Explicit container for impure function results.

We also sometimes call it “marker” since once it is marked, it cannot be ever unmarked. There’s no way to directly get its internal value.

Note that `IO` represents a computation that never fails.

Examples of such computations are:

- read / write to `localStorage`
- get the current time
- write to the console
- get a random number

Use `IOResult[...]` for operations that might fail. Like DB access or network operations.

See also:

- <https://dev.to/gcanti/getting-started-with-fp-ts-io-36p6>
- <https://gist.github.com/chris-taylor/4745921>

Parameters

inner_value (`TypeVar[_ValueType, covariant=True]`) –

equals(*other*)

Typesafe equality comparison with other *Result* objects.

Parameters

- **self** (`KindN[TypeVar[_EqualType, bound=Equable], Any, Any, Any]`) –
- **other** (`KindN[TypeVar[_EqualType, bound=Equable], Any, Any, Any]`) –

Return type

bool

map(*function*)

Applies function to the inner value.

Applies ‘function’ to the contents of the IO instance and returns a new IO object containing the result. ‘function’ should accept a single “normal” (non-container) argument and return a non-container result.

```
>>> def mappable(string: str) -> str:
...     return string + 'b'

>>> assert IO('a').map(mappable) == IO('ab')
```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)]], covariant=True) –

Return type

IO[TypeVar(_NewValueType)]

apply(*container*)

Calls a wrapped function in a container on this container.

```
>>> from returns.io import IO
>>> assert IO('a').apply(IO(lambda inner: inner + 'b')) == IO('ab')
```

Or more complex example that shows how we can work with regular functions and multiple IO arguments:

```
>>> from returns.curry import curry

>>> @curry
... def applicable(first: str, second: str) -> str:
...     return first + second

>>> assert IO('b').apply(IO('a').apply(IO(applicable))) == IO('ab')
```

Parameters

container (*KindN*[*IO*, Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)], Any, Any]] – covariant=True),

Return type

IO[TypeVar(_NewValueType)]

bind(*function*)

Applies ‘function’ to the result of a previous calculation.

‘function’ should accept a single “normal” (non-container) argument and return IO type object.

```
>>> def bindable(string: str) -> IO[str]:
...     return IO(string + 'b')

>>> assert IO('a').bind(bindable) == IO('ab')
```

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)], *KindN*[IO, TypeVar(_NewValueType), Any, Any]) –

Return type

IO[TypeVar(_NewValueType)]

bind_io(*function*)

Alias for *bind* method. Part of the *IOLikeN* interface.

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)], *KindN*[IO, TypeVar(_NewValueType), Any, Any]) –

Return type

IO[TypeVar(_NewValueType)]

classmethod do(*expr*)

Allows working with unwrapped values of containers in a safe way.

```
>>> from returns.io import IO
>>> assert IO.do(
...     first + second
...     for first in IO(2)
...     for second in IO(3)
... ) == IO(5)
```

See *Do Notation* to learn more.

Parameters

expr (Generator[TypeVar(_NewValueType), None, None]) –

Return type

IO[TypeVar(_NewValueType)]

classmethod from_value(*inner_value*)

Unit function to construct new IO values.

Is the same as regular constructor:

```
>>> from returns.io import IO
>>> assert IO(1) == IO.from_value(1)
```

Part of the *returns.interfaces.applicative.ApplicativeN* interface.

Parameters

inner_value (TypeVar(_NewValueType)) –

Return type

IO[TypeVar(_NewValueType)]

classmethod from_io(*inner_value*)

Unit function to construct new IO values from existing IO.

```
>>> from returns.io import IO
>>> assert IO(1) == IO.from_io(IO(1))
```

Part of the *returns.interfaces.specific.IO.IOLikeN* interface.

Parameters**inner_value** (*IO*[TypeVar(_NewValueType)]) –**Return type***IO*[TypeVar(_NewValueType)]**classmethod** **from_ioresult**(*inner_value*)Converts *IOResult*[a, b] back to *IO*[*Result*[a, b]].

Can be really helpful for composition.

```
>>> from returns.io import IO, IOSuccess
>>> from returns.result import Success
>>> assert IO.from_ioresult(IOSuccess(1)) == IO(Success(1))
```

Is the reverse of *returns.io.IOResult.from_typecast()*.**Parameters****inner_value** (*IOResult*[TypeVar(_NewValueType), TypeVar(_NewErrorType)]) –**Return type***IO*[*Result*[TypeVar(_NewValueType), TypeVar(_NewErrorType)]]**impure**(*function*)Decorator to mark function that it returns *IO* container.If you need to mark async function as impure, use *returns.future.future()* instead. This decorator only works with sync functions. Example:

```
>>> from returns.io import IO, impure

>>> @impure
... def function(arg: int) -> int:
...     return arg + 1 # this action is pure, just an example
...

>>> assert function(1) == IO(2)
```

Parameters**function** (Callable[[ParamSpec(_FuncParams)], TypeVar(_NewValueType)]) –**Return type**Callable[[ParamSpec(_FuncParams)], *IO*[TypeVar(_NewValueType)]]**class** **IOResult**(*inner_value*)Bases: *BaseContainer*, *SupportsKindN*[*IOResult*, _ValueType, _ErrorType, NoReturn], *IOResultBasedN*[_ValueType, _ErrorType, NoReturn]

Explicit container for impure function results that might fail.

Definition

This type is similar to `returns.result.Result`. This basically a more useful version of `IO[Result[a, b]]`. Use this type for IO computations that might fail. Examples of IO computations that might fail are:

- access database
- access network
- access filesystem

Use `IO` for operations that do IO but do not fail.

Note, that even methods like `unwrap`()` and `value_or()` return values wrapped in `IO`.

`IOResult` is a complex compound value that consists of:

- raw value
- `Result`
- `IO`

This is why it has so many helper and factory methods:

- You can construct `IOResult` from raw values with `IOSuccess()` and `IOFailure()` public type constructors
- You can construct `IOResult` from `IO` values with `from_failed_io()` and `IOResult.from_io()`
- You can construct `IOResult` from `Result` values with `from_result()`

We also have a lot of utility methods for better function composition like:

- `bind_result()` to work with functions which return `Result`
- `from_typecast()` to work with `IO[Result[...]]` values

See also:

<https://github.com/gcanti/fp-ts/blob/master/docs/modules/IOEither.ts.md>

Implementation

This class contains all the methods that can be delegated to `Result`. But, some methods are not implemented which means that we have to use special `_IOSuccess` and `_IOFailure` implementation details to correctly handle these callbacks.

Do not rely on them! Use public functions and types instead.

Parameters

inner_value (`Result`[`TypeVar`(`_ValueType`, `covariant=True`), `TypeVar`(`_ErrorType`, `covariant=True`)])

success_type

Success type that is used to represent the successful computation.

Parameters

inner_value (`TypeVar`(`_ValueType`, `covariant=True`))

alias of `IOSuccess`

failure_type

Failure type that is used to represent the failed computation.

Parameters

inner_value (TypeVar(_ErrorType, covariant=True)) –

alias of *IOFailure*

equals(*other*)

Typesafe equality comparison with other *IOResult* objects.

Parameters

- **self** (*KindN*[TypeVar(_EqualType, bound= *Equable*), Any, Any, Any]) –
- **other** (*KindN*[TypeVar(_EqualType, bound= *Equable*), Any, Any, Any]) –

Return type

bool

property trace: Optional[List[FrameInfo]]

Returns a stack trace when *IOFailure()* was called.

swap()

Swaps value and error types.

So, values become errors and errors become values. It is useful when you have to work with errors a lot. And since we have a lot of `.bind_` related methods and only a single `.lash` - it is easier to work with values.

```
>>> from returns.io import IOSuccess, IOFailure
>>> assert IOSuccess(1).swap() == IOFailure(1)
>>> assert IOFailure(1).swap() == IOSuccess(1)
```

Return type

IOResult[TypeVar(_ErrorType, covariant=True), TypeVar(_ValueType, covariant=True)]

map(*function*)

Composes successful container with a pure function.

```
>>> from returns.io import IOSuccess
>>> assert IOSuccess(1).map(lambda num: num + 1) == IOSuccess(2)
```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], TypeVar(_NewValueType)]) –

Return type

IOResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

apply(*container*)

Calls a wrapped function in a container on this container.

```
>>> from returns.io import IOSuccess, IOFailure
>>> def applicable(first: str) -> str:
```

(continues on next page)

(continued from previous page)

```

...     return first + 'b'
>>> assert IOSuccess('a').apply(
...     IOSuccess(appliable),
... ) == IOSuccess('ab')
>>> assert IOFailure('a').apply(
...     IOSuccess(appliable),
... ) == IOFailure('a')

>>> assert IOSuccess('a').apply(IOFailure(1)) == IOFailure(1)
>>> assert IOFailure('a').apply(IOFailure('b')) == IOFailure('a')

```

Parameters

container (*KindN*[*IOResult*, Callable[[TypeVar(_ValueType, covariant=True)], TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), Any]] –

Return type

IOResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind(*function*)

Composes successful container with a function that returns a container.

```

>>> from returns.io import IOResult, IOFailure, IOSuccess
>>> def bindable(string: str) -> IOResult[str, str]:
...     if len(string) > 1:
...         return IOSuccess(string + 'b')
...         return IOFailure(string + 'c')

>>> assert IOSuccess('aa').bind(bindable) == IOSuccess('aab')
>>> assert IOSuccess('a').bind(bindable) == IOFailure('ac')
>>> assert IOFailure('a').bind(bindable) == IOFailure('a')

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], *KindN*[*IOResult*, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), Any]]) –

Return type

IOResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_ioresult(*function*)

Alias for *bind_ioresult* method. Part of the *IOResultBasedN* interface.

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], *KindN*[*IOResult*, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), Any]]) –

Return type

IOResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_result(*function*)

Composes successful container with a function that returns a container.

Similar to *bind()*, but works with containers that return *returns.result.Result* instead of *IOResult*.

```

>>> from returns.io import IOFailure, IOSuccess
>>> from returns.result import Result, Success

>>> def bindable(string: str) -> Result[str, str]:
...     if len(string) > 1:
...         return Success(string + 'b')
...     return Failure(string + 'c')

>>> assert IOSuccess('aa').bind_result(bindable) == IOSuccess('aab')
>>> assert IOSuccess('a').bind_result(bindable) == IOFailure('ac')
>>> assert IOFailure('a').bind_result(bindable) == IOFailure('a')

```

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)],
Result[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]) –

Return type

IOResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_io(function)

Composes successful container with a function that returns a container.

Similar to *bind()*, but works with containers that return *returns.io.IO* instead of *IOResult*.

```

>>> from returns.io import IO, IOFailure, IOSuccess

>>> def bindable(string: str) -> IO[str]:
...     return IO(string + 'z')

>>> assert IOSuccess('a').bind_io(bindable) == IOSuccess('az')
>>> assert IOFailure('a').bind_io(bindable) == IOFailure('a')

```

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)],
IO[TypeVar(_NewValueType)]) –

Return type

IOResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

alt(function)

Composes failed container with a pure function to modify failure.

```

>>> from returns.io import IOFailure
>>> assert IOFailure(1).alt(float) == IOFailure(1.0)

```

Parameters

function (Callable[[TypeVar(_ErrorType), covariant=True)],
TypeVar(_NewErrorType)) –

Return type

IOResult[TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType)]

lash(function)

Composes failed container with a function that returns a container.

```
>>> from returns.io import IOFailure, IOSuccess, IOResult
>>> def lashable(state: str) -> IOResult[int, str]:
...     if len(state) > 1:
...         return IOSuccess(len(state))
...     return IOFailure('oops')

>>> assert IOFailure('a').lash(lashable) == IOFailure('oops')
>>> assert IOFailure('abc').lash(lashable) == IOSuccess(3)
>>> assert IOSuccess('a').lash(lashable) == IOSuccess('a')
```

Parameters

function (Callable[[TypeVar(_ErrorType, covariant=True)], KindN[IOResult, TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType), Any]]) –

Return type

IOResult[TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType)]

value_or(default_value)

Get value from successful container or default value from failed one.

```
>>> from returns.io import IO, IOFailure, IOSuccess
>>> assert IOSuccess(1).value_or(None) == IO(1)
>>> assert IOFailure(1).value_or(None) == IO(None)
```

Parameters

default_value (TypeVar(_NewValueType)) –

Return type

IO[Union[TypeVar(_ValueType, covariant=True), TypeVar(_NewValueType)]]

unwrap()

Get value from successful container or raise exception for failed one.

```
>>> from returns.io import IO, IOFailure, IOSuccess
>>> assert IOSuccess(1).unwrap() == IO(1)

>>> IOFailure(1).unwrap()
Traceback (most recent call last):
...
returns.primitives.exceptions.UnwrapFailedError
```

Return type

IO[TypeVar(_ValueType, covariant=True)]

failure()

Get failed value from failed container or raise exception from success.

```
>>> from returns.io import IO, IOFailure, IOSuccess
>>> assert IOFailure(1).failure() == IO(1)
```

(continues on next page)

(continued from previous page)

```
>>> IOSuccess(1).failure()
Traceback (most recent call last):
...
returns.primitives.exceptions.UnwrapFailedError
```

Return type`IO[TypeVar(_ErrorType, covariant=True)]`**compose_result**(*function*)Composes inner `Result` with `IOResult` returning function.

Can be useful when you need an access to both states of the result.

```
>>> from returns.io import IOResult, IOSuccess, IOFailure
>>> from returns.result import Result

>>> def count(container: Result[int, int]) -> IOResult[int, int]:
...     return IOResult.from_result(
...         container.map(lambda x: x + 1).alt(abs),
...     )

>>> assert IOSuccess(1).compose_result(count) == IOSuccess(2)
>>> assert IOFailure(-1).compose_result(count) == IOFailure(1)
```

Parameters

function (Callable[[`Result`[`TypeVar`(`_ValueType`, `covariant=True`), `TypeVar`(`_ErrorType`, `covariant=True`)]], `KindN`[`IOResult`, `TypeVar`(`_NewValueType`), `TypeVar`(`_ErrorType`, `covariant=True`), `Any`]]) –

Return type`IOResult`[`TypeVar`(`_NewValueType`), `TypeVar`(`_ErrorType`, `covariant=True`)]**classmethod** `do`(*expr*)

Allows working with unwrapped values of containers in a safe way.

```
>>> from returns.io import IOResult, IOFailure, IOSuccess

>>> assert IOResult.do(
...     first + second
...     for first in IOSuccess(2)
...     for second in IOSuccess(3)
... ) == IOSuccess(5)

>>> assert IOResult.do(
...     first + second
...     for first in IOFailure('a')
...     for second in IOSuccess(3)
... ) == IOFailure('a')
```

See *Do Notation* to learn more. This feature requires our *mypy plugin*.

Parameters**expr** (Generator[TypeVar(_NewValueType), None, None]) –**Return type**`IOResult`[TypeVar(_NewValueType), TypeVar(_NewErrorType)]**classmethod** `from_typecast`(*inner_value*)Converts `IO[Result[_ValueType, _ErrorType]]` to `IOResult`.Also prevails the type of `Result` to `IOResult`, so: `IO[Result[_ValueType, _ErrorType]]` would become `IOResult[_ValueType, _ErrorType]`.

```

>>> from returns.result import Success
>>> from returns.io import IO, IOResult, IOSuccess
>>> container = IO(Success(1))
>>> assert IOResult.from_typecast(container) == IOSuccess(1)

```

Can be reverted via `returns.io.IO.from_ioresult()` method.**Parameters****inner_value** (`IO[Result`[TypeVar(_NewValueType), TypeVar(_NewErrorType)]) –**Return type**`IOResult`[TypeVar(_NewValueType), TypeVar(_NewErrorType)]**classmethod** `from_failed_io`(*inner_value*)Creates new `IOResult` from “failed” `IO` container.

```

>>> from returns.io import IO, IOResult, IOFailure
>>> container = IO(1)
>>> assert IOResult.from_failed_io(container) == IOFailure(1)

```

Parameters**inner_value** (`IO`[TypeVar(_NewErrorType)]) –**Return type**`IOResult`[Any, TypeVar(_NewErrorType)]**classmethod** `from_io`(*inner_value*)Creates new `IOResult` from “successful” `IO` container.

```

>>> from returns.io import IO, IOResult, IOSuccess
>>> container = IO(1)
>>> assert IOResult.from_io(container) == IOSuccess(1)

```

Parameters**inner_value** (`IO`[TypeVar(_NewValueType)]) –**Return type**`IOResult`[TypeVar(_NewValueType), Any]**classmethod** `from_result`(*inner_value*)Creates `IOResult` from `Result` value.

```
>>> from returns.io import IOResult, IOSuccess, IOFailure
>>> from returns.result import Success, Failure

>>> assert IOResult.from_result(Success(1)) == IOSuccess(1)
>>> assert IOResult.from_result(Failure(2)) == IOFailure(2)
```

Parameters

inner_value (*Result*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)])

Return type

IOResult[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)]

classmethod from_ioresult(*inner_value*)

Creates *IOResult* from existing *IOResult* value.

```
>>> from returns.io import IOResult, IOSuccess, IOFailure

>>> assert IOResult.from_ioresult(IOSuccess(1)) == IOSuccess(1)
>>> assert IOResult.from_ioresult(IOFailure(2)) == IOFailure(2)
```

Parameters

inner_value (*IOResult*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)])

Return type

IOResult[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)]

classmethod from_value(*inner_value*)

One more value to create success unit values.

It is useful as a united way to create a new value from any container.

```
>>> from returns.io import IOResult, IOSuccess
>>> assert IOResult.from_value(1) == IOSuccess(1)
```

You can use this method or *IOSuccess()*, choose the most convenient for you.

Parameters

inner_value (*TypeVar*(*_NewValueType*))

Return type

IOResult[*TypeVar*(*_NewValueType*), *Any*]

classmethod from_failure(*inner_value*)

One more value to create failure unit values.

It is useful as a united way to create a new value from any container.

```
>>> from returns.io import IOResult, IOFailure
>>> assert IOResult.from_failure(1) == IOFailure(1)
```

You can use this method or *IOFailure()*, choose the most convenient for you.

Parameters

inner_value (*TypeVar*(*_NewErrorType*))

Return type

IOResult[*Any*, *TypeVar*(*_NewErrorType*)]

final class IOFailure(*inner_value*)

Bases: `IOResult`[Any, _ErrorType]

IOFailure representation.

Parameters

inner_value (TypeVar(_ErrorType, covariant=True)) –

bind(*function*)

Does nothing for IOFailure.

bind_ioresult(*function*)

Alias for `bind_ioresult` method. Part of the `IOResultBasedN` interface. # noqa: E501

bind_result(*function*)

Does nothing for IOFailure.

bind_io(*function*)

Does nothing for IOFailure.

lash(*function*)

Composes this container with a function returning `IOResult`.

final class IOSuccess(*inner_value*)

Bases: `IOResult`[_ValueType, Any]

IOSuccess representation.

Parameters

inner_value (TypeVar(_ValueType, covariant=True)) –

bind(*function*)

Composes this container with a function returning `IOResult`.

bind_ioresult(*function*)

Alias for `bind_ioresult` method. Part of the `IOResultBasedN` interface. # noqa: E501

bind_result(*function*)

Binds `Result` returning function to current container.

bind_io(*function*)

Binds IO returning function to current container.

lash(*function*)

Does nothing for IOSuccess.

IOResultE

Alias for a popular case when `IOResult` has `Exception` as error type.

alias of `IOResult`[_ValueType, `Exception`]

impure_safe(*function*)

Decorator to mark function that it returns `IOResult` container.

Should be used with care, since it only catches `Exception` subclasses. It does not catch `BaseException` subclasses.

If you need to mark `async` function as impure, use `returns.future.future_safe()` instead. This decorator only works with sync functions. Example:

```

>>> from returns.io import IOSuccess, impure_safe

>>> @impure_safe
... def function(arg: int) -> float:
...     return 1 / arg
...

>>> assert function(1) == IOSuccess(1.0)
>>> assert function(0).failure()

```

Similar to `returns.future.future_safe()` and `returns.result.safe()` decorators.

Parameters

`function` (Callable[[ParamSpec(_FuncParams)], TypeVar(_NewValueType)]) –

Return type

Callable[[ParamSpec(_FuncParams)], `IOResult`[TypeVar(_NewValueType), Exception]]

unsafe_perform_io(*wrapped_in_io*)

Compatibility utility and escape mechanism from IO world.

Just unwraps the internal value from `returns.io.IO` container. Should be used with caution! Since it might be overused by lazy and ignorant developers.

It is recommended to have only one place (module / file) in your program where you allow unsafe operations.

We recommend to use `import-linter` to enforce this rule.

```

>>> from returns.io import IO
>>> assert unsafe_perform_io(IO(1)) == 1

```

See also:

- <https://github.com/seddonym/import-linter>

Parameters

`wrapped_in_io` (`IO`[TypeVar(_ValueType)]) –

Return type

TypeVar(_ValueType)

9.9 Future

A set of primitives to work with `async` functions.

Can be used with `asyncio`, `trio`, and `curio`. And any event-loop!

Tested with `anyio`.

What problems do we solve with these containers? Basically these ones:

1. You cannot call `async` function from a `sync` one
2. Any unexpectedly thrown exception can ruin your whole event loop
3. Ugly composition with lots of `await` statements

9.9.1 Future container

Without `Future` container it is impossible to compose two functions: sync and async one.

You simply cannot `await` coroutines inside a sync context. It is even a `SyntaxError`.

```
def test():
    await some()
# SyntaxError: 'await' outside async function
```

So, you have to turn you function into async one. And all callers of this function in async functions. And all their callers.

This is really hard to model. When you code has two types of uncomposable functions you increase your mental complexity by extreme levels.

Instead, you can use `Future` container, it allows you to model async interactions in a sync manner:

```
>>> from returns.future import Future

>>> async def first() -> int:
...     return 1

>>> async def second(arg: int) -> int:
...     return arg + 1

>>> def main() -> Future[int]: # sync function!
...     return Future(first()).bind_awaitable(second)
```

Now we can compose async functions and maintaining a sync context! It is also possible run a `Future` with regular tools like `anyio.run` or `anyio.run`:

```
>>> import anyio
>>> from returns.io import IO

>>> assert anyio.run(main().awaitable) == IO(2)
```

One more very useful thing `Future` does behind the scenes is converting its result to IO-based containers. This helps a lot when separating pure and impure (async functions are impure) code inside your app.

9.9.2 FutureResult

This container becomes very useful when working with async function that can fail.

It works the similar way regular `Result` does. And is literally a wrapper around `Future[Result[_V, _E]]` type.

Let's see how it can be used in a real program:

```
1 import asyncio # we use `asyncio` only as an example, you can use any io lib
2 from typing import Final, Sequence, cast
3
4 import httpx # you would need to `pip install httpx`
5 from typing_extensions import TypedDict
6
7 from returns.future import FutureResultE, future_safe
8 from returns.io import IOResultE
```

(continues on next page)

```
9  from returns.iterables import Fold
10
11  _URL: Final = 'https://jsonplaceholder.typicode.com/posts/{0}'
12
13
14  class _Post(TypedDict):
15      id: int
16      user_id: int
17      title: str
18      body: str
19
20
21  @future_safe
22  async def _fetch_post(post_id: int) -> _Post:
23      # Ideally, we can use `ReaderFutureResult` to provide `client` from deps.
24      async with httpx.AsyncClient(timeout=5) as client:
25          response = await client.get(_URL.format(post_id))
26          response.raise_for_status()
27          return cast(_Post, response.json()) # or validate the response
28
29
30  def _show_titles(number_of_posts: int) -> Sequence[FutureResultE[str]]:
31      def factory(post: _Post) -> str:
32          return post['title']
33
34      return [
35          # Notice how easily we compose async and sync functions:
36          _fetch_post(post_id).map(factory)
37          # TODO: try `for post_id in (2, 1, 0):` to see how async errors work
38          for post_id in range(1, number_of_posts + 1)
39      ]
40
41
42  async def main() -> IOResultE[Sequence[str]]:
43      """
44      Main entrypoint for the async world.
45
46      Let's fetch 3 titles of posts asynchronously.
47      We use `gather` to run requests in "parallel".
48      """
49      futures: Sequence[IOResultE[str]] = await asyncio.gather(*_show_titles(3))
50      return Fold.collect(futures, IOResultE.from_value(()))
51
52
53  if __name__ == '__main__':
54      print(asyncio.run(main())) # noqa: WPS421
55      # <IOResult: <Success: (
56      #   'sunt aut facere repellat provident occaecati ...',
57      #   'qui est esse',
58      #   'ea molestias quasi exercitationem repellat qui ipsa sit aut',
59      # )>>
```

What is different?

1. We can now easily make `show_titles` sync, we can also make `_fetch_post` sync, but we would need to use `ReaderFutureResult` container with proper dependencies in this case
2. We now don't care about errors at all. In this example any error will cancel the whole pipeline
3. We now have `.map` method to easily compose sync and async functions

You can see the next example with `RequiresContextFutureResult` and without a single `async/await`. That example illustrates the whole point of our actions: writing sync code that executes asynchronously without any magic at all.

9.9.3 Aliases

There are several useful aliases for `FutureResult` type with some common values:

- `returns.future.FutureResultE` is an alias for `FutureResult[... Exception]`, just use it when you want to work with `FutureResult` containers that use exceptions as error type. It is named `FutureResultE` because it is `FutureResultException` and `FutureResultError` at the same time.

9.9.4 Decorators

future

This decorator helps to easily transform `async def` into `Future`:

```
>>> import anyio
>>> from returns.future import future, Future
>>> from returns.io import IO

>>> @future
... async def test(arg: int) -> float:
...     return arg / 2

>>> future_instance = test(1)
>>> assert isinstance(future_instance, Future)
>>> assert anyio.run(future_instance.awaitable) == IO(0.5)
```

Make sure that you decorate with `@future` only coroutines that do not throw exceptions. For ones that do, use `future_safe`.

future_safe

This decorator converts `async def` into `FutureResult`, which means that it becomes:

1. Full featured `Future` like container
2. Safe from any exceptions

Let's dig into it:

```
>>> import anyio
>>> from returns.future import future_safe, FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> @future_safe
```

(continues on next page)

```
... async def test(arg: int) -> float:
...     return 1 / arg

>>> future_instance = test(2)
>>> assert isinstance(future_instance, FutureResult)
>>> assert anyio.run(future_instance.awaitable) == IOSuccess(0.5)

>>> str(anyio.run(test(0).awaitable)) # this will fail
'<IOResult: <Failure: division by zero>>'
```

Never miss exceptions ever again!

asyncify

Helper decorator to transform regular sync function into async ones.

```
>>> import anyio
>>> from inspect import iscoroutinefunction
>>> from returns.future import asyncify

>>> @asyncify
... def your_function(x: int) -> int:
...     return x + 1

>>> assert iscoroutinefunction(your_function) is True
>>> assert anyio.run(your_function, 1) == 2
```

Very important note: making your function async does not mean it will work asynchronously. It can still block if it uses blocking calls. Here's an example of how you **must not** do:

```
import requests
from returns.future import asyncify

@asyncify
def please_do_not_do_that():
    return requests.get('...') # this will still block!
```

Do not overuse this decorator. It is only useful for some basic composition with `Future` and `FutureResult`.

9.9.5 FAQ

Is it somehow related to Future object from asyncio?

Nope, we just use the same naming there are in other languages and platforms. Python happens to have its own meaning for this word.

In our worldview, these two `Future` entities should never meet each other in a single codebase.

It is also not related to `concurrent.Future`.

How to create unit objects?

For `Future` container:

- `from_value` when you have a raw value
- `from_io` when you have existing IO container
- `from_future_result` when you have existing `FutureResult`

For `FutureResult` container:

- `from_value` when you want to mark some raw value as a `Success`
- `from_failure` when you want to mark some raw value as a `Failure`
- `from_result` when you already have `Result` container
- `from_io` when you have successful IO
- `from_failed_io` when you have failed IO
- `from_future` when you have successful `Future`
- `from_failed_future` when you have failed `Future`
- `from_typecast` when you have existing `Future[Result]`

What is the difference between `Future[Result[a, b]]` and `FutureResult[a, b]`?

There's almost none.

The only difference is that `FutureResult[a, b]` is a handy wrapper around `Future[Result[a, b]]`, so you won't need to use methods like `.map` and `.bind` twice.

You can always convert it with methods like `.from_typecast` and `.from_future_result`.

9.9.6 Further reading

- [How Async Should Have Been](#)
- [What Color is Your Function?](#)
- [From Promises to Futures](#)

9.9.7 API Reference

async `async_identity(instance)`

Async function that returns its argument.

```
>>> import anyio
>>> from returns.future import async_identity
>>> assert anyio.run(async_identity, 1) == 1
```

See `returns.functions.identity()` for sync version of this function and more docs and examples.

Parameters

instance (`TypeVar(_FirstType)`) –

Return type

TypeVar(_FirstType)

final class Future(*inner_value*)Bases: *BaseContainer*, *SupportsKindN[Future, _ValueType, NoReturn, NoReturn]*, *FutureBasedN[_ValueType, NoReturn, NoReturn]*

Container to easily compose async functions.

Represents a better abstraction over a simple coroutine.

Is framework, event-loop, and IO-library agnostics. Works with `asyncio`, `curio`, `trio`, or any other tool. Internally we use `anyio` to test that it works as expected for any io stack.Note that `Future[a]` represents a computation that never fails and returns `IO[a]` type. Use `FutureResult[a, b]` for operations that might fail. Like DB access or network operations.Is not related to `asyncio.Future` in any kind.**Tradeoffs**

Due to possible performance issues we move all coroutines definitions to a separate module.

See also:

- <https://gcanti.github.io/fp-ts/modules/Task.ts.html>
- https://zio.dev/docs/overview/overview_basic_concurrency

Parameters**inner_value** (Awaitable[TypeVar(_ValueType, covariant=True)]) –**async awaitable()**Transforms `Future[a]` to `Awaitable[IO[a]]`.Use this method when you need a real coroutine. Like for `asyncio.run` calls.Note, that returned value will be wrapped in `returns.io.IO` container.

```
>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO
>>> assert anyio.run(Future.from_value(1).awaitable) == IO(1)
```

Return type`IO[TypeVar(_ValueType, covariant=True)]`**map**(*function*)

Applies function to the inner value.

Applies ‘function’ to the contents of the IO instance and returns a new `Future` object containing the result. ‘function’ should accept a single “normal” (non-container) argument and return a non-container result.

```
>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO
```

(continues on next page)

(continued from previous page)

```

>>> def mappable(x: int) -> int:
...     return x + 1

>>> assert anyio.run(
...     Future.from_value(1).map(mappable).awaitable,
... ) == IO(2)

```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)]], covariant=True) –

Return type

Future[TypeVar(_NewValueType)]

apply(container)

Calls a wrapped function in a container on this container.

```

>>> import anyio
>>> from returns.future import Future

>>> def transform(arg: int) -> str:
...     return str(arg) + 'b'

>>> assert anyio.run(
...     Future.from_value(1).apply(
...         Future.from_value(transform),
...     ).awaitable,
... ) == IO('1b')

```

Parameters

container (*KindN*[*Future*, Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)], Any, Any]] – covariant=True) –

Return type

Future[TypeVar(_NewValueType)]

bind(function)

Applies ‘function’ to the result of a previous calculation.

‘function’ should accept a single “normal” (non-container) argument and return Future type object.

```

>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO

>>> def bindable(x: int) -> Future[int]:
...     return Future.from_value(x + 1)

>>> assert anyio.run(
...     Future.from_value(1).bind(bindable).awaitable,
... ) == IO(2)

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], KindN[Future, TypeVar(_NewValueType), Any, Any]]) –

Return type

Future[TypeVar(_NewValueType)]

bind_future(*function*)

Alias for *bind* method. Part of the *FutureBasedN* interface.

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], KindN[Future, TypeVar(_NewValueType), Any, Any]]) –

Return type

Future[TypeVar(_NewValueType)]

bind_async(*function*)

Compose a container and async function returning a container.

This function should return a container value. See *bind_awaitable*() to bind async function that returns a plain value.

```
>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO

>>> async def coroutine(x: int) -> Future[str]:
...     return Future.from_value(str(x + 1))

>>> assert anyio.run(
...     Future.from_value(1).bind_async(coroutine).awaitable,
... ) == IO('2')
```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], Awaitable[KindN[Future, TypeVar(_NewValueType), Any, Any]]]) –

Return type

Future[TypeVar(_NewValueType)]

bind_async_future(*function*)

Alias for *bind_async* method. Part of the *FutureBasedN* interface.

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], Awaitable[KindN[Future, TypeVar(_NewValueType), Any, Any]]]) –

Return type

Future[TypeVar(_NewValueType)]

bind_awaitable(*function*)

Allows to compose a container and a regular async function.

This function should return plain, non-container value. See *bind_async*() to bind async function that returns a container.

```

>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO

>>> async def coroutine(x: int) -> int:
...     return x + 1

>>> assert anyio.run(
...     Future.from_value(1).bind_awaitable(coroutine).awaitable,
... ) == IO(2)

```

Parameters

function (Callable[[TypeVar(_ValueType), Awaitable[TypeVar(_NewValueType)]]] – covariant=True),

Return type

Future[TypeVar(_NewValueType)]

bind_io(*function*)

Applies ‘function’ to the result of a previous calculation.

‘function’ should accept a single “normal” (non-container) argument and return IO type object.

```

>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO

>>> def bindable(x: int) -> IO[int]:
...     return IO(x + 1)

>>> assert anyio.run(
...     Future.from_value(1).bind_io(bindable).awaitable,
... ) == IO(2)

```

Parameters

function (Callable[[TypeVar(_ValueType), IO[TypeVar(_NewValueType)]]] – covariant=True),

Return type

Future[TypeVar(_NewValueType)]

classmethod do(*expr*)

Allows working with unwrapped values of containers in a safe way.

```

>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO

>>> async def main() -> bool:
...     return await Future.do(
...         first + second
...         async for first in Future.from_value(2)
...         async for second in Future.from_value(3)

```

(continues on next page)

(continued from previous page)

```
...     ) == IO(5)
>>> assert anyio.run(main) is True
```

See *Do Notation* to learn more.

Parameters

expr (AsyncGenerator[TypeVar(_NewValueType), None]) –

Return type

Future[TypeVar(_NewValueType)]

classmethod from_value(*inner_value*)

Allows to create a Future from a plain value.

The resulting Future will just return the given value wrapped in *returns.io.IO* container when awaited.

```
>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO

>>> async def main() -> bool:
...     return (await Future.from_value(1)) == IO(1)

>>> assert anyio.run(main) is True
```

Parameters

inner_value (TypeVar(_NewValueType)) –

Return type

Future[TypeVar(_NewValueType)]

classmethod from_future(*inner_value*)

Creates a new Future from the existing one.

```
>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO

>>> future = Future.from_value(1)
>>> assert anyio.run(Future.from_future(future).awaitable) == IO(1)
```

Part of the FutureBasedN interface.

Parameters

inner_value (*Future*[TypeVar(_NewValueType)]) –

Return type

Future[TypeVar(_NewValueType)]

classmethod from_io(*inner_value*)

Allows to create a Future from IO container.

```
>>> import anyio
>>> from returns.future import Future
```

(continues on next page)

(continued from previous page)

```
>>> from returns.io import IO
>>> async def main() -> bool:
...     return (await Future.from_io(IO(1))) == IO(1)
>>> assert anyio.run(main) is True
```

Parameters**inner_value** (*IO*[*TypeVar*(*_NewValueType*)])**Return type***Future*[*TypeVar*(*_NewValueType*)]**classmethod from_future_result**(*inner_value*)Creates *Future*[*Result*[*a*, *b*]] instance from *FutureResult*[*a*, *b*].This method is the inverse of *from_typecast*().

```
>>> import anyio
>>> from returns.future import Future, FutureResult
>>> from returns.io import IO
>>> from returns.result import Success
>>> container = Future.from_future_result(FutureResult.from_value(1))
>>> assert anyio.run(container.awaitable) == IO(Success(1))
```

Parameters**inner_value** (*FutureResult*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)])

–

Return type*Future*[*Result*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)]]**future**(*function*)Decorator to turn a coroutine definition into *Future* container.

```
>>> import anyio
>>> from returns.io import IO
>>> from returns.future import future
>>> @future
... async def test(x: int) -> int:
...     return x + 1
>>> assert anyio.run(test(1).awaitable) == IO(2)
```

Parameters**function** (*Callable*[[*ParamSpec*(*_FuncParams*)], *Coroutine*[*TypeVar*(*_FirstType*), *TypeVar*(*_SecondType*), *TypeVar*(*_ValueType*, *covariant=True*)]])**Return type***Callable*[[*ParamSpec*(*_FuncParams*)], *Future*[*TypeVar*(*_ValueType*, *covariant=True*)]]

asyncify(*function*)

Decorator to turn a common function into an asynchronous function.

This decorator is useful for composition with `Future` and `FutureResult` containers.

Warning: This function will not your sync function **run** like `async` one. It will still be a blocking function that looks like `async` one. We recommend to only use this decorator with functions that do not access network or filesystem. It is only a composition helper, not a transformer.

Usage example:

```
>>> import anyio
>>> from returns.future import asyncify

>>> @asyncify
... def test(x: int) -> int:
...     return x + 1

>>> assert anyio.run(test, 1) == 2
```

Read more about `async` and `sync` functions: <https://journal.stuffwithstuff.com/2015/02/01/what-color-is-your-function/>

Parameters

function (Callable[[ParamSpec(_FuncParams)], TypeVar(_ValueType, covariant=True)]) –

Return type

Callable[[ParamSpec(_FuncParams)], Coroutine[Any, Any, TypeVar(_ValueType, covariant=True)]]

final class FutureResult(*inner_value*)

Bases: `BaseContainer`, `SupportsKindN[FutureResult, _ValueType, _ErrorType, NoReturn]`, `FutureResultBasedN[_ValueType, _ErrorType, NoReturn]`

Container to easily compose `async` functions.

Represents a better abstraction over a simple coroutine.

Is framework, event-loop, and IO-library agnostics. Works with `asyncio`, `curio`, `trio`, or any other tool. Internally we use `anyio` to test that it works as expected for any `io` stack.

Note that `FutureResult[a, b]` represents a computation that can fail and returns `IOResult[a, b]` type. Use `Future[a]` for operations that cannot fail.

This is a `Future` that returns `Result` type. By providing this utility type we make developers' lives easier. `FutureResult` has a lot of composition helpers to turn complex nested operations into a one function calls.

Tradeoffs

Due to possible performance issues we move all coroutines definitions to a separate module.

See also:

- <https://gcanti.github.io/fp-ts/modules/TaskEither.ts.html>
- https://zio.dev/docs/overview/overview_basic_concurrency

Parameters

inner_value (Awaitable[Result[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType, covariant=True)]])

async awaitable()

Transforms FutureResult[a, b] to Awaitable[IOResult[a, b]].

Use this method when you need a real coroutine. Like for `asyncio.run` calls.

Note, that returned value will be wrapped in `returns.io.IOResult` container.

```
>>> import anyio
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess
>>> assert anyio.run(
...     FutureResult.from_value(1).awaitable,
... ) == IOSuccess(1)
```

Return type

`IOResult[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType, covariant=True)]`

swap()

Swaps value and error types.

So, values become errors and errors become values. It is useful when you have to work with errors a lot. And since we have a lot of `.bind_` related methods and only a single `.lash`. It is easier to work with values than with errors.

```
>>> import anyio
>>> from returns.future import FutureSuccess, FutureFailure
>>> from returns.io import IOSuccess, IOFailure

>>> assert anyio.run(FutureSuccess(1).swap) == IOFailure(1)
>>> assert anyio.run(FutureFailure(1).swap) == IOSuccess(1)
```

Return type

`FutureResult[TypeVar(_ErrorType, covariant=True), TypeVar(_ValueType, covariant=True)]`

map(function)

Applies function to the inner value.

Applies ‘function’ to the contents of the IO instance and returns a new `FutureResult` object containing the result. ‘function’ should accept a single “normal” (non-container) argument and return a non-container result.

```

>>> import anyio
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> def mappable(x: int) -> int:
...     return x + 1

>>> assert anyio.run(
...     FutureResult.from_value(1).map(mappable).awaitable,
... ) == IOSuccess(2)
>>> assert anyio.run(
...     FutureResult.from_failure(1).map(mappable).awaitable,
... ) == IOFailure(1)

```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)]] – covariant=True),

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

apply(*container*)

Calls a wrapped function in a container on this container.

```

>>> import anyio
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> def applicable(x: int) -> int:
...     return x + 1

>>> assert anyio.run(
...     FutureResult.from_value(1).apply(
...         FutureResult.from_value(applicable),
...     ).awaitable,
... ) == IOSuccess(2)
>>> assert anyio.run(
...     FutureResult.from_failure(1).apply(
...         FutureResult.from_value(applicable),
...     ).awaitable,
... ) == IOFailure(1)

>>> assert anyio.run(
...     FutureResult.from_value(1).apply(
...         FutureResult.from_failure(2),
...     ).awaitable,
... ) == IOFailure(2)
>>> assert anyio.run(
...     FutureResult.from_failure(1).apply(
...         FutureResult.from_failure(2),
...     ).awaitable,
... ) == IOFailure(1)

```


Parameters

container (*KindN*[*FutureResult*, Callable[[TypeVar(_ValueType, covariant=True)], TypeVar(_NewValueType)], TypeVar(_ErrorType, covariant=True), Any]) –

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind(*function*)

Applies ‘function’ to the result of a previous calculation.

‘function’ should accept a single “normal” (non-container) argument and return Future type object.

```
>>> import anyio
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> def bindable(x: int) -> FutureResult[int, str]:
...     return FutureResult.from_value(x + 1)

>>> assert anyio.run(
...     FutureResult.from_value(1).bind(bindable).awaitable,
... ) == IOSuccess(2)
>>> assert anyio.run(
...     FutureResult.from_failure(1).bind(bindable).awaitable,
... ) == IOFailure(1)
```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], *KindN*[*FutureResult*, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), Any]]) –

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_future_result(*function*)

Alias for *bind* method. Part of the *FutureResultBasedN* interface.

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], *KindN*[*FutureResult*, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), Any]]) –

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_async(*function*)

Composes a container and async function returning container.

This function should return a container value. See *bind_awaitable()* to bind async function that returns a plain value.

```
>>> import anyio
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> async def coroutine(x: int) -> FutureResult[str, int]:
...     return FutureResult.from_value(str(x + 1))
```

(continues on next page)

(continued from previous page)

```

>>> assert anyio.run(
...     FutureResult.from_value(1).bind_async(coroutine).awaitable,
... ) == IOSuccess('2')
>>> assert anyio.run(
...     FutureResult.from_failure(1).bind_async(coroutine).awaitable,
... ) == IOFailure(1)

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], Awaitable[KindN[FutureResult, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), Any]]]) –

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_async_future_result(function)

Alias for *bind_async* method. Part of the *FutureResultBasedN* interface.

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], Awaitable[KindN[FutureResult, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), Any]]]) –

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_awaitable(function)

Allows to compose a container and a regular async function.

This function should return plain, non-container value. See *bind_async()* to bind async function that returns a container.

```

>>> import anyio
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> async def coro(x: int) -> int:
...     return x + 1

>>> assert anyio.run(
...     FutureResult.from_value(1).bind_awaitable(coro).awaitable,
... ) == IOSuccess(2)
>>> assert anyio.run(
...     FutureResult.from_failure(1).bind_awaitable(coro).awaitable,
... ) == IOFailure(1)

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], Awaitable[TypeVar(_NewValueType)]] –

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_result(*function*)

Binds a function returning `Result[a, b]` container.

```
>>> import anyio
>>> from returns.io import IOSuccess, IOFailure
>>> from returns.result import Result, Success
>>> from returns.future import FutureResult

>>> def bind(inner_value: int) -> Result[int, str]:
...     return Success(inner_value + 1)

>>> assert anyio.run(
...     FutureResult.from_value(1).bind_result(bind).awaitable,
... ) == IOSuccess(2)
>>> assert anyio.run(
...     FutureResult.from_failure('a').bind_result(bind).awaitable,
... ) == IOFailure('a')
```

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)], `Result`[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]] –

Return type

`FutureResult`[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_ioresult(*function*)

Binds a function returning `IOResult[a, b]` container.

```
>>> import anyio
>>> from returns.io import IOResult, IOSuccess, IOFailure
>>> from returns.future import FutureResult

>>> def bind(inner_value: int) -> IOResult[int, str]:
...     return IOSuccess(inner_value + 1)

>>> assert anyio.run(
...     FutureResult.from_value(1).bind_ioresult(bind).awaitable,
... ) == IOSuccess(2)
>>> assert anyio.run(
...     FutureResult.from_failure('a').bind_ioresult(bind).awaitable,
... ) == IOFailure('a')
```

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)], `IOResult`[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]] –

Return type

`FutureResult`[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_io(*function*)

Binds a function returning `IO[a]` container.

```

>>> import anyio
>>> from returns.io import IO, IOSuccess, IOFailure
>>> from returns.future import FutureResult

>>> def bind(inner_value: int) -> IO[float]:
...     return IO(inner_value + 0.5)

>>> assert anyio.run(
...     FutureResult.from_value(1).bind_io(bind).awaitable,
... ) == IOSuccess(1.5)
>>> assert anyio.run(
...     FutureResult.from_failure(1).bind_io(bind).awaitable,
... ) == IOFailure(1)

```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)], TypeVar(_ErrorType)], covariant=True) –

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_future(*function*)

Binds a function returning Future[a] container.

```

>>> import anyio
>>> from returns.io import IOSuccess, IOFailure
>>> from returns.future import Future, FutureResult

>>> def bind(inner_value: int) -> Future[float]:
...     return Future.from_value(inner_value + 0.5)

>>> assert anyio.run(
...     FutureResult.from_value(1).bind_future(bind).awaitable,
... ) == IOSuccess(1.5)
>>> assert anyio.run(
...     FutureResult.from_failure(1).bind_future(bind).awaitable,
... ) == IOFailure(1)

```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)], TypeVar(_ErrorType)], covariant=True) –

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

bind_async_future(*function*)

Composes a container and async function returning Future.

Similar to *bind_future()* but works with async functions.

```

>>> import anyio
>>> from returns.future import Future, FutureResult
>>> from returns.io import IOSuccess, IOFailure

```

(continues on next page)

(continued from previous page)

```

>>> async def coroutine(x: int) -> Future[str]:
...     return Future.from_value(str(x + 1))

>>> assert anyio.run(
...     FutureResult.from_value(1).bind_async_future,
...     coroutine,
... ) == IOSuccess('2')
>>> assert anyio.run(
...     FutureResult.from_failure(1).bind_async,
...     coroutine,
... ) == IOFailure(1)

```

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)], Awaitable[Future[TypeVar(_NewValueType)]])

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

alt(function)

Composes failed container with a pure function to modify failure.

```

>>> import anyio
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> def altable(arg: int) -> int:
...     return arg + 1

>>> assert anyio.run(
...     FutureResult.from_value(1).alt(altable).awaitable,
... ) == IOSuccess(1)
>>> assert anyio.run(
...     FutureResult.from_failure(1).alt(altable).awaitable,
... ) == IOFailure(2)

```

Parameters

function (Callable[[TypeVar(_ErrorType), covariant=True)], TypeVar(_NewErrorType))

Return type

FutureResult[TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType)]

lash(function)

Composes failed container with a function that returns a container.

```

>>> import anyio
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess

>>> def lashable(x: int) -> FutureResult[int, str]:

```

(continues on next page)

(continued from previous page)

```

...     return FutureResult.from_value(x + 1)

>>> assert anyio.run(
...     FutureResult.from_value(1).lash(lashable).awaitable,
... ) == IOSuccess(1)
>>> assert anyio.run(
...     FutureResult.from_failure(1).lash(lashable).awaitable,
... ) == IOSuccess(2)

```

Parameters

function (Callable[[TypeVar(_ErrorType, covariant=True)], KindN[FutureResult, TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType), Any]]) –

Return type

FutureResult[TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType)]

compose_result(function)

Composes inner Result with FutureResult returning function.

Can be useful when you need an access to both states of the result.

```

>>> import anyio
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure
>>> from returns.result import Result

>>> def count(container: Result[int, int]) -> FutureResult[int, int]:
...     return FutureResult.from_result(
...         container.map(lambda x: x + 1).alt(abs),
...     )

>>> assert anyio.run(
...     FutureResult.from_value(1).compose_result, count,
... ) == IOSuccess(2)
>>> assert anyio.run(
...     FutureResult.from_failure(-1).compose_result, count,
... ) == IOFailure(1)

```

Parameters

function (Callable[[Result[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType, covariant=True)]], KindN[FutureResult, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), Any]]) –

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]

classmethod do(expr)

Allows working with unwrapped values of containers in a safe way.

```

>>> import anyio
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

```

(continues on next page)

(continued from previous page)

```

>>> async def success() -> bool:
...     return await FutureResult.do(
...         first + second
...         async for first in FutureResult.from_value(2)
...         async for second in FutureResult.from_value(3)
...     ) == IOSuccess(5)

>>> assert anyio.run(success) is True

>>> async def failure() -> bool:
...     return await FutureResult.do(
...         first + second
...         async for first in FutureResult.from_value(2)
...         async for second in FutureResult.from_failure(3)
...     ) == IOFailure(3)

>>> assert anyio.run(failure) is True

```

See *Do Notation* to learn more.

Parameters

expr (AsyncGenerator[TypeVar(_NewValueType), None]) –

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_NewErrorType)]

classmethod from_typecast(*inner_value*)

Creates *FutureResult*[*a*, *b*] from *Future*[*Result*[*a*, *b*]].

```

>>> import anyio
>>> from returns.io import IOSuccess, IOFailure
>>> from returns.result import Success, Failure
>>> from returns.future import Future, FutureResult

>>> async def main():
...     assert await FutureResult.from_typecast(
...         Future.from_value(Success(1)),
...     ) == IOSuccess(1)
...     assert await FutureResult.from_typecast(
...         Future.from_value(Failure(1)),
...     ) == IOFailure(1)

>>> anyio.run(main)

```

Parameters

inner_value (*Future*[*Result*[TypeVar(_NewValueType), TypeVar(_NewErrorType)]] –

Return type

FutureResult[TypeVar(_NewValueType), TypeVar(_NewErrorType)]

classmethod from_future(*inner_value*)

Creates *FutureResult* from successful *Future* value.

```

>>> import anyio
>>> from returns.io import IOSuccess
>>> from returns.future import Future, FutureResult

>>> async def main():
...     assert await FutureResult.from_future(
...         Future.from_value(1),
...         ) == IOSuccess(1)

>>> anyio.run(main)

```

Parameters

inner_value (*Future*[TypeVar(_NewValueType)]) –

Return type

FutureResult[TypeVar(_NewValueType), Any]

classmethod from_failed_future(*inner_value*)

Creates FutureResult from failed Future value.

```

>>> import anyio
>>> from returns.io import IOFailure
>>> from returns.future import Future, FutureResult

>>> async def main():
...     assert await FutureResult.from_failed_future(
...         Future.from_value(1),
...         ) == IOFailure(1)

>>> anyio.run(main)

```

Parameters

inner_value (*Future*[TypeVar(_NewErrorType)]) –

Return type

FutureResult[Any, TypeVar(_NewErrorType)]

classmethod from_future_result(*inner_value*)

Creates new FutureResult from existing one.

```

>>> import anyio
>>> from returns.io import IOSuccess
>>> from returns.future import FutureResult

>>> async def main():
...     assert await FutureResult.from_future_result(
...         FutureResult.from_value(1),
...         ) == IOSuccess(1)

>>> anyio.run(main)

```

Part of the FutureResultLikeN interface.

Parameters**inner_value** (*FutureResult*[TypeVar(_NewValueType), TypeVar(_NewErrorType)])

–

Return type*FutureResult*[TypeVar(_NewValueType), TypeVar(_NewErrorType)]**classmethod** `from_io(inner_value)`Creates *FutureResult* from successful IO value.

```

>>> import anyio
>>> from returns.io import IO, IOSuccess
>>> from returns.future import FutureResult

>>> async def main():
...     assert await FutureResult.from_io(
...         IO(1),
...     ) == IOSuccess(1)

>>> anyio.run(main)

```

Parameters**inner_value** (*IO*[TypeVar(_NewValueType)]) –**Return type***FutureResult*[TypeVar(_NewValueType), Any]**classmethod** `from_failed_io(inner_value)`Creates *FutureResult* from failed IO value.

```

>>> import anyio
>>> from returns.io import IO, IOFailure
>>> from returns.future import FutureResult

>>> async def main():
...     assert await FutureResult.from_failed_io(
...         IO(1),
...     ) == IOFailure(1)

>>> anyio.run(main)

```

Parameters**inner_value** (*IO*[TypeVar(_NewErrorType)]) –**Return type***FutureResult*[Any, TypeVar(_NewErrorType)]**classmethod** `from_ioresult(inner_value)`Creates *FutureResult* from *IOResult* value.

```

>>> import anyio
>>> from returns.io import IOSuccess, IOFailure
>>> from returns.future import FutureResult

```

(continues on next page)

(continued from previous page)

```

>>> async def main():
...     assert await FutureResult.from_ioresult(
...         IOSuccess(1),
...     ) == IOSuccess(1)
...     assert await FutureResult.from_ioresult(
...         IOFailure(1),
...     ) == IOFailure(1)
>>> anyio.run(main)

```

Parameters**inner_value** (*IOResult*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)])**Return type***FutureResult*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)]**classmethod from_result**(*inner_value*)Creates *FutureResult* from *Result* value.

```

>>> import anyio
>>> from returns.io import IOSuccess, IOFailure
>>> from returns.result import Success, Failure
>>> from returns.future import FutureResult

>>> async def main():
...     assert await FutureResult.from_result(
...         Success(1),
...     ) == IOSuccess(1)
...     assert await FutureResult.from_result(
...         Failure(1),
...     ) == IOFailure(1)
>>> anyio.run(main)

```

Parameters**inner_value** (*Result*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)])**Return type***FutureResult*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)]**classmethod from_value**(*inner_value*)Creates *FutureResult* from successful value.

```

>>> import anyio
>>> from returns.io import IOSuccess
>>> from returns.future import FutureResult

>>> async def main():
...     assert await FutureResult.from_value(
...         1,
...     ) == IOSuccess(1)

```

(continues on next page)

(continued from previous page)

```
>>> anyio.run(main)
```

Parameters

inner_value (TypeVar(_NewValueType)) –

Return type

FutureResult[TypeVar(_NewValueType), Any]

classmethod `from_failure(inner_value)`

Creates *FutureResult* from failed value.

```
>>> import anyio
>>> from returns.io import IOFailure
>>> from returns.future import FutureResult

>>> async def main():
...     assert await FutureResult.from_failure(
...         1,
...     ) == IOFailure(1)

>>> anyio.run(main)
```

Parameters

inner_value (TypeVar(_NewErrorType)) –

Return type

FutureResult[Any, TypeVar(_NewErrorType)]

FutureSuccess(inner_value)

Public unit function to create successful *FutureResult* objects.

Is the same as *from_value()*.

```
>>> import anyio
>>> from returns.future import FutureResult, FutureSuccess

>>> assert anyio.run(FutureSuccess(1).awaitable) == anyio.run(
...     FutureResult.from_value(1).awaitable,
... )
```

Parameters

inner_value (TypeVar(_NewValueType)) –

Return type

FutureResult[TypeVar(_NewValueType), Any]

FutureFailure(inner_value)

Public unit function to create failed *FutureResult* objects.

Is the same as *from_failure()*.

```

>>> import anyio
>>> from returns.future import FutureResult, FutureFailure

>>> assert anyio.run(FutureFailure(1).awaitable) == anyio.run(
...     FutureResult.from_failure(1).awaitable,
... )

```

Parameters

inner_value (TypeVar(_NewErrorType)) –

Return type

FutureResult[Any, TypeVar(_NewErrorType)]

FutureResultE

Alias for a popular case when Result has Exception as error type.

alias of *FutureResult*[_ValueType, Exception]

future_safe(function)

Decorator to convert exception-throwing coroutine to FutureResult.

Should be used with care, since it only catches Exception subclasses. It does not catch BaseException subclasses.

If you need to mark sync function as safe, use *returns.future.future_safe()* instead. This decorator only works with async functions. Example:

```

>>> import anyio
>>> from returns.future import future_safe
>>> from returns.io import IOSuccess, IOResult

>>> @future_safe
... async def might_raise(arg: int) -> float:
...     return 1 / arg
...

>>> assert anyio.run(might_raise(2).awaitable) == IOSuccess(0.5)
>>> assert isinstance(
...     anyio.run(might_raise(0).awaitable),
...     IOResult.failure_type,
... )

```

Similar to *returns.io.impure_safe()* and *returns.result.safe()* decorators, but works with async functions.

Parameters

function (Callable[[ParamSpec(_FuncParams)], Coroutine[TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ValueType, covariant=True)]] –

Return type

Callable[[ParamSpec(_FuncParams)], *FutureResult*[TypeVar(_ValueType, covariant=True), Exception]]

9.10 Context

Dependency injection is a popular software architecture pattern.

It's main idea is that you provide [Inversion of Control](#) and can pass different things into your logic instead of hardcoding you stuff. And by doing this you are on your way to achieve [Single Responsibility](#) for your functions and objects.

9.10.1 Using the context

A lot of programs we write rely on the context implicitly or explicitly. We can rely on configuration, env variables, stubs, logical dependencies, etc.

Let's look at the example.

Simple app

One of the most popular errors Python developers do in Django is that they overuse `settings` object inside the business logic. This makes your logic framework-oriented and hard to reason about in large projects.

Because values just pop out of nowhere in a deeply nested functions. And can be changed from the outside, from the context of your app.

Imagine that you have a django based game, where you award users with points for each guessed letter in a word (unguessed letters are marked as '. '):

```
from django.http import HttpRequest, HttpResponse
from words_app.logic import calculate_points

def view(request: HttpRequest) -> HttpResponse:
    user_word: str = request.POST['word'] # just an example
    points = calculate_points(user_word)
    ... # later you show the result to user somehow
```

```
# Somewhere in your `words_app/logic.py`:

def calculate_points(word: str) -> int:
    guessed_letters_count = len([letter for letter in word if letter != '.'])
    return _award_points_for_letters(guessed_letters_count)

def _award_points_for_letters(guessed: int) -> int:
    return 0 if guessed < 5 else guessed # minimum 6 points possible!
```

Straight and simple!

Adding configuration

But, later you decide to make the game more fun: let's make the minimal accountable letters threshold configurable for an extra challenge.

You can just do it directly:

```
def _award_points_for_letters(guessed: int, threshold: int) -> int:
    return 0 if guessed < threshold else guessed
```

And now your code won't simply type-check. Because that's how our caller looks like:

```
def calculate_points(word: str) -> int:
    guessed_letters_count = len([letter for letter in word if letter != '.'])
    return _award_points_for_letters(guessed_letters_count)
```

To fix this `calculate_points` function (and all other upper caller functions) will have to accept `threshold: int` as a parameter and pass it to `_award_points_for_letters`.

Imagine that your large project has multiple things to configure in multiple functions. What a mess it would be!

Ok, you can directly use `django.settings` (or similar) in your `_award_points_for_letters` function. And ruin your pure logic with framework-specific details. That's ugly!

Explicitly relying on context

We have learned that this tiny change showed us that it is not so easy to rely on implicit app context.

And instead of passing parameters for all callstack or using dirty framework specific magic you can use `RequiresContext` container. That was built just for this case.

Let's see how our code changes:

```
from django.conf import settings
from django.http import HttpRequest, HttpResponse
from words_app.logic import calculate_points

def view(request: HttpRequest) -> HttpResponse:
    user_word: str = request.POST['word'] # just an example
    points = calculate_points(user_words)(settings) # passing the dependencies
    ... # later you show the result to user somehow
```

```
# Somewhere in your `words_app/logic.py`:

from typing import Protocol
from returns.context import RequiresContext

class _Deps(Protocol): # we rely on abstractions, not direct values or types
    WORD_THRESHOLD: int

def calculate_points(word: str) -> RequiresContext[int, _Deps]:
    guessed_letters_count = len([letter for letter in word if letter != '.'])
    return _award_points_for_letters(guessed_letters_count)

def _award_points_for_letters(guessed: int) -> RequiresContext[int, _Deps]:
```

(continues on next page)

(continued from previous page)

```

return RequiresContext(
    lambda deps: 0 if guessed < deps.WORD_THRESHOLD else guessed,
)

```

And now you can pass your dependencies in a really direct and explicit way.

ask

Let's try to configure how we mark our unguessed letters (previously unguessed letters were marked as '.'). Let's say, we want to change this to be _.

How can we do that with our existing function?

```

def calculate_points(word: str) -> RequiresContext[int, _Deps]:
    guessed_letters_count = len([letter for letter in word if letter != '.'])
    return _award_points_for_letters(guessed_letters_count)

```

We are already using RequiresContext, but its dependencies are just hidden from us! We have a special helper for this case: `.ask()`, which returns us current dependencies.

The only thing we need to is to properly annotate the type for our case: `RequiresContext[int, _Deps].ask()` Sadly, currently mypy is not able to infer the dependency type out of the context and we need to explicitly provide it.

Let's see the final result:

```

from returns.context import RequiresContext

class _Deps(Protocol): # we rely on abstractions, not direct values or types
    WORD_THRESHOLD: int
    UNGUESSED_CHAR: str

def calculate_points(word: str) -> RequiresContext[int, _Deps]:
    def factory(deps: _Deps) -> RequiresContext[int, _Deps]:
        guessed_letters_count = len([
            letter for letter in word if letter != deps.UNGUESSED_CHAR
        ])
        return _award_points_for_letters(guessed_letters_count)

    return RequiresContext[int, _Deps].ask().bind(factory)

```

And now we access the current context from any place in our callstack. Isn't it convenient?

Warning: RequiresContext and similar types are not recursion safe. If you would have nesting of more than `sys.getrecursionlimit()` you will end up with `RecursionError`. Will this ever happen to you? Probably not.

9.10.2 RequiresContext container

The concept behind *RequiresContext* container is really simple. It is a container around `Callable[[EnvType], ReturnType]` function.

By its definition it works with pure functions that never fails.

It can be illustrated as a simple nested function:

```
>>> from typing import Callable
>>> def first(limit: int) -> Callable[[str], bool]:
...     def inner(deps: str) -> bool:
...         return len(deps) > limit
...     return inner

>>> assert first(2)('abc') # first(limit)(deps)
>>> assert not first(5)('abc') # first(limit)(deps)
```

That's basically enough to make dependency injection possible. But how would you compose *first* function? Let's say with the following function:

```
>>> def bool_to_str(arg: bool) -> str:
...     return 'ok' if arg else 'nope'
```

It would be hard, knowing that it returns another function to be called later when the context is known.

We can wrap it in *RequiresContext* container to allow better composition!

```
>>> from returns.context import RequiresContext

>>> def first(limit: int) -> RequiresContext[bool, str]:
...     def inner(deps: str) -> bool:
...         return len(deps) > limit
...     return RequiresContext(inner) # wrapping function here!

>>> assert first(1).map(bool_to_str)('abc') == 'ok'
>>> assert first(5).map(bool_to_str)('abc') == 'nope'
```

There's how execution flows:

Fig. 7: RequiresContext execution flow.

The rule is: the dependencies are injected at the very last moment in time. And then normal logical execution happens.

9.10.3 RequiresContextResult container

RequiresContextResult container is a combination of `RequiresContext[Result[a, b], env]`. Which means that it is a wrapper around pure function that might fail.

We also added a lot of useful methods for this container, so you can work easily with it:

- `bind_result()` allows to bind functions that return `Result` with just one call
- `bind_context()` allows to bind functions that return `RequiresContext` easily
- There are also several useful constructors from any possible type

Use it when you work with pure context-related functions that might fail.

9.10.4 RequiresContextIOResult container

RequiresContextIOResult container is a combination of `RequiresContext[IOResult[a, b], env]`. Which means that it is a wrapper around impure function that might fail.

We also added a lot of useful methods for this container, so you can work easily with it:

- `bind_result()` allows to bind functions that return `Result` with just one call
- `bind_io()` allows to bind functions that return `IO` with just one call
- `bind_ioresult()` allows to bind functions that return `IOResult` with just one call
- `bind_context()` allows to bind functions that return `RequiresContext` easily
- `bind_context_result()` allows to bind functions that return `RequiresContextResult` easily
- There are also several useful constructors from any possible type

Use it when you work with impure context-related functions that might fail. This is basically **the main type** that is going to be used in most apps.

9.10.5 RequiresContextFutureResult container

RequiresContextFutureResult container is a combination of `RequiresContext[FutureResult[a, b], env]`. Which means that it is a wrapper around impure async function that might fail.

Here's how it should be used:

```

1  from typing import Final, Sequence, cast
2
3  import anyio # you would need to `pip install anyio`
4  import httpx # you would need to `pip install httpx`
5  from typing_extensions import TypedDict
6
7  from returns.context import RequiresContextFutureResultE
8  from returns.functions import tap
9  from returns.future import FutureResultE, future_safe
10 from returns.iterables import Fold
11 from returns.pipeline import managed
12 from returns.result import ResultE, safe
13
14 _URL: Final = 'https://jsonplaceholder.typicode.com/posts/{0}'
15
16
17 class _Post(TypedDict):
18     id: int
19     user_id: int
20     title: str
21     body: str
22
23
24 def _close(
25     client: httpx.AsyncClient,
26     raw_value: ResultE[Sequence[str]],
27 ) -> FutureResultE[None]:
28     return future_safe(client.aclose)()

```

(continues on next page)

```

29
30
31 def _fetch_post(
32     post_id: int,
33 ) -> RequiresContextFutureResultE[_Post, httpx.AsyncClient]:
34     context: RequiresContextFutureResultE[
35         httpx.AsyncClient,
36         httpx.AsyncClient,
37     ] = RequiresContextFutureResultE.ask()
38
39     return context.bind_future_result(
40         lambda client: future_safe(client.get)(_URL.format(post_id)),
41     ).bind_result(
42         safe(tap(httpx.Response.raise_for_status)),
43     ).map(
44         lambda response: cast(_Post, response.json()), # or validate it
45     )
46
47
48 def _show_titles(
49     number_of_posts: int,
50 ) -> RequiresContextFutureResultE[Sequence[str], httpx.AsyncClient]:
51     def factory(post: _Post) -> str:
52         return post['title']
53
54     titles = [
55         # Notice how easily we compose async and sync functions:
56         _fetch_post(post_id).map(factory)
57         # TODO: try `for post_id in (2, 1, 0):` to see how errors work
58         for post_id in range(1, number_of_posts + 1)
59     ]
60     return Fold.collect(titles, RequiresContextFutureResultE.from_value(()))
61
62
63 if __name__ == '__main__':
64     # Let's fetch 3 titles of posts one-by-one, but with async client,
65     # because we want to highlight `managed` in this example:
66     managed_httpx = managed(_show_titles(3), _close)
67     future_result = managed_httpx(
68         FutureResultE.from_value(httpx.AsyncClient(timeout=5)),
69     )
70     print(anyio.run(future_result.awaitable)) # noqa: WPS421
71     # <IOResult: <Success: (
72     #     'sunt aut facere repellat provident occaecati ...',
73     #     'qui est esse',
74     #     'ea molestias quasi exercitationem repellat qui ipsa sit aut',
75     # )>>

```

This example illustrates the whole point of our actions: writing sync code that executes asynchronously without any magic at all!

We also added a lot of useful methods for this container, so you can work easily with it.

These methods are identical with `RequiresContextIOResult`:

- `bind_result()` allows to bind functions that return `Result` with just one call
- `bind_io()` allows to bind functions that return `IO` with just one call
- `bind_ioresult()` allows to bind functions that return `IOResult` with just one call
- `bind_future_result()` allows to bind functions that return `FutureResult` with just one call
- `bind_context()` allows to bind functions that return `RequiresContext` easily
- `bind_context_result()` allows to bind functions that return `RequiresContextResult` easily

There are new ones:

- `bind_future()` allows to bind functions that return `Future` container
- `bind_future_result()` allows to bind functions that return `FutureResult` container
- `bind_async_future()` allows to bind async functions that return `Future` container
- `bind_async_future_result()` allows to bind async functions that return `FutureResult` container
- `bind_context_ioresult()` allows to bind functions that return `RequiresContextIOResult`
- `bind_async()` allows to bind async functions that return `RequiresContextFutureResult` container
- `bind_awaitable()` allows to bind async function that return raw values

Use it when you work with impure context-related functions that might fail. This is basically **the main type** that is going to be used in most apps.

9.10.6 Aliases

There are several useful aliases for `RequiresContext` and friends with some common values:

- `Reader` is an alias for `RequiresContext[...]` to save you some typing. Uses `Reader` because it is a native name for this concept from Haskell.
- `RequiresContextResultE` is an alias for `RequiresContextResult[... , Exception]`, just use it when you want to work with `RequiresContextResult` containers that use exceptions as error type. It is named `ResultE` because it is `ResultException` and `ResultError` at the same time.
- `ReaderResult` is an alias for `RequiresContextResult[...]` to save you some typing.
- `ReaderResultE` is an alias for `RequiresContextResult[... , Exception]`
- `RequiresContextIOResultE` is an alias for `RequiresContextIOResult[... , Exception]`
- `ReaderIOResult` is an alias for `RequiresContextIOResult[...]` to save you some typing.
- `ReaderIOResultE` is an alias for `RequiresContextIOResult[... , Exception]`
- `RequiresContextFutureResultE` is an alias for `RequiresContextFutureResult[... , Exception]`
- `ReaderFutureResult` is an alias for `RequiresContextFutureResult[...]` to save you some typing.
- `ReaderFutureResultE` is an alias for `RequiresContextFutureResult[... , Exception]`

9.10.7 FAQ

How to create unit objects?

`RequiresContext` requires you to use one of the following methods:

- `from_value` when you have a raw value
- `from_requires_context_result` when you have `RequiresContextResult`
- `from_requires_context_ioresult` when you have `RequiresContextIOResult`

`RequiresContextResult` requires you to use one of the following methods:

- `from_value` when you want to mark some raw value as a `Success`
- `from_failure` when you want to mark some raw value as a `Failure`
- `from_result` when you already have `Result` container
- `from_context` when you have successful `RequiresContext`
- `from_failed_context` when you have failed `RequiresContext`
- `from_typecast` when you have `RequiresContext[... , Result]`

`RequiresContextIOResult` requires you to use one of the following methods:

- `from_value` when you want to mark some raw value as a `Success`
- `from_failure` when you want to mark some raw value as a `Failure`
- `from_result` when you already have `Result` container
- `from_io` when you have successful `IO` container
- `from_failed_io` when you have failed `IO` container
- `from_ioresult` when you already have `IOResult` container
- `from_context` when you have successful `RequiresContext` container
- `from_failed_context` when you have failed `RequiresContext` container
- `from_result_context` when you have `RequiresContextResult` container
- `from_typecast` when you have `RequiresContext[... , IOResult]`

`RequiresContextFutureResult` requires you to use one of the following methods:

- `from_value` when you want to mark some raw value as a `Success`
- `from_failure` when you want to mark some raw value as a `Failure`
- `from_result` when you already have `Result` container
- `from_io` when you have successful `IO` container
- `from_failed_io` when you have failed `IO` container
- `from_ioresult` when you already have `IOResult` container
- `from_future` when you already have successful `Future` container
- `from_failed_future` when you already have failed `Future` container
- `from_future_result` when you already have `FutureResult` container
- `from_context` when you have successful `RequiresContext`

- `from_failed_context` when you have failed `RequiresContext`
- `from_result_context` when you have `RequiresContextResult` container
- `from_ioresult_context` when you have `RequiresContextIOResult` container
- `from_typecast` when you have `RequiresContext[... , IOResult]`

How can I access dependencies inside the context?

Use `.ask()` method!

See *this guide*.

RequiresContext looks like a decorator with arguments

Yes, this container might remind a traditional decorator with arguments, let see an example:

```
>>> def example(print_result: bool):
...     def decorator(function):
...         def factory(*args, **kwargs):
...             original = function(*args, **kwargs)
...             if print_result:
...                 print(original)
...             return original
...         return factory
...     return decorator
```

And it can be used like so:

```
>>> @example(print_result=True)
... def my_function(first: int, second: int) -> int:
...     return first + second

>>> assert my_function(2, 3) == 5
5
```

We can model the similar idea with `RequiresContext`:

```
>>> from returns.context import RequiresContext

>>> def my_function(first: int, second: int) -> RequiresContext[int, bool]:
...     def factory(print_result: bool) -> int:
...         original = first + second
...         if print_result:
...             print(original)
...         return original
...     return RequiresContext(factory)

>>> assert my_function(2, 3)(False) == 5
>>> assert my_function(2, 3)(True) == 5
5
```

As you can see, it is easier to change the behaviour of a function with `RequiresContext`. While decorator with arguments glues values to a function forever. Decide when you need which behaviour carefully.

Why can't we use `RequiresContext[Result, e]` instead of `RequiresContextResult`?

We actually can! But, it is harder to write. And `RequiresContextResult` is actually the very same thing as `RequiresContext[Result, e]`, but has nicer API:

```
x: RequiresContext[Result[int, str], int]
x.map(lambda result: result.map(lambda number: number + 1))

# Is the same as:

y: RequiresContextResult[int, str, int]
y.map(lambda number: number + 1)
```

The second one looks better, doesn't it? The same applies for `RequiresContextIOResult` and `RequiresContextFutureResult` as well.

Why do I have to use explicit type annotation for ask method?

Because mypy cannot possibly know the type of current context. This is hard even for a plugin.

So, using this technique is better:

```
from returns.context import RequiresContext

def some_context(*args, **kwargs) -> RequiresContext[str, int]:
    def factory(deps: int) -> RequiresContext[str, int]:
        ...
    return RequiresContext[str, int].ask().bind(factory)
```

What is the difference between DI and `RequiresContext`?

Dependency Injection pattern and [Inversion of Control](#) principle forms a lot of ideas and tooling that do pretty much the same as `RequiresContext` container.

What is the difference? Why do we need each of them?

Let's find out! Tools like [dependencies](#) or [punq](#) tries to:

1. Inspect (by name or type respectively) function or class that needs dependencies
2. Build the required dependency tree from the source defined in the service container

There are other tools like `inject` that also invades your code with `@inject` decorator.

`RequiresContext` works completely different. It respects your code and does not try to inspect in any manner. It also does not care about building dependencies at all.

All it does is: provides simple API to compose functions that need additional context (or dependencies) to run.

You can even use them together: `RequiresContext` will pass dependencies built by `punq` (or any other tool of your choice) as a `deps` parameter to `RequiresContext` instance.

When to use which? Let's dig into it!

- `RequiresContext` offers explicit context passing for the whole function stack inside your program. This means two things: you will have to pass it through all your code and use it everywhere inside your program explicitly, when you need to access the environment and dependencies

- Traditional DI allows to leave a lot of code unaware of dependency injection. Because you don't have to maintain the context everywhere. You just need to adjust your API to meet the dependency injector requirements. On the other hand, you lose explicitness here.

So when to use `RequiresContext`?

1. When you write pure functional code
2. When you want to know which code relies on context and which is free from it, `RequiresContext` makes this explicit and typed
3. When you rely on types inside your program
4. When you want to rely on functions rather than magic

When not to use `RequiresContext` and use traditional DI?

1. When you already have a lot of code written in a different approach: in OOP and/or imperative styles
2. When you need to pass dependencies into a very deep level of your call stack implicitly (without modifying the whole stack), this is called magic
3. When you not rely on types for dependencies. There are cases when DI is made by names or tags

Here's an example that might give you a better understanding of how `RequiresContext` is used on real and rather big projects:

```
from typing import Callable, Dict, Protocol, final
from returns.io import IOResultE
from returns.context import ReaderIOResultE

class _SyncPermissionsDeps(Protocol):
    fetch_metadata: Callable[[], IOResultE['Metadata']]
    get_user_permissions: Callable[['Metadata'], Dict[int, str]] # pure
    update_bi_permissions: Callable[[Dict[int, str]], IOResultE['Payload']]

def sync_permissions() -> ReaderIOResultE[_SyncPermissionsDeps, 'Payload']:
    """
    This functions runs a scheduled task once a day.

    It syncs permissions from the metadata storage to our BI system.
    """
    def factory(deps: _SyncPermissionsDeps) -> IOResultE['Payload']:
        return deps.fetch_metadata().map(
            deps.get_user_permissions,
        ).bind_ioresult(
            deps.update_bi_permissions,
        )
    return ReaderIOResult(factory)
```

And then it is called like so:

```
# tasks.py
from celery import shared_task
from returns.functions import raise_exception

from logic.usecases.sync_permissions import sync_permissions
from infrastructure.implemented import Container
```

(continues on next page)

```

from infrastructure.services import bi
from infrastructure.repositories import db

@shared_task(autoretry_for=(ConnectionError,), max_retries=3)
def queue_sync_permissions():
    # Building the container with dependencies to pass it into the context.
    # We also make sure that we don't forget to raise internal exceptions
    # and trigger celery retries.
    return sync_permissions().alt(raise_exception)(Container(
        fetch_metadata=db.select_user_metadata,
        get_user_permissions=bi.permissions_from_user,
        update_bi_permissions=bi.put_user_permissions,
    ))

```

9.10.8 Further reading

- Enforcing Single Responsibility Principle in Python
- Typed functional Dependency Injection in Python
- Three-Useful-Monads: Reader
- Getting started with fp-ts: Reader
- Reader & Constructor-based Dependency Injection in Scala - friend or foe?

9.10.9 API Reference

RequiresContext

NoDeps = `typing.Any`

Sometimes `RequiresContext` and other similar types might be used with no explicit dependencies so we need to have this type alias for `Any`.

final class `RequiresContext`(*inner_value*)

Bases: `BaseContainer`, `SupportsKindN[RequiresContext, _ReturnType, _EnvType, NoReturn]`, `ReaderBased2[_ReturnType, _EnvType]`

The `RequiresContext` container.

It's main purpose is to wrap some specific function and to provide tools to compose other functions around it without actually calling it.

The `RequiresContext` container passes the state you want to share between functions. Functions may read that state, but can't change it. The `RequiresContext` container lets us access shared immutable state within a specific context.

It can be used for lazy evaluation and typed dependency injection.

`RequiresContext` is used with functions that never fail. If you want to use `RequiresContext` with `returns.Result` then consider using `RequiresContextResult` instead.

Note: This container does not wrap ANY value. It wraps only functions. You won't be able to supply arbitrary types!

See also:

- <https://dev.to/gcanti/getting-started-with-fp-ts-reader-lie5>
- https://en.wikipedia.org/wiki/Lazy_evaluation
- <https://bit.ly/2R814WK>

Parameters

inner_value (Callable[[TypeVar(_EnvType), TypeVar(_ReturnType, covariant=True)], TypeVar(_ReturnType, covariant=True)]) –

no_args: `ClassVar[NoDeps] = <object object>`

A convenient placeholder to call methods created by `.from_value()`:

map(*function*)

Allows to compose functions inside the wrapped container.

Here's how it works:

```
>>> from returns.context import RequiresContext
>>> def first(lg: bool) -> RequiresContext[int, float]:
...     # `deps` has `float` type here:
...     return RequiresContext(
...         lambda deps: deps if lg else -deps,
...     )
>>> assert first(True).map(lambda number: number * 10)(2.5) == 25.0
>>> assert first(False).map(lambda number: number * 10)(0.1) == -1.0
```

Parameters

function (Callable[[TypeVar(_ReturnType), TypeVar(_NewReturnType)], TypeVar(_NewReturnType)]) –

Return type

`RequiresContext[TypeVar(_NewReturnType), TypeVar(_EnvType, contravariant=True)]`

apply(*container*)

Calls a wrapped function in a container on this container.

```
>>> from returns.context import RequiresContext
>>> assert RequiresContext.from_value('a').apply(
...     RequiresContext.from_value(lambda inner: inner + 'b')
... )(...) == 'ab'
```

Parameters

container (`KindN[RequiresContext, Callable[[TypeVar(_ReturnType, covariant=True)], TypeVar(_NewReturnType)], TypeVar(_EnvType, contravariant=True), Any]`) –

Return type

`RequiresContext[TypeVar(_NewReturnType), TypeVar(_EnvType, contravariant=True)]`

bind(*function*)

Composes a container with a function returning another container.

This is useful when you do several computations that rely on the same context.

```
>>> from returns.context import RequiresContext

>>> def first(lg: bool) -> RequiresContext[int, float]:
...     # `deps` has `float` type here:
...     return RequiresContext(
...         lambda deps: deps if lg else -deps,
...     )

>>> def second(number: int) -> RequiresContext[str, float]:
...     # `deps` has `float` type here:
...     return RequiresContext(
...         lambda deps: '>=' if number >= deps else '<',
...     )

>>> assert first(True).bind(second)(1) == '>='
>>> assert first(False).bind(second)(2) == '<'
```

Parameters

function (Callable[[TypeVar(_ReturnType), covariant=True]], KindN[RequiresContext, TypeVar(_NewReturnType), TypeVar(_EnvType, contravariant=True), Any]]) –

Return type

RequiresContext[TypeVar(_NewReturnType), TypeVar(_EnvType, contravariant=True)]

bind_context(*function*)

Alias for `bind_context` method, it is the same as `bind` here.

Parameters

function (Callable[[TypeVar(_ReturnType), covariant=True]], KindN[RequiresContext, TypeVar(_NewReturnType), TypeVar(_EnvType, contravariant=True), Any]]) –

Return type

RequiresContext[TypeVar(_NewReturnType), TypeVar(_EnvType, contravariant=True)]

modify_env(*function*)

Allows to modify the environment type.

```
>>> from returns.context import RequiresContext

>>> def mul(arg: int) -> RequiresContext[float, int]:
...     return RequiresContext(lambda deps: arg * deps)

>>> assert mul(3).modify_env(int)('2') == 6
```

Parameters

```
function (Callable[[TypeVar(_NewEnvType)], TypeVar(_EnvType, contravariant=True)]) –
```

Return type

```
RequiresContext[TypeVar(_ReturnType, covariant=True), TypeVar(_NewEnvType)]
```

classmethod ask()

Get current context to use the dependencies.

It is a common scenario when you need to use the environment. For example, you want to do some context-related computation, but you don't have the context instance at your disposal. That's where `.ask()` becomes useful!

```
>>> from typing_extensions import TypedDict
>>> class Deps(TypedDict):
...     message: str

>>> def first(lg: bool) -> RequiresContext[int, Deps]:
...     # `deps` has `Deps` type here:
...     return RequiresContext(
...         lambda deps: deps['message'] if lg else 'error',
...     )

>>> def second(text: str) -> RequiresContext[int, Deps]:
...     return first(len(text) > 3)

>>> assert second('abc')({'message': 'ok'}) == 'error'
>>> assert second('abcd')({'message': 'ok'}) == 'ok'
```

And now imagine that you have to change this 3 limit. And you want to be able to set it via environment as well. Ok, let's fix it with the power of `RequiresContext.ask()`!

```
>>> from typing_extensions import TypedDict
>>> class Deps(TypedDict):
...     message: str
...     limit: int # note this new field!

>>> def new_first(lg: bool) -> RequiresContext[int, Deps]:
...     # `deps` has `Deps` type here:
...     return RequiresContext(
...         lambda deps: deps['message'] if lg else 'err',
...     )

>>> def new_second(text: str) -> RequiresContext[int, Deps]:
...     return RequiresContext[int, Deps].ask().bind(
...         lambda deps: new_first(len(text) > deps.get('limit', 3)),
...     )

>>> assert new_second('abc')({'message': 'ok', 'limit': 2}) == 'ok'
>>> assert new_second('abcd')({'message': 'ok'}) == 'ok'
>>> assert new_second('abcd')({'message': 'ok', 'limit': 5}) == 'err'
```

That's how `ask` works.

This class contains methods that require to explicitly set type annotations. Why? Because it is impossible to figure out the type without them. So, here's how you should use them:

```
RequiresContext[int, Dict[str, str]].ask()
```

Otherwise, your `.ask()` method will return `RequiresContext[<nothing>, <nothing>]`, which is unusable:

```
env = RequiresContext.ask()
env(some_deps)
```

And `mypy` will warn you: `error: Need type annotation for '...' :rtype: RequiresContext[TypeVar(_EnvType, contravariant=True), TypeVar(_EnvType, contravariant=True)]`

See also:

- <https://dev.to/gcanti/getting-started-with-fp-ts-reader-lie5>

classmethod `from_value(inner_value)`

Used to return some specific value from the container.

Consider this method as some kind of factory. Passed value will be a return type. Make sure to use `no_args` for getting the unit value.

```
>>> from returns.context import RequiresContext
>>> unit = RequiresContext.from_value(5)
>>> assert unit(RequiresContext.no_args) == 5
```

Might be used with or without direct type hint.

Parameters

`inner_value` (`TypeVar(_FirstType)`) –

Return type

`RequiresContext`[`TypeVar(_FirstType)`, `Any`]

classmethod `from_context(inner_value)`

Used to create new containers from existing ones.

Used as a part of `ReaderBased2` interface.

```
>>> from returns.context import RequiresContext
>>> unit = RequiresContext.from_value(5)
>>> assert RequiresContext.from_context(unit)(...) == unit(...)
```

Parameters

`inner_value` (`RequiresContext`[`TypeVar(_NewReturnType)`, `TypeVar(_NewEnvType)`]) –

Return type

`RequiresContext`[`TypeVar(_NewReturnType)`, `TypeVar(_NewEnvType)`]

classmethod `from_requires_context_result(inner_value)`

Typecasts `RequiresContextResult` to `RequiresContext` instance.

Breaks `RequiresContextResult[a, b, e]` into `RequiresContext[Result[a, b], e]`.

```

>>> from returns.context import RequiresContext
>>> from returns.context import RequiresContextResult
>>> from returns.result import Success
>>> assert RequiresContext.from_requires_context_result(
...     RequiresContextResult.from_value(1),
... )(...) == Success(1)

```

Can be reverted with `RequiresContextResult.from_typecast`.

Parameters

inner_value (*RequiresContextResult*[*TypeVar*(*_ValueType*), *TypeVar*(*_ErrorType*), *TypeVar*(*_EnvType*, *contravariant=True*)]) –

Return type

RequiresContext[*Result*[*TypeVar*(*_ValueType*), *TypeVar*(*_ErrorType*)], *TypeVar*(*_EnvType*, *contravariant=True*)]

classmethod `from_requires_context_ioresult`(*inner_value*)

Typecasts `RequiresContextIOResult` to `RequiresContext` instance.

Breaks `RequiresContextIOResult[a, b, e]` into `RequiresContext[IOResult[a, b], e]`.

```

>>> from returns.context import RequiresContext
>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess
>>> assert RequiresContext.from_requires_context_ioresult(
...     RequiresContextIOResult.from_value(1),
... )(...) == IOSuccess(1)

```

Can be reverted with `RequiresContextIOResult.from_typecast`.

Parameters

inner_value (*RequiresContextIOResult*[*TypeVar*(*_ValueType*), *TypeVar*(*_ErrorType*), *TypeVar*(*_EnvType*, *contravariant=True*)]) –

Return type

RequiresContext[*IOResult*[*TypeVar*(*_ValueType*), *TypeVar*(*_ErrorType*)], *TypeVar*(*_EnvType*, *contravariant=True*)]

classmethod `from_requires_context_future_result`(*inner_value*)

Typecasts `RequiresContextIOResult` to `RequiresContext` instance.

Breaks `RequiresContextIOResult[a, b, e]` into `RequiresContext[IOResult[a, b], e]`.

```

>>> import anyio
>>> from returns.context import RequiresContext
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOSuccess

>>> container = RequiresContext.from_requires_context_future_result(
...     RequiresContextFutureResult.from_value(1),
... )
>>> assert anyio.run(
...     container, RequiresContext.no_args,
... ) == IOSuccess(1)

```

Can be reverted with `RequiresContextFutureResult.from_typecast`.

Parameters

inner_value (*RequiresContextFutureResult*[TypeVar(_ValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]) –

Return type

RequiresContext[*FutureResult*[TypeVar(_ValueType), TypeVar(_ErrorType)], TypeVar(_EnvType, contravariant=True)]

Reader

Sometimes *RequiresContext* is too long to type.

Parameters

inner_value (Callable[[TypeVar(_EnvType, contravariant=True)], TypeVar(_ReturnType, covariant=True)]) –

RequiresContextResult

final class RequiresContextResult(*inner_value*)

Bases: *BaseContainer*, *SupportsKindN*[*RequiresContextResult*, _ValueType, _ErrorType, _EnvType], *ReaderResultBasedN*[_ValueType, _ErrorType, _EnvType]

The *RequiresContextResult* combinator.

See *returns.context.requires_context.RequiresContext* for more docs.

This is just a handy wrapper around *RequiresContext*[*Result*[a, b], env] which represents a context-dependent pure operation that might fail and return *returns.result.Result*.

It has several important differences from the regular *Result* classes. It does not have *Success* and *Failure* subclasses. Because, the computation is not yet performed. And we cannot know the type in advance.

So, this is a thin wrapper, without any changes in logic.

Why do we need this wrapper? That's just for better usability!

```
>>> from returns.context import RequiresContext
>>> from returns.result import Success, Result

>>> def function(arg: int) -> Result[int, str]:
...     return Success(arg + 1)

>>> # Without wrapper:
>>> assert RequiresContext.from_value(Success(1)).map(
...     lambda result: result.bind(function),
... )(...) == Success(2)

>>> # With wrapper:
>>> assert RequiresContextResult.from_value(1).bind_result(
...     function,
... )(...) == Success(2)
```

This way *RequiresContextResult* allows to simply work with:

- raw values and pure functions
- *RequiresContext* values and pure functions returning it
- *Result* and functions returning it

Important implementation details

Due to its meaning, `RequiresContextResult` cannot have `Success` and `Failure` subclasses.

We only have just one type. That's by design.

Different converters are also not supported for this type. Use converters inside the `RequiresContext` context, not outside.

See also:

- <https://dev.to/gcanti/getting-started-with-fp-ts-reader-1ie5>
- https://en.wikipedia.org/wiki/Lazy_evaluation
- <https://bit.ly/2R8I4WK>
- <https://bit.ly/2RwP4fp>

Parameters

inner_value (Callable[[TypeVar(_EnvType, contravariant=True)], `Result`[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType, covariant=True)]])

no_args: `ClassVar[NoDeps] = <object object>`

A convenient placeholder to call methods created by `.from_value()`.

swap()

Swaps value and error types.

So, values become errors and errors become values. It is useful when you have to work with errors a lot. And since we have a lot of `.bind_` related methods and only a single `.lash` - it is easier to work with values.

```
>>> from returns.context import RequiresContextResult
>>> from returns.result import Failure, Success

>>> success = RequiresContextResult.from_value(1)
>>> failure = RequiresContextResult.from_failure(1)

>>> assert success.swap(...) == Failure(1)
>>> assert failure.swap(...) == Success(1)
```

Return type

`RequiresContextResult`[TypeVar(_ErrorType, covariant=True), TypeVar(_ValueType, covariant=True), TypeVar(_EnvType, contravariant=True)]

map(function)

Composes successful container with a pure function.

```
>>> from returns.context import RequiresContextResult
>>> from returns.result import Success, Failure

>>> assert RequiresContextResult.from_value(1).map(
...     lambda x: x + 1,
... )(...) == Success(2)
```

(continues on next page)

(continued from previous page)

```
>>> assert RequiresContextResult.from_failure(1).map(
...     lambda x: x + 1,
... )(...) == Failure(1)
```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)]] – covariant=True),

Return type

RequiresContextResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

apply(*container*)

Calls a wrapped function in a container on this container.

```
>>> from returns.context import RequiresContextResult
>>> from returns.result import Success, Failure, Result

>>> def transform(arg: str) -> str:
...     return arg + 'b'

>>> assert RequiresContextResult.from_value('a').apply(
...     RequiresContextResult.from_value(transform),
... )(...) == Success('ab')

>>> assert RequiresContextResult.from_failure('a').apply(
...     RequiresContextResult.from_value(transform),
... )(...) == Failure('a')

>>> assert isinstance(RequiresContextResult.from_value('a').apply(
...     RequiresContextResult.from_failure(transform),
... )(…), Result.failure_type) is True
```

Parameters

container (*KindN*[*RequiresContextResult*, Callable[[TypeVar(_ValueType, covariant=True)], TypeVar(_NewValueType)], TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]] –

Return type

RequiresContextResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

bind(*function*)

Composes this container with a function returning the same type.

```
>>> from returns.context import RequiresContextResult
>>> from returns.result import Success, Failure

>>> def first(lg: bool) -> RequiresContextResult[int, int, float]:
...     # `deps` has `float` type here:
...     return RequiresContextResult(
...         lambda deps: Success(deps) if lg else Failure(-deps),
```

(continues on next page)

(continued from previous page)

```

...     )

>>> def second(
...     number: int,
... ) -> RequiresContextResult[str, int, float]:
...     # `deps` has `float` type here:
...     return RequiresContextResult(
...         lambda deps: Success('>=' if number >= deps else '<'),
...     )

>>> assert first(True).bind(second)(1) == Success('>=')
>>> assert first(False).bind(second)(2) == Failure('<')

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], KindN[RequiresContextResult, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]])

Return type

RequiresContextResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_context_result(function)

Alias for `bind_context_result` method, it is the same as `bind` here.

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], KindN[RequiresContextResult, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]])

Return type

RequiresContextResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_result(function)

Binds Result returning function to current container.

```

>>> from returns.context import RequiresContextResult
>>> from returns.result import Success, Failure, Result

>>> def function(num: int) -> Result[str, int]:
...     return Success(num + 1) if num > 0 else Failure('<0')

>>> assert RequiresContextResult.from_value(1).bind_result(
...     function,
... )(RequiresContextResult.no_args) == Success(2)

>>> assert RequiresContextResult.from_value(0).bind_result(
...     function,
... )(RequiresContextResult.no_args) == Failure('<0')

>>> assert RequiresContextResult.from_failure(':', ':').bind_result(
...     function,
... )(RequiresContextResult.no_args) == Failure(':', ':')

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)],
Result[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]] –

Return type

RequiresContextResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_context(function)

Binds RequiresContext returning function to current container.

```
>>> from returns.context import RequiresContext
>>> from returns.result import Success, Failure

>>> def function(arg: int) -> RequiresContext[int, str]:
...     return RequiresContext(lambda deps: len(deps) + arg)

>>> assert function(2)('abc') == 5

>>> assert RequiresContextResult.from_value(2).bind_context(
...     function,
... )('abc') == Success(5)

>>> assert RequiresContextResult.from_failure(2).bind_context(
...     function,
... )('abc') == Failure(2)
```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)],
RequiresContext[TypeVar(_NewValueType), TypeVar(_EnvType, contravariant=True)]] –

Return type

RequiresContextResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

alt(function)

Composes failed container with a pure function.

```
>>> from returns.context import RequiresContextResult
>>> from returns.result import Success, Failure

>>> assert RequiresContextResult.from_value(1).alt(
...     lambda x: x + 1,
... )(...) == Success(1)

>>> assert RequiresContextResult.from_failure(1).alt(
...     lambda x: x + 1,
... )(...) == Failure(2)
```

Parameters

function (Callable[[TypeVar(_ErrorType, covariant=True)],
TypeVar(_NewErrorType)] –

Return type

`RequiresContextResult`[`TypeVar`(`_ValueType`, `covariant=True`),
`TypeVar`(`_NewErrorType`), `TypeVar`(`_EnvType`, `contravariant=True`)]

lash(*function*)

Composes this container with a function returning the same type.

```
>>> from returns.context import RequiresContextResult
>>> from returns.result import Success, Failure

>>> def lashable(arg: str) -> RequiresContextResult[str, str, str]:
...     if len(arg) > 1:
...         return RequiresContextResult(
...             lambda deps: Success(deps + arg),
...         )
...     return RequiresContextResult(
...         lambda deps: Failure(arg + deps),
...     )

>>> assert RequiresContextResult.from_value('a').lash(
...     lashable,
... )('c') == Success('a')
>>> assert RequiresContextResult.from_failure('a').lash(
...     lashable,
... )('c') == Failure('ac')
>>> assert RequiresContextResult.from_failure('aa').lash(
...     lashable,
... )('b') == Success('baa')
```

Parameters

function (`Callable`[[`TypeVar`(`_ErrorType`, `covariant=True`)],
`KindN`[`RequiresContextResult`, `TypeVar`(`_ValueType`, `covariant=True`),
`TypeVar`(`_NewErrorType`), `TypeVar`(`_EnvType`, `contravariant=True`)]) –

Return type

`RequiresContextResult`[`TypeVar`(`_ValueType`, `covariant=True`),
`TypeVar`(`_NewErrorType`), `TypeVar`(`_EnvType`, `contravariant=True`)]

modify_env(*function*)

Allows to modify the environment type.

```
>>> from returns.context import RequiresContextResultE
>>> from returns.result import Success, safe

>>> def div(arg: int) -> RequiresContextResultE[float, int]:
...     return RequiresContextResultE(
...         safe(lambda deps: arg / deps),
...     )

>>> assert div(3).modify_env(int>('2')) == Success(1.5)
>>> assert div(3).modify_env(int>('0')).failure()
```

Parameters

```
function (Callable[[TypeVar(_NewEnvType)], TypeVar(_EnvType, contravariant=True)]) –
```

Return type

```
RequiresContextResult[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType, covariant=True), TypeVar(_NewEnvType)]
```

classmethod `ask()`

Is used to get the current dependencies inside the call stack.

Similar to `returns.context.requires_context.RequiresContext.ask()`, but returns `Result` instead of a regular value.

Please, refer to the docs there to learn how to use it.

One important note that is worth duplicating here: you might need to provide `_EnvType` explicitly, so mypy will know about it statically.

```
>>> from returns.context import RequiresContextResultE
>>> from returns.result import Success
>>> assert RequiresContextResultE[int, int].ask().map(
...     str,
... )(1) == Success('1')
```

Return type

```
RequiresContextResult[TypeVar(_EnvType, contravariant=True), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]
```

classmethod `from_result(inner_value)`

Creates new container with `Result` as a unit value.

```
>>> from returns.context import RequiresContextResult
>>> from returns.result import Success, Failure
>>> deps = RequiresContextResult.no_args

>>> assert RequiresContextResult.from_result(
...     Success(1),
... )(deps) == Success(1)

>>> assert RequiresContextResult.from_result(
...     Failure(1),
... )(deps) == Failure(1)
```

Parameters

```
inner_value (Result[TypeVar(_NewValueType), TypeVar(_NewErrorType)]) –
```

Return type

```
RequiresContextResult[TypeVar(_NewValueType), TypeVar(_NewErrorType), Any]
```

classmethod `from_typecast(inner_value)`

You might end up with `RequiresContext[Result[...]]` as a value.

This method is designed to turn it into `RequiresContextResult`. It will save all the typing information.

It is just more useful!

```

>>> from returns.context import RequiresContext
>>> from returns.result import Success, Failure

>>> assert RequiresContextResult.from_typecast(
...     RequiresContext.from_value(Success(1)),
... )(RequiresContextResult.no_args) == Success(1)

>>> assert RequiresContextResult.from_typecast(
...     RequiresContext.from_value(Failure(1)),
... )(RequiresContextResult.no_args) == Failure(1)

```

Parameters

inner_value (*RequiresContext*[*Result*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)], *TypeVar*(*_EnvType*, contravariant=True)]) –

Return type

RequiresContextResult[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*), *TypeVar*(*_EnvType*, contravariant=True)]

classmethod from_context(*inner_value*)

Creates new container from *RequiresContext* as a success unit.

```

>>> from returns.context import RequiresContext
>>> from returns.result import Success
>>> assert RequiresContextResult.from_context(
...     RequiresContext.from_value(1),
... )(...) == Success(1)

```

Parameters

inner_value (*RequiresContext*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewEnvType*)]) –

Return type

RequiresContextResult[*TypeVar*(*_NewValueType*), *Any*, *TypeVar*(*_NewEnvType*)]

classmethod from_failed_context(*inner_value*)

Creates new container from *RequiresContext* as a failure unit.

```

>>> from returns.context import RequiresContext
>>> from returns.result import Failure
>>> assert RequiresContextResult.from_failed_context(
...     RequiresContext.from_value(1),
... )(...) == Failure(1)

```

Parameters

inner_value (*RequiresContext*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewEnvType*)]) –

Return type

RequiresContextResult[*Any*, *TypeVar*(*_NewValueType*), *TypeVar*(*_NewEnvType*)]

classmethod from_result_context(*inner_value*)

Creates *RequiresContextResult* from another instance of it.

```

>>> from returns.context import ReaderResult
>>> from returns.result import Success, Failure

>>> assert ReaderResult.from_result_context(
...     ReaderResult.from_value(1),
... )(...) == Success(1)

>>> assert ReaderResult.from_result_context(
...     ReaderResult.from_failure(1),
... )(...) == Failure(1)

```

Parameters

inner_value (*RequiresContextResult*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*), *TypeVar*(*_NewEnvType*)]) –

Return type

RequiresContextResult[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*), *TypeVar*(*_NewEnvType*)]

classmethod from_value(*inner_value*)

Creates new container with `Success(inner_value)` as a unit value.

```

>>> from returns.context import RequiresContextResult
>>> from returns.result import Success
>>> assert RequiresContextResult.from_value(1)(...) == Success(1)

```

Parameters

inner_value (*TypeVar*(*_FirstType*)) –

Return type

RequiresContextResult[*TypeVar*(*_FirstType*), *Any*, *Any*]

classmethod from_failure(*inner_value*)

Creates new container with `Failure(inner_value)` as a unit value.

```

>>> from returns.context import RequiresContextResult
>>> from returns.result import Failure
>>> assert RequiresContextResult.from_failure(1)(...) == Failure(1)

```

Parameters

inner_value (*TypeVar*(*_FirstType*)) –

Return type

RequiresContextResult[*Any*, *TypeVar*(*_FirstType*), *Any*]

RequiresContextResultE

Alias for a popular case when `Result` has `Exception` as error type.

alias of *RequiresContextResult*[*_ValueType*, *Exception*, *_EnvType*]

ReaderResult

Alias to save you some typing. Uses original name from Haskell.

Parameters

inner_value (Callable[[TypeVar(_EnvType, contravariant=True)],
Result[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType, covariant=True)]] –

ReaderResultE

Alias to save you some typing. Has *Exception* as error type.

alias of *RequiresContextResult*[_ValueType, *Exception*, _EnvType]

RequiresContextIOResult

final class *RequiresContextIOResult*(*inner_value*)

Bases: *BaseContainer*, *SupportsKindN*[*RequiresContextIOResult*, _ValueType, _ErrorType, _EnvType], *ReaderIOResultBasedN*[_ValueType, _ErrorType, _EnvType]

The *RequiresContextIOResult* combinator.

See *returns.context.requires_context.RequiresContext* and *returns.context.requires_context_result.RequiresContextResult* for more docs.

This is just a handy wrapper around *RequiresContext*[*IOResult*[a, b], env] which represents a context-dependent impure operation that might fail.

It has several important differences from the regular *Result* classes. It does not have *Success* and *Failure* subclasses. Because, the computation is not yet performed. And we cannot know the type in advance.

So, this is a thin wrapper, without any changes in logic.

Why do we need this wrapper? That's just for better usability!

```
>>> from returns.context import RequiresContext
>>> from returns.io import IOSuccess, IOResult

>>> def function(arg: int) -> IOResult[int, str]:
...     return IOSuccess(arg + 1)

>>> # Without wrapper:
>>> assert RequiresContext.from_value(IOSuccess(1)).map(
...     lambda ioresult: ioresult.bind(function),
... )(...) == IOSuccess(2)

>>> # With wrapper:
>>> assert RequiresContextIOResult.from_value(1).bind_ioresult(
...     function,
... )(...) == IOSuccess(2)
```

This way *RequiresContextIOResult* allows to simply work with:

- raw values and pure functions
- *RequiresContext* values and pure functions returning it
- *RequiresContextResult* values and pure functions returning it
- *Result* and pure functions returning it
- *IOResult* and functions returning it
- other *RequiresContextIOResult* related functions and values

This is a complex type for complex tasks!

Important implementation details

Due to its meaning, `RequiresContextIOResult` cannot have `Success` and `Failure` subclasses.

We only have just one type. That's by design.

Different converters are also not supported for this type. Use converters inside the `RequiresContext` context, not outside.

See also:

- <https://dev.to/gcanti/getting-started-with-fp-ts-reader-lie5>
- https://en.wikipedia.org/wiki/Lazy_evaluation
- <https://bit.ly/2R8I4WK>
- <https://bit.ly/2RwP4fp>

Parameters

inner_value (Callable[[TypeVar(_EnvType), TypeVar(_ErrorType)], IOResult[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType)]] –

no_args: `ClassVar[NoDeps] = <object object>`

A convenient placeholder to call methods created by `.from_value()`.

swap()

Swaps value and error types.

So, values become errors and errors become values. It is useful when you have to work with errors a lot. And since we have a lot of `.bind_` related methods and only a single `.lash` - it is easier to work with values.

```
>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess, IOFailure

>>> success = RequiresContextIOResult.from_value(1)
>>> failure = RequiresContextIOResult.from_failure(1)

>>> assert success.swap(...) == IOFailure(1)
>>> assert failure.swap(...) == IOSuccess(1)
```

Return type

`RequiresContextIOResult[TypeVar(_ErrorType), TypeVar(_ValueType, covariant=True), TypeVar(_EnvType, contravariant=True)]`

map(function)

Composes successful container with a pure function.

```
>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess, IOFailure

>>> assert RequiresContextIOResult.from_value(1).map(
...     lambda x: x + 1,
... )(...) == IOSuccess(2)
```

(continues on next page)

(continued from previous page)

```
>>> assert RequiresContextIOResult.from_failure(1).map(
...     lambda x: x + 1,
... )(...) == IOFailure(1)
```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)]] – covariant=True)],

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]

apply(container)

Calls a wrapped function in a container on this container.

```
>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess, IOFailure

>>> def transform(arg: str) -> str:
...     return arg + 'b'

>>> assert RequiresContextIOResult.from_value('a').apply(
...     RequiresContextIOResult.from_value(transform),
... )(...) == IOSuccess('ab')

>>> assert RequiresContextIOResult.from_value('a').apply(
...     RequiresContextIOResult.from_failure(1),
... )(...) == IOFailure(1)

>>> assert RequiresContextIOResult.from_failure('a').apply(
...     RequiresContextIOResult.from_value(transform),
... )(...) == IOFailure('a')

>>> assert RequiresContextIOResult.from_failure('a').apply(
...     RequiresContextIOResult.from_failure('b'),
... )(...) == IOFailure('a')
```

Parameters

container (*KindN*[*RequiresContextIOResult*, Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)], TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]] –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]

bind(function)

Composes this container with a function returning the same type.

```
>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess, IOFailure
```

(continues on next page)

(continued from previous page)

```

>>> def first(lg: bool) -> RequiresContextIOResult[int, int, float]:
...     # `deps` has `float` type here:
...     return RequiresContextIOResult(
...         lambda deps: IOSuccess(deps) if lg else IOFailure(-deps),
...     )
>>> def second(
...     number: int,
... ) -> RequiresContextIOResult[str, int, float]:
...     # `deps` has `float` type here:
...     return RequiresContextIOResult(
...         lambda deps: IOSuccess('>=' if number >= deps else '<'),
...     )
>>> assert first(True).bind(second)(1) == IOSuccess('>=')
>>> assert first(False).bind(second)(2) == IOFailure(-2)

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], KindN[RequiresContextIOResult, TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]] –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]

bind_context_ioresult(function)

Alias for `bind_context_ioresult` method, it is the same as `bind` here.

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], KindN[RequiresContextIOResult, TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]] –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]

bind_result(function)

Binds Result returning function to the current container.

```

>>> from returns.context import RequiresContextIOResult
>>> from returns.result import Success, Failure, Result
>>> from returns.io import IOSuccess, IOFailure

>>> def function(num: int) -> Result[int, str]:
...     return Success(num + 1) if num > 0 else Failure('<0')

>>> assert RequiresContextIOResult.from_value(1).bind_result(
...     function,
... )(RequiresContextIOResult.no_args) == IOSuccess(2)

>>> assert RequiresContextIOResult.from_value(0).bind_result(

```

(continues on next page)

(continued from previous page)

```

...     function,
... )(RequiresContextIOResult.no_args) == IOFailure('<0')

>>> assert RequiresContextIOResult.from_failure(':',').bind_result(
...     function,
... )(RequiresContextIOResult.no_args) == IOFailure(':',')

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)],
Result[TypeVar(_NewValueType), TypeVar(_ErrorType)]] –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_ErrorType),
TypeVar(_EnvType, contravariant=True)]

bind_context(function)

Binds RequiresContext returning function to current container.

```

>>> from returns.context import RequiresContext
>>> from returns.io import IOSuccess, IOFailure

>>> def function(arg: int) -> RequiresContext[int, str]:
...     return RequiresContext(lambda deps: len(deps) + arg)

>>> assert function(2)('abc') == 5

>>> assert RequiresContextIOResult.from_value(2).bind_context(
...     function,
... )('abc') == IOSuccess(5)

>>> assert RequiresContextIOResult.from_failure(2).bind_context(
...     function,
... )('abc') == IOFailure(2)

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)],
RequiresContext[TypeVar(_NewValueType), TypeVar(_EnvType, contravari-
ant=True)]] –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_ErrorType),
TypeVar(_EnvType, contravariant=True)]

bind_context_result(function)

Binds RequiresContextResult returning function to the current one.

```

>>> from returns.context import RequiresContextResult
>>> from returns.io import IOSuccess, IOFailure
>>> from returns.result import Success, Failure

>>> def function(arg: int) -> RequiresContextResult[int, int, str]:
...     if arg > 0:

```

(continues on next page)

(continued from previous page)

```

...     return RequiresContextResult(
...         lambda deps: Success(len(deps) + arg),
...     )
...     return RequiresContextResult(
...         lambda deps: Failure(len(deps) + arg),
...     )

>>> assert function(2)('abc') == Success(5)
>>> assert function(-1)('abc') == Failure(2)

>>> assert RequiresContextIOResult.from_value(
...     2,
... ).bind_context_result(
...     function,
... )('abc') == IOSuccess(5)

>>> assert RequiresContextIOResult.from_value(
...     -1,
... ).bind_context_result(
...     function,
... )('abc') == IOFailure(2)

>>> assert RequiresContextIOResult.from_failure(
...     2,
... ).bind_context_result(
...     function,
... )('abc') == IOFailure(2)

```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_EnvType, contravariant=True)], RequiresContextResult[TypeVar(_NewValueType), TypeVar(_ErrorType)]] – covariant=True),

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]

bind_io(function)

Binds IO returning function to the current container.

```

>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IO, IOSuccess, IOFailure

>>> def function(number: int) -> IO[str]:
...     return IO(str(number))

>>> assert RequiresContextIOResult.from_value(1).bind_io(
...     function,
... )(RequiresContextIOResult.no_args) == IOSuccess('1')

>>> assert RequiresContextIOResult.from_failure(1).bind_io(
...     function,

```

(continues on next page)

(continued from previous page)

```
... )(RequiresContextIOResult.no_args) == IOFailure(1)
```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], IO[TypeVar(_NewValueType)]] –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]

bind_ioresult(*function*)

Binds IOResult returning function to the current container.

```
>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOResult, IOSuccess, IOFailure

>>> def function(num: int) -> IOResult[int, str]:
...     return IOSuccess(num + 1) if num > 0 else IOFailure('<0')

>>> assert RequiresContextIOResult.from_value(1).bind_ioresult(
...     function,
... )(RequiresContextIOResult.no_args) == IOSuccess(2)

>>> assert RequiresContextIOResult.from_value(0).bind_ioresult(
...     function,
... )(RequiresContextIOResult.no_args) == IOFailure('<0')

>>> assert RequiresContextIOResult.from_failure(':', '(').bind_ioresult(
...     function,
... )(RequiresContextIOResult.no_args) == IOFailure(':', '(')
```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], IOResult[TypeVar(_NewValueType), TypeVar(_ErrorType)]] –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]

alt(*function*)

Composes failed container with a pure function.

```
>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess, IOFailure

>>> assert RequiresContextIOResult.from_value(1).alt(
...     lambda x: x + 1,
... )(...) == IOSuccess(1)

>>> assert RequiresContextIOResult.from_failure(1).alt(
...     lambda x: x + 1,
... )(...) == IOFailure(2)
```

Parameters

function (Callable[[TypeVar(_ErrorType)], TypeVar(_NewErrorType)]) –

Return type

RequiresContextIOResult[TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType), TypeVar(_EnvType, contravariant=True)]

lash(function)

Composes this container with a function returning the same type.

```
>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess, IOFailure

>>> def lashable(
...     arg: str,
... ) -> RequiresContextIOResult[str, str, str]:
...     if len(arg) > 1:
...         return RequiresContextIOResult(
...             lambda deps: IOSuccess(deps + arg),
...         )
...     return RequiresContextIOResult(
...         lambda deps: IOFailure(arg + deps),
...     )

>>> assert RequiresContextIOResult.from_value('a').lash(
...     lashable,
... )('c') == IOSuccess('a')
>>> assert RequiresContextIOResult.from_failure('a').lash(
...     lashable,
... )('c') == IOFailure('ac')
>>> assert RequiresContextIOResult.from_failure('aa').lash(
...     lashable,
... )('b') == IOSuccess('baa')
```

Parameters

function (Callable[[TypeVar(_ErrorType)], *KindN*[*RequiresContextIOResult*, TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType), TypeVar(_EnvType, contravariant=True)]] –

Return type

RequiresContextIOResult[TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType), TypeVar(_EnvType, contravariant=True)]

compose_result(function)

Composes inner *Result* with *ReaderIOResult* returning function.

Can be useful when you need an access to both states of the result.

```
>>> from returns.context import ReaderIOResult, NoDeps
>>> from returns.io import IOSuccess, IOFailure
>>> from returns.result import Result

>>> def count(
...     container: Result[int, int],
```

(continues on next page)

(continued from previous page)

```

... ) -> ReaderIOResult[int, int, NoDeps]:
...     return ReaderIOResult.from_result(
...         container.map(lambda x: x + 1).alt(abs),
...     )
>>> success = ReaderIOResult.from_value(1)
>>> failure = ReaderIOResult.from_failure(-1)
>>> assert success.compose_result(count)(...) == IOSuccess(2)
>>> assert failure.compose_result(count)(...) == IOFailure(1)

```

Parameters

function (Callable[[*Result*[TypeVar(_ValueType), covariant=True), TypeVar(_ErrorType)], KindN[RequiresContextIOResult, TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]]]) –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]

modify_env(*function*)

Allows to modify the environment type.

```

>>> from returns.context import RequiresContextIOResultE
>>> from returns.io import IOSuccess, impure_safe

>>> def div(arg: int) -> RequiresContextIOResultE[float, int]:
...     return RequiresContextIOResultE(
...         impure_safe(lambda deps: arg / deps),
...     )

>>> assert div(3).modify_env(int)('2') == IOSuccess(1.5)
>>> assert div(3).modify_env(int)('0').failure()

```

Parameters

function (Callable[[TypeVar(_NewEnvType)], TypeVar(_EnvType, contravariant=True)]] –

Return type

RequiresContextIOResult[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType), TypeVar(_NewEnvType)]

classmethod ask()

Is used to get the current dependencies inside the call stack.

Similar to `returns.context.requires_context.RequiresContext.ask()`, but returns `IOResult` instead of a regular value.

Please, refer to the docs there to learn how to use it.

One important note that is worth duplicating here: you might need to provide `_EnvType` explicitly, so mypy will know about it statically.

```

>>> from returns.context import RequiresContextIOResultE
>>> from returns.io import IOSuccess
>>> assert RequiresContextIOResultE[int, int].ask().map(
...     str,
... )(1) == IOSuccess('1')

```

Return type

RequiresContextIOResult[TypeVar(_EnvType), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True), TypeVar(_ErrorType), TypeVar(_EnvType, contravariant=True)]

classmethod from_result(inner_value)

Creates new container with Result as a unit value.

```

>>> from returns.context import RequiresContextIOResult
>>> from returns.result import Success, Failure
>>> from returns.io import IOSuccess, IOFailure
>>> deps = RequiresContextIOResult.no_args

>>> assert RequiresContextIOResult.from_result(
...     Success(1),
... )(deps) == IOSuccess(1)

>>> assert RequiresContextIOResult.from_result(
...     Failure(1),
... )(deps) == IOFailure(1)

```

Parameters

inner_value (*Result*[TypeVar(_NewValueType), TypeVar(_NewErrorType)]) –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_NewErrorType), Any]

classmethod from_io(inner_value)

Creates new container from successful IO value.

```

>>> from returns.io import IO, IOSuccess
>>> from returns.context import RequiresContextIOResult

>>> assert RequiresContextIOResult.from_io(IO(1))(
...     RequiresContextIOResult.no_args,
... ) == IOSuccess(1)

```

Parameters

inner_value (*IO*[TypeVar(_NewValueType)]) –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), Any, Any]

classmethod from_failed_io(inner_value)

Creates a new container from failed IO value.


```

>>> from returns.io import IO, IOFailure
>>> from returns.context import RequiresContextIOResult

>>> assert RequiresContextIOResult.from_failed_io(IO(1))(
...     RequiresContextIOResult.no_args,
... ) == IOFailure(1)

```

Parameters

inner_value (*IO*[*TypeVar*(*_NewErrorType*))] –

Return type

RequiresContextIOResult[*Any*, *TypeVar*(*_NewErrorType*), *Any*]

classmethod `from_ioresult`(*inner_value*)

Creates new container with *IOResult* as a unit value.

```

>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess, IOFailure
>>> deps = RequiresContextIOResult.no_args

>>> assert RequiresContextIOResult.from_ioresult(
...     IOSuccess(1),
... )(deps) == IOSuccess(1)

>>> assert RequiresContextIOResult.from_ioresult(
...     IOFailure(1),
... )(deps) == IOFailure(1)

```

Parameters

inner_value (*IOResult*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*))] –

Return type

RequiresContextIOResult[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*), *Any*]

classmethod `from_ioresult_context`(*inner_value*)

Creates new container with *ReaderIOResult* as a unit value.

```

>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess, IOFailure

>>> assert RequiresContextIOResult.from_ioresult_context(
...     RequiresContextIOResult.from_value(1),
... )(…) == IOSuccess(1)

>>> assert RequiresContextIOResult.from_ioresult_context(
...     RequiresContextIOResult.from_failure(1),
... )(…) == IOFailure(1)

```

Parameters

inner_value (*RequiresContextIOResult*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*), *TypeVar*(*_NewEnvType*))] –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_NewErrorType), TypeVar(_NewEnvType)]

classmethod from_typecast(*inner_value*)

You might end up with `RequiresContext[IOResult]` as a value.

This method is designed to turn it into `RequiresContextIOResult`. It will save all the typing information.

It is just more useful!

```
>>> from returns.context import RequiresContext
>>> from returns.io import IOSuccess, IOFailure

>>> assert RequiresContextIOResult.from_typecast(
...     RequiresContext.from_value(IOSuccess(1)),
... )(RequiresContextIOResult.no_args) == IOSuccess(1)

>>> assert RequiresContextIOResult.from_typecast(
...     RequiresContext.from_value(IOFailure(1)),
... )(RequiresContextIOResult.no_args) == IOFailure(1)
```

Parameters

inner_value (*RequiresContext*[*IOResult*[TypeVar(_NewValueType), TypeVar(_NewErrorType)], TypeVar(_EnvType, contravariant=True)]) –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_NewErrorType), TypeVar(_EnvType, contravariant=True)]

classmethod from_context(*inner_value*)

Creates new container from `RequiresContext` as a success unit.

```
>>> from returns.context import RequiresContext
>>> from returns.io import IOSuccess

>>> assert RequiresContextIOResult.from_context(
...     RequiresContext.from_value(1),
... )(...) == IOSuccess(1)
```

Parameters

inner_value (*RequiresContext*[TypeVar(_NewValueType), TypeVar(_NewEnvType)]) –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), Any, TypeVar(_NewEnvType)]

classmethod from_failed_context(*inner_value*)

Creates new container from `RequiresContext` as a failure unit.

```
>>> from returns.context import RequiresContext
>>> from returns.io import IOFailure

>>> assert RequiresContextIOResult.from_failed_context(
```

(continues on next page)

(continued from previous page)

```
...     RequiresContext.from_value(1),
... )(...) == IOFailure(1)
```

Parameters

inner_value (*RequiresContext*[TypeVar(_NewValueType), TypeVar(_NewEnvType)]) –

Return type

RequiresContextIOResult[Any, TypeVar(_NewValueType), TypeVar(_NewEnvType)]

classmethod from_result_context(inner_value)

Creates new container from *RequiresContextResult* as a unit value.

```
>>> from returns.context import RequiresContextResult
>>> from returns.io import IOSuccess, IOFailure

>>> assert RequiresContextIOResult.from_result_context(
...     RequiresContextResult.from_value(1),
... )(...) == IOSuccess(1)

>>> assert RequiresContextIOResult.from_result_context(
...     RequiresContextResult.from_failure(1),
... )(...) == IOFailure(1)
```

Parameters

inner_value (*RequiresContextResult*[TypeVar(_NewValueType), TypeVar(_NewErrorType), TypeVar(_NewEnvType)]) –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_NewErrorType), TypeVar(_NewEnvType)]

classmethod from_value(inner_value)

Creates new container with *IOSuccess*(inner_value) as a unit value.

```
>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess

>>> assert RequiresContextIOResult.from_value(1)(
...     RequiresContextIOResult.no_args,
... ) == IOSuccess(1)
```

Parameters

inner_value (TypeVar(_NewValueType)) –

Return type

RequiresContextIOResult[TypeVar(_NewValueType), Any, Any]

classmethod from_failure(inner_value)

Creates new container with *IOFailure*(inner_value) as a unit value.

```

>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOFailure

>>> assert RequiresContextIOResult.from_failure(1)(
...     RequiresContextIOResult.no_args,
... ) == IOFailure(1)

```

Parameters

inner_value (TypeVar(_NewErrorType)) –

Return type

RequiresContextIOResult[Any, TypeVar(_NewErrorType), Any]

RequiresContextIOResultE

Alias for a popular case when Result has Exception as error type.

alias of *RequiresContextIOResult*[_ValueType, Exception, _EnvType]

ReaderIOResult

Alias to save you some typing. Uses original name from Haskell.

Parameters

inner_value (Callable[[TypeVar(_EnvType), *IOResult*[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType)]]], contravariant=True) –

ReaderIOResultE

Alias to save you some typing. Uses Exception as error type.

alias of *RequiresContextIOResult*[_ValueType, Exception, _EnvType]

RequiresContextFutureResult

final class RequiresContextFutureResult(*inner_value*)

Bases: *BaseContainer*, *SupportsKindN[RequiresContextFutureResult, _ValueType, _ErrorType, _EnvType]*, *ReaderFutureResultBasedN[_ValueType, _ErrorType, _EnvType]*, *FutureResultLikeN[_ValueType, _ErrorType, _EnvType]*

The RequiresContextFutureResult combinator.

This probably the main type people are going to use in async programs.

See *returns.context.requires_context.RequiresContext*, *returns.context.requires_context_result.RequiresContextResult*, and *returns.context.requires_context_result.RequiresContextIOResult* for more docs.

This is just a handy wrapper around *RequiresContext*[*FutureResult*[a, b], env] which represents a context-dependent impure async operation that might fail.

So, this is a thin wrapper, without any changes in logic. Why do we need this wrapper? That's just for better usability!

This way *RequiresContextIOResult* allows to simply work with:

- raw values and pure functions
- *RequiresContext* values and pure functions returning it
- *RequiresContextResult* values and pure functions returning it

- `RequiresContextIOResult` values and pure functions returning it
- `Result` and pure functions returning it
- `IOResult` and functions returning it
- `FutureResult` and functions returning it
- other `RequiresContextFutureResult` related functions and values

This is a complex type for complex tasks!

Important implementation details

Due to its meaning, `RequiresContextFutureResult` cannot have `Success` and `Failure` subclasses.

We only have just one type. That's by design.

Different converters are also not supported for this type. Use converters inside the `RequiresContext` context, not outside.

See also:

- <https://dev.to/gcanti/getting-started-with-fp-ts-reader-lie5>
- https://en.wikipedia.org/wiki/Lazy_evaluation
- <https://bit.ly/2R8l4WK>
- <https://bit.ly/2RwP4fp>

Parameters

inner_value (Callable[[TypeVar(_EnvType, contravariant=True)], `FutureResult`[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType, covariant=True)]])

no_args: `ClassVar[NoDeps] = <object object>`

A convenient placeholder to call methods created by `.from_value()`.

swap()

Swaps value and error types.

So, values become errors and errors become values. It is useful when you have to work with errors a lot. And since we have a lot of `.bind_` related methods and only a single `.lash` - it is easier to work with values.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> success = RequiresContextFutureResult.from_value(1)
>>> failure = RequiresContextFutureResult.from_failure(1)

>>> assert anyio.run(success.swap(), ...) == IOFailure(1)
>>> assert anyio.run(failure.swap(), ...) == IOSuccess(1)
```

Return type

`RequiresContextFutureResult`[TypeVar(_ErrorType, covariant=True), TypeVar(_ValueType, covariant=True), TypeVar(_EnvType, contravariant=True)]

map(*function*)

Composes successful container with a pure function.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> assert anyio.run(RequiresContextFutureResult.from_value(1).map(
...     lambda x: x + 1,
... )(...).awaitable) == IOSuccess(2)

>>> assert anyio.run(RequiresContextFutureResult.from_failure(1).map(
...     lambda x: x + 1,
... )(...).awaitable) == IOFailure(1)
```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)]], covariant=True) –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType), covariant=True], TypeVar(_EnvType, contravariant=True)]

apply(*container*)

Calls a wrapped function in a container on this container.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> def transform(arg: str) -> str:
...     return arg + 'b'

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value('a').apply(
...         RequiresContextFutureResult.from_value(transform),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess('ab')

>>> assert anyio.run(
...     RequiresContextFutureResult.from_failure('a').apply(
...         RequiresContextFutureResult.from_value(transform),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure('a')
```

Parameters

container (*KindN*[*RequiresContextFutureResult*, Callable[[TypeVar(_ValueType), TypeVar(_NewValueType)], TypeVar(_ErrorType), covariant=True)], TypeVar(_EnvType, contravariant=True)) –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType),

```
covariant=True), TypeVar(_EnvType, contravariant=True)]
```

bind(function)

Composes this container with a function returning the same type.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> def function(
...     number: int,
... ) -> RequiresContextFutureResult[str, int, int]:
...     # `deps` has `int` type here:
...     return RequiresContextFutureResult(
...         lambda deps: FutureResult.from_value(str(number + deps)),
...     )

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(2).bind(function),
...     3,
... ) == IOSuccess('5')
>>> assert anyio.run(
...     RequiresContextFutureResult.from_failure(2).bind(function),
...     3,
... ) == IOFailure(2)
```

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)], KindN[RequiresContextFutureResult, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]])

–

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_context_future_result(function)

Alias for `bind_context_future_result` method, it is the same as `bind` here.

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)], KindN[RequiresContextFutureResult, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]])

–

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_async(function)

Composes this container with a `async` function returning the same type.

```

>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> async def function(
...     number: int,
... ) -> RequiresContextFutureResult[str, int, int]:
...     return RequiresContextFutureResult.from_value(number + 1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(1).bind_async(
...         function,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(2)
>>> assert anyio.run(
...     RequiresContextFutureResult.from_failure(1).bind_async(
...         function,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], Awaitable[KindN[RequiresContextFutureResult, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]]]) –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_async_context_future_result(function)

Alias for *bind_async_context_future_result* method, it is the same as *bind_async* here.

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], Awaitable[KindN[RequiresContextFutureResult, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]]]) –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_awaitable(function)

Allows to compose a container and a regular async function.

This function should return plain, non-container value. See *bind_async()* to bind async function that returns a container.

```

>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> async def coroutine(x: int) -> int:

```

(continues on next page)

(continued from previous page)

```

...     return x + 1
>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(1).bind_awaitable(
...         coroutine,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(2)
>>> assert anyio.run(
...     RequiresContextFutureResult.from_failure(1).bind_awaitable(
...         coroutine,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)

```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_EnvType), TypeVar(_ErrorType)], Awaitable[TypeVar(_NewValueType)]] – covariant=True),

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType), TypeVar(_EnvType), covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_result(function)

Binds *Result* returning function to the current container.

```

>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.result import Success, Result
>>> from returns.io import IOSuccess, IOFailure
>>> def function(num: int) -> Result[int, str]:
...     return Success(num + 1)
>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(1).bind_result(
...         function,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(2)
>>> assert anyio.run(
...     RequiresContextFutureResult.from_failure(':(').bind_result(
...         function,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(':(')

```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_EnvType), TypeVar(_ErrorType)], *Result*[TypeVar(_NewValueType), TypeVar(_ErrorType), covariant=True]] – covariant=True),

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_context(function)

Binds *RequiresContext* returning function to current container.

```
>>> import anyio
>>> from returns.context import RequiresContext
>>> from returns.io import IOSuccess, IOFailure

>>> def function(arg: int) -> RequiresContext[int, str]:
...     return RequiresContext(lambda deps: len(deps) + arg)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(2).bind_context(
...         function,
...     ),
...     'abc',
... ) == IOSuccess(5)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_failure(0).bind_context(
...         function,
...     ),
...     'abc',
... ) == IOFailure(0)
```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], *RequiresContext*[TypeVar(_NewValueType), TypeVar(_EnvType, contravariant=True)]])

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_context_result(function)

Binds *RequiresContextResult* returning function to the current one.

```
>>> import anyio
>>> from returns.context import RequiresContextResult
>>> from returns.io import IOSuccess, IOFailure
>>> from returns.result import Success

>>> def function(arg: int) -> RequiresContextResult[int, int, str]:
...     return RequiresContextResult(
...         lambda deps: Success(len(deps) + arg),
...     )

>>> instance = RequiresContextFutureResult.from_value(
...     2,
... ).bind_context_result(
```

(continues on next page)

(continued from previous page)

```

...     function,
...   )('abc')
>>> assert anyio.run(instance.awaitable) == IOSuccess(5)

>>> instance = RequiresContextFutureResult.from_failure(
...     2,
...   ).bind_context_result(
...     function,
...   )('abc')
>>> assert anyio.run(instance.awaitable) == IOFailure(2)

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], RequiresContextResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]] –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_context_ioresult(function)

Binds RequiresContextIOResult returning function to the current one.

```

>>> import anyio
>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess, IOFailure

>>> def function(arg: int) -> RequiresContextIOResult[int, int, str]:
...     return RequiresContextIOResult(
...         lambda deps: IOSuccess(len(deps) + arg),
...     )

>>> instance = RequiresContextFutureResult.from_value(
...     2,
...   ).bind_context_ioresult(
...     function,
...   )('abc')
>>> assert anyio.run(instance.awaitable) == IOSuccess(5)

>>> instance = RequiresContextFutureResult.from_failure(
...     2,
...   ).bind_context_ioresult(
...     function,
...   )('abc')
>>> assert anyio.run(instance.awaitable) == IOFailure(2)

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], RequiresContextIOResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]] –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType,

```
covariant=True), TypeVar(_EnvType, contravariant=True)]
```

bind_io(*function*)

Binds IO returning function to the current container.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IO, IOSuccess, IOFailure

>>> def do_io(number: int) -> IO[str]:
...     return IO(str(number)) # not IO operation actually

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(1).bind_io(do_io),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess('1')

>>> assert anyio.run(
...     RequiresContextFutureResult.from_failure(1).bind_io(do_io),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)
```

Parameters

function (Callable[[TypeVar(_ValueType), TypeVar(_EnvType), covariant=True)], IO[TypeVar(_NewValueType)]) –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType), covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_ioresult(*function*)

Binds IOResult returning function to the current container.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOResult, IOSuccess, IOFailure

>>> def function(num: int) -> IOResult[int, str]:
...     return IOSuccess(num + 1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(1).bind_ioresult(
...         function,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(2)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_failure(':',').bind_ioresult(
...         function,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(':',')
```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)],
IOResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]])

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType,
covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_future(function)

Binds Future returning function to the current container.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.future import Future
>>> from returns.io import IOSuccess, IOFailure

>>> def function(num: int) -> Future[int]:
...     return Future.from_value(num + 1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(1).bind_future(
...         function,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(2)

>>> failed = RequiresContextFutureResult.from_failure(':(')
>>> assert anyio.run(
...     failed.bind_future(function),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(':(')
```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)],
Future[TypeVar(_NewValueType)]])

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType,
covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_future_result(function)

Binds FutureResult returning function to the current container.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> def function(num: int) -> FutureResult[int, str]:
...     return FutureResult.from_value(num + 1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(1).bind_future_result(
...         function,
```

(continues on next page)

(continued from previous page)

```

...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(2)

>>> failed = RequiresContextFutureResult.from_failure(':(')
>>> assert anyio.run(
...     failed.bind_future_result(function),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(':(')

```

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)],
FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]] –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType,
covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_async_future(*function*)

Binds Future returning async function to the current container.

```

>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.future import Future
>>> from returns.io import IOSuccess, IOFailure

>>> async def function(num: int) -> Future[int]:
...     return Future.from_value(num + 1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(1).bind_async_future(
...         function,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(2)

>>> failed = RequiresContextFutureResult.from_failure(':(')
>>> assert anyio.run(
...     failed.bind_async_future(function),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(':(')

```

Parameters

function (Callable[[TypeVar(_ValueType), covariant=True)],
Awaitable[*Future*[TypeVar(_NewValueType)]]] –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType,
covariant=True), TypeVar(_EnvType, contravariant=True)]

bind_async_future_result(*function*)

Bind *FutureResult* returning async function to the current container.

```

>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> async def function(num: int) -> FutureResult[int, str]:
...     return FutureResult.from_value(num + 1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(
...         1,
...     ).bind_async_future_result(
...         function,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(2)

>>> failed = RequiresContextFutureResult.from_failure(':(')
>>> assert anyio.run(
...     failed.bind_async_future_result(function),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(':(')

```

Parameters

function (Callable[[TypeVar(_ValueType, covariant=True)], Awaitable[FutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True)]]]) –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

alt(function)

Composes failed container with a pure function.

```

>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value(1).alt(
...         lambda x: x + 1,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_failure(1).alt(
...         lambda x: x + 1,
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(2)

```

Parameters

function (Callable[[TypeVar(_ErrorType, covariant=True), TypeVar(_NewErrorType)]] –

Return type

RequiresContextFutureResult[TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType), TypeVar(_EnvType, contravariant=True)]

lash(function)

Composes this container with a function returning the same type.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess

>>> def lashable(
...     arg: str,
... ) -> RequiresContextFutureResult[str, str, str]:
...     return RequiresContextFutureResult(
...         lambda deps: FutureResult.from_value(
...             deps + arg,
...         ),
...     )

>>> assert anyio.run(
...     RequiresContextFutureResult.from_value('a').lash(lashable),
...     'c',
... ) == IOSuccess('a')

>>> assert anyio.run(
...     RequiresContextFutureResult.from_failure('aa').lash(
...         lashable,
...     ),
...     'b',
... ) == IOSuccess('baa')
```

Parameters

function (Callable[[TypeVar(_ErrorType, covariant=True), *KindN*[*RequiresContextFutureResult*, TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType), TypeVar(_EnvType, contravariant=True)]]] –

Return type

RequiresContextFutureResult[TypeVar(_ValueType, covariant=True), TypeVar(_NewErrorType), TypeVar(_EnvType, contravariant=True)]

compose_result(function)

Composes inner Result with ReaderFutureResult returning func.

Can be useful when you need an access to both states of the result.

```
>>> import anyio
>>> from returns.context import ReaderFutureResult, NoDeps
>>> from returns.io import IOSuccess, IOFailure
```

(continues on next page)

(continued from previous page)

```

>>> from returns.result import Result

>>> def count(
...     container: Result[int, int],
... ) -> ReaderFutureResult[int, int, NoDeps]:
...     return ReaderFutureResult.from_result(
...         container.map(lambda x: x + 1).alt(abs),
...     )

>>> success = ReaderFutureResult.from_value(1)
>>> failure = ReaderFutureResult.from_failure(-1)

>>> assert anyio.run(
...     success.compose_result(count), ReaderFutureResult.no_args,
... ) == IOSuccess(2)
>>> assert anyio.run(
...     failure.compose_result(count), ReaderFutureResult.no_args,
... ) == IOFailure(1)

```

Parameters

function (Callable[[*Result*[TypeVar(_ValueType), covariant=True), TypeVar(_ErrorType, covariant=True)], *KindN*[*RequiresContextFutureResult*, TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]] –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_ErrorType, covariant=True), TypeVar(_EnvType, contravariant=True)]

modify_env(*function*)

Allows to modify the environment type.

```

>>> import anyio
>>> from returns.future import future_safe, asyncify
>>> from returns.context import RequiresContextFutureResultE
>>> from returns.io import IOSuccess

>>> def div(arg: int) -> RequiresContextFutureResultE[float, int]:
...     return RequiresContextFutureResultE(
...         future_safe(asyncify(lambda deps: arg / deps)),
...     )

>>> assert anyio.run(div(3).modify_env(int), '2') == IOSuccess(1.5)
>>> assert anyio.run(div(3).modify_env(int), '0').failure()

```

Parameters

function (Callable[[TypeVar(_NewEnvType)], TypeVar(_EnvType, contravariant=True)]] –

Return type

RequiresContextFutureResult[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType, covariant=True), TypeVar(_NewEnvType)]

classmethod ask()

Is used to get the current dependencies inside the call stack.

Similar to `returns.context.requires_context.RequiresContext.ask()`, but returns `FutureResult` instead of a regular value.

Please, refer to the docs there to learn how to use it.

One important note that is worth duplicating here: you might need to provide type annotations explicitly, so mypy will know about it statically.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResultE
>>> from returns.io import IOSuccess

>>> assert anyio.run(
...     RequiresContextFutureResultE[int, int].ask().map(str),
...     1,
... ) == IOSuccess('1')
```

Return type

`RequiresContextFutureResult`[`TypeVar`(`_EnvType`, `contravariant=True`),
`TypeVar`(`_ErrorType`, `covariant=True`), `TypeVar`(`_EnvType`, `contravariant=True`)]

classmethod from_result(inner_value)

Creates new container with `Result` as a unit value.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.result import Success, Failure
>>> from returns.io import IOSuccess, IOFailure

>>> assert anyio.run(
...     RequiresContextFutureResult.from_result(Success(1)),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_result(Failure(1)),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)
```

Parameters

inner_value (`Result`[`TypeVar`(`_NewValueType`), `TypeVar`(`_NewErrorType`)])

Return type

`RequiresContextFutureResult`[`TypeVar`(`_NewValueType`),
`TypeVar`(`_NewErrorType`), `Any`]

classmethod from_io(inner_value)

Creates new container from successful IO value.

```
>>> import anyio
>>> from returns.io import IO, IOSuccess
```

(continues on next page)

(continued from previous page)

```
>>> from returns.context import RequiresContextFutureResult
>>> assert anyio.run(
...     RequiresContextFutureResult.from_io(IO(1)),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(1)
```

Parameters**inner_value** (*IO*[*TypeVar*(*_NewValueType*)]) –**Return type***RequiresContextFutureResult*[*TypeVar*(*_NewValueType*), *Any*, *Any*]**classmethod from_failed_io**(*inner_value*)

Creates a new container from failed IO value.

```
>>> import anyio
>>> from returns.io import IO, IOFailure
>>> from returns.context import RequiresContextFutureResult
>>> assert anyio.run(
...     RequiresContextFutureResult.from_failed_io(IO(1)),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)
```

Parameters**inner_value** (*IO*[*TypeVar*(*_NewErrorType*)]) –**Return type***RequiresContextFutureResult*[*Any*, *TypeVar*(*_NewErrorType*), *Any*]**classmethod from_ioresult**(*inner_value*)

Creates new container with IOResult as a unit value.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOSuccess, IOFailure
>>> assert anyio.run(
...     RequiresContextFutureResult.from_ioresult(IOSuccess(1)),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(1)
>>> assert anyio.run(
...     RequiresContextFutureResult.from_ioresult(IOFailure(1)),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)
```

Parameters**inner_value** (*IOResult*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*)]) –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType),
TypeVar(_NewErrorType), Any]

classmethod from_future(*inner_value*)

Creates new container with successful Future as a unit value.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.future import Future
>>> from returns.io import IOSuccess

>>> assert anyio.run(
...     RequiresContextFutureResult.from_future(Future.from_value(1)),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(1)
```

Parameters

inner_value (*Future*[TypeVar(_NewValueType)]) –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), Any, Any]

classmethod from_failed_future(*inner_value*)

Creates new container with failed Future as a unit value.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.future import Future
>>> from returns.io import IOFailure

>>> assert anyio.run(
...     RequiresContextFutureResult.from_failed_future(
...         Future.from_value(1),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)
```

Parameters

inner_value (*Future*[TypeVar(_NewErrorType)]) –

Return type

RequiresContextFutureResult[Any, TypeVar(_NewErrorType), Any]

classmethod from_future_result_context(*inner_value*)

Creates new container with ReaderFutureResult as a unit value.

```
>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> assert anyio.run(
...     RequiresContextFutureResult.from_future_result_context(
```

(continues on next page)

(continued from previous page)

```

...     RequiresContextFutureResult.from_value(1),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_future_result_context(
...         RequiresContextFutureResult.from_failure(1),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)

```

Parameters

inner_value (*RequiresContextFutureResult*[TypeVar(_NewValueType), TypeVar(_NewErrorType), TypeVar(_NewEnvType)]) –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_NewErrorType), TypeVar(_NewEnvType)]

classmethod from_future_result(inner_value)

Creates new container with FutureResult as a unit value.

```

>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> assert anyio.run(
...     RequiresContextFutureResult.from_future_result(
...         FutureResult.from_value(1),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_future_result(
...         FutureResult.from_failure(1),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)

```

Parameters

inner_value (*FutureResult*[TypeVar(_NewValueType), TypeVar(_NewErrorType)]) –

Return type

RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_NewErrorType), Any]

classmethod from_typecast(inner_value)

You might end up with `RequiresContext[FutureResult]` as a value.

This method is designed to turn it into `RequiresContextFutureResult`. It will save all the typing information.

It is just more useful!

```
>>> import anyio
>>> from returns.context import RequiresContext
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> assert anyio.run(
...     RequiresContextFutureResult.from_typecast(
...         RequiresContext.from_value(FutureResult.from_value(1)),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_typecast(
...         RequiresContext.from_value(FutureResult.from_failure(1)),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)
```

Parameters

inner_value (`RequiresContext[FutureResult[TypeVar(_NewValueType), TypeVar(_NewErrorType)], TypeVar(_EnvType, contravariant=True)]`) –

Return type

`RequiresContextFutureResult[TypeVar(_NewValueType), TypeVar(_NewErrorType), TypeVar(_EnvType, contravariant=True)]`

classmethod `from_context(inner_value)`

Creates new container from `RequiresContext` as a success unit.

```
>>> import anyio
>>> from returns.context import RequiresContext
>>> from returns.io import IOSuccess

>>> assert anyio.run(
...     RequiresContextFutureResult.from_context(
...         RequiresContext.from_value(1),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(1)
```

Parameters

inner_value (`RequiresContext[TypeVar(_NewValueType), TypeVar(_NewEnvType)]`) –

Return type

`RequiresContextFutureResult[TypeVar(_NewValueType), Any, TypeVar(_NewEnvType)]`

classmethod `from_failed_context`(*inner_value*)

Creates new container from `RequiresContext` as a failure unit.

```
>>> import anyio
>>> from returns.context import RequiresContext
>>> from returns.io import IOFailure

>>> assert anyio.run(
...     RequiresContextFutureResult.from_failed_context(
...         RequiresContext.from_value(1),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)
```

Parameters

inner_value (`RequiresContext`[`TypeVar`(`_NewValueType`), `TypeVar`(`_NewEnvType`)]) –

Return type

`RequiresContextFutureResult`[`Any`, `TypeVar`(`_NewValueType`), `TypeVar`(`_NewEnvType`)]

classmethod `from_result_context`(*inner_value*)

Creates new container from `RequiresContextResult` as a unit value.

```
>>> import anyio
>>> from returns.context import RequiresContextResult
>>> from returns.io import IOSuccess, IOFailure

>>> assert anyio.run(
...     RequiresContextFutureResult.from_result_context(
...         RequiresContextResult.from_value(1),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_result_context(
...         RequiresContextResult.from_failure(1),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)
```

Parameters

inner_value (`RequiresContextResult`[`TypeVar`(`_NewValueType`), `TypeVar`(`_NewErrorType`), `TypeVar`(`_NewEnvType`)]) –

Return type

`RequiresContextFutureResult`[`TypeVar`(`_NewValueType`), `TypeVar`(`_NewErrorType`), `TypeVar`(`_NewEnvType`)]

classmethod `from_ioresult_context`(*inner_value*)

Creates new container from `RequiresContextIOResult` as a unit value.

```

>>> import anyio
>>> from returns.context import RequiresContextIOResult
>>> from returns.io import IOSuccess, IOFailure

>>> assert anyio.run(
...     RequiresContextFutureResult.from_ioresult_context(
...         RequiresContextIOResult.from_value(1),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOSuccess(1)

>>> assert anyio.run(
...     RequiresContextFutureResult.from_ioresult_context(
...         RequiresContextIOResult.from_failure(1),
...     ),
...     RequiresContextFutureResult.no_args,
... ) == IOFailure(1)

```

Parameters

inner_value (*RequiresContextIOResult*[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*), *TypeVar*(*_NewEnvType*)]) –

Return type

RequiresContextFutureResult[*TypeVar*(*_NewValueType*), *TypeVar*(*_NewErrorType*), *TypeVar*(*_NewEnvType*)]

classmethod from_value(*inner_value*)

Creates new container with successful *FutureResult* as a unit value.

```

>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOSuccess

>>> assert anyio.run(RequiresContextFutureResult.from_value(1)(
...     RequiresContextFutureResult.no_args,
... ).awaitable) == IOSuccess(1)

```

Parameters

inner_value (*TypeVar*(*_FirstType*)) –

Return type

RequiresContextFutureResult[*TypeVar*(*_FirstType*), *Any*, *Any*]

classmethod from_failure(*inner_value*)

Creates new container with failed *FutureResult* as a unit value.

```

>>> import anyio
>>> from returns.context import RequiresContextFutureResult
>>> from returns.io import IOFailure

>>> assert anyio.run(RequiresContextFutureResult.from_failure(1)(
...     RequiresContextFutureResult.no_args,
... ).awaitable) == IOFailure(1)

```


Parameters**inner_value** (TypeVar(_FirstType)) –**Return type***RequiresContextFutureResult*[Any, TypeVar(_FirstType), Any]**RequiresContextFutureResultE**Alias for a popular case when *Result* has *Exception* as error type.alias of *RequiresContextFutureResult*[_ValueType, *Exception*, _EnvType]**ReaderFutureResult**Sometimes *RequiresContextFutureResult* is too long to type.**Parameters****inner_value** (Callable[[TypeVar(_EnvType, contravariant=True)], *FutureResult*[TypeVar(_ValueType, covariant=True), TypeVar(_ErrorType, covariant=True)]] –**ReaderFutureResultE**Alias to save you some typing. Uses *Exception* as error type.alias of *RequiresContextFutureResult*[_ValueType, *Exception*, _EnvType]

9.11 Create your own container

This tutorial will guide you through the process of creating your own containers.

9.11.1 Step 0: Motivation

First things first, why would anyone want to create a custom containers?

The great idea about “containers” in functional programming is that it can be literally anything. There are endless use-cases.

You can create your own primitives for working with some language-or-framework specific problem, or just model your business domain.

You can copy ideas from other languages or just compose existing containers for better usability (like *IOResult* is the composition of *IO* and *Result*).

Example

We are going to implement a *Pair* container for this example. What is a *Pair*? Well, it is literally a pair of two values. No more, no less. Similar to a *Tuple*[*FirstType*, *SecondType*]. But with extra goodies.

Note: You can find all [code samples here](#).

9.11.2 Step 1: Choosing right interfaces

After you came up with the idea, you will need to make a decision: what capabilities my container must have?

Basically, you should decide what *Interfaces* you will subtype and what methods and laws will be present in your type. You can create just a `returns.interfaces.mappable.MappableN` or choose a full featured `returns.interfaces.container.ContainerN`.

You can also choose some specific interfaces to use, like `returns.interfaces.specific.result.ResultLikeN` or any other.

Summing up, decide what laws and methods you need to solve your problem. And then subtype the interfaces that provide these methods and laws.

Example

What interfaces a Pair type needs?

- `returns.interfaces.equable.Equable`, because two Pair instances can be compared
- `returns.interfaces.mappable.MappableN`, because the first type can be composed with pure functions
- `returns.interfaces.bindable.BindableN`, because a Pair can be bound to a function returning a new Pair based on the first type
- `returns.interfaces.altable.AltableN`, because the second type can be composed with pure functions
- `returns.interfaces.lashable.LashableN`, because a Pair can be bound to a function returning a new Pair based on the second type

Now, after we know about all interfaces we would need, let's find pre-defined aliases we can reuse.

Turns out, there are some of them!

- `returns.interfaces.bimappable.BiMappableN` which combines `MappableN` and `AltableN`
- `returns.interfaces.swappable.SwappableN` is an alias for `BiMappableN` with a new method called `swap` to change values order

Let's look at the result:

Note: A special note on `returns.primitives.container.BaseContainer`. It is a very useful class with lots of pre-defined features, like: immutability, better cloning, serialization, and comparison.

You can skip it if you wish, but it is highly recommended.

Later we will talk about an actual implementation of all required methods.

9.11.3 Step 2: Initial implementation

So, let's start writing some code!

We would need to implement all interface methods, otherwise mypy won't be happy. That's what it currently says on our type definition:

```
error: Final class test_pair1.Pair has abstract attributes "alt", "bind", "equals", "lash  
↪", "map", "swap"
```

Looks like it already knows what methods should be there!

Ok, let's drop some initial and straight forward implementation. We will later make it more complex step by step.

```

1  from typing import Callable, Tuple, TypeVar, final
2
3  from returns.interfaces import bindable, equable, lashable, swappable
4  from returns.primitives.container import BaseContainer, container_equality
5  from returns.primitives.hkt import Kind2, SupportsKind2, dekind
6
7  _FirstType = TypeVar('_FirstType')
8  _SecondType = TypeVar('_SecondType')
9
10 _NewFirstType = TypeVar('_NewFirstType')
11 _NewSecondType = TypeVar('_NewSecondType')
12
13
14 @final
15 class Pair(
16     BaseContainer,
17     SupportsKind2['Pair', _FirstType, _SecondType],
18     bindable.Bindable2[_FirstType, _SecondType],
19     swappable.Swappable2[_FirstType, _SecondType],
20     lashable.Lashable2[_FirstType, _SecondType],
21     equable.Equable,
22 ):
23     """
24     A type that represents a pair of something.
25
26     Like to coordinates ``(x, y)`` or two best friends.
27     Or a question and an answer.
28
29     """
30
31     def __init__(
32         self,
33         inner_value: Tuple[_FirstType, _SecondType],
34     ) -> None:
35         """Saves passed tuple as ``._inner_value`` inside this instance."""
36         super().__init__(inner_value)
37
38     # `Equable` part:
39
40     equals = container_equality # we already have this defined for all types
41
42     # `Mappable` part via `BiMappable`:
43
44     def map(
45         self,
46         function: Callable[[_FirstType], _NewFirstType],
47     ) -> 'Pair[_NewFirstType, _SecondType]':
48         return Pair((function(self._inner_value[0]), self._inner_value[1]))
49
50     # `BindableN` part:

```

(continues on next page)

```

51
52 def bind(
53     self,
54     function: Callable[
55         [_FirstType],
56         Kind2['Pair', _NewFirstType, _SecondType],
57     ],
58 ) -> 'Pair[_NewFirstType, _SecondType]':
59     return dekind(function(self._inner_value[0]))
60
61 # `AltableN` part via `BiMappableN`:
62
63 def alt(
64     self,
65     function: Callable[_SecondType, _NewSecondType],
66 ) -> 'Pair[_FirstType, _NewSecondType]':
67     return Pair((self._inner_value[0], function(self._inner_value[1])))
68
69 # `LashableN` part:
70
71 def lash(
72     self,
73     function: Callable[
74         [_SecondType],
75         Kind2['Pair', _FirstType, _NewSecondType],
76     ],
77 ) -> 'Pair[_FirstType, _NewSecondType]':
78     return dekind(function(self._inner_value[1]))
79
80 # `SwappableN` part:
81
82 def swap(self) -> 'Pair[_SecondType, _FirstType]':
83     return Pair((self._inner_value[1], self._inner_value[0]))

```

You can check our resulting source with mypy. It would be happy this time.

9.11.4 Step 3: New interfaces

As you can see our existing interfaces do not cover everything. We can potentially want several extra things:

1. A method that takes two arguments and returns a new `Pair` instance
2. A named constructor to create a `Pair` from a single value
3. A named constructor to create a `Pair` from two values

We can define an interface just for this! It would be also nice to add all other interfaces there as supertypes.

That's how it is going to look:

```

1 class PairLikeN(
2     bindable.BindableN[_FirstType, _SecondType, _ThirdType],
3     swappable.SwappableN[_FirstType, _SecondType, _ThirdType],
4     lashable.LashableN[_FirstType, _SecondType, _ThirdType],

```

(continues on next page)

(continued from previous page)

```

5     equable.Equable,
6 ):
7     """Special interface for types that look like a ``Pair``."""
8
9     @abstractmethod
10    def pair(
11        self: _PairLikeKind,
12        function: Callable[
13            [_FirstType, _SecondType],
14            KindN[_PairLikeKind, _NewFirstType, _NewSecondType, _ThirdType],
15        ],
16    ) -> KindN[_PairLikeKind, _NewFirstType, _NewSecondType, _ThirdType]:
17        """Allows to work with both arguments at the same time."""
18
19    @classmethod
20    @abstractmethod
21    def from_paired(
22        cls: Type[_PairLikeKind],
23        first: _NewFirstType,
24        second: _NewSecondType,
25    ) -> KindN[_PairLikeKind, _NewFirstType, _NewSecondType, _ThirdType]:
26        """Allows to create a PairLikeN from just two values."""
27
28    @classmethod
29    @abstractmethod
30    def from_unpaired(
31        cls: Type[_PairLikeKind],
32        inner_value: _NewFirstType,
33    ) -> KindN[_PairLikeKind, _NewFirstType, _NewFirstType, _ThirdType]:
34        """Allows to create a PairLikeN from just a single object."""

```

Awesome! Now we have a new interface to implement. Let's do that!

```

1     def pair(
2         self,
3         function: Callable[
4             [_FirstType, _SecondType],
5             Kind2['Pair', _NewFirstType, _NewSecondType],
6         ],
7     ) -> 'Pair[_NewFirstType, _NewSecondType]':
8         return dekind(function(self._inner_value[0], self._inner_value[1]))

```

```

1     @classmethod
2     def from_unpaired(
3         cls,
4         inner_value: _NewFirstType,
5     ) -> 'Pair[_NewFirstType, _NewFirstType]':
6         return Pair((inner_value, inner_value))

```

Looks like we are done!

9.11.5 Step 4: Writing tests and docs

The best part about this type is that it is pure. So, we can write our tests inside docs!

We are going to use `doctests` builtin module for that.

This gives us several key benefits:

- All our docs has usage examples
- All our examples are correct, because they are executed and tested
- We don't need to write regular boring tests

Let's add docs and doctests! Let's use `map` method as a short example:

```

1  def map(
2      self,
3      function: Callable[_FirstType, _NewFirstType],
4  ) -> 'Pair[_NewFirstType, _SecondType]':
5      """
6      Changes the first type with a pure function.
7
8      >>> assert Pair((1, 2)).map(str) == Pair('1', 2)
9
10     """
11     return Pair((function(self._inner_value[0]), self._inner_value[1]))

```

By adding these simple tests we would already have 100% coverage. But, what if we can completely skip writing tests, but still have 100%?

Let's discuss how we can achieve that with "Laws as values".

9.11.6 Step 5: Checking laws

We already ship lots of laws with our interfaces. See our docs on *laws and checking them*.

Moreover, you can also define your own laws! Let's add them to our `PairLikeN` interface.

Let's start with laws definition:

```

1  class _LawSpec(LawSpecDef):
2      @law_definition
3      def pair_equality_law(
4          raw_value: _FirstType,
5          container: 'PairLikeN[_FirstType, _SecondType, _ThirdType]',
6      ) -> None:
7          """Ensures that unpaired and paired constructors work fine."""
8          assert_equal(
9              container.from_unpaired(raw_value),
10             container.from_paired(raw_value, raw_value),
11         )
12
13     @law_definition
14     def pair_left_identity_law(
15         pair: Tuple[_FirstType, _SecondType],
16         container: 'PairLikeN[_FirstType, _SecondType, _ThirdType]',

```

(continues on next page)

(continued from previous page)

```

17     function: Callable[
18         [_FirstType, _SecondType],
19         KindN['PairLikeN', _NewFirstType, _NewSecondType, _ThirdType],
20     ],
21 ) -> None:
22     """Ensures that unpaired and paired constructors work fine."""
23     assert_equal(
24         container.from_paired(*pair).pair(function),
25         function(*pair),
26     )

```

And then let's add them to our PairLikeN interface:

```

1  class PairLikeN(
2      bindable.BindableN[_FirstType, _SecondType, _ThirdType],
3      swappable.SwappableN[_FirstType, _SecondType, _ThirdType],
4      lashable.LashableN[_FirstType, _SecondType, _ThirdType],
5      equable.Equable,
6  ):
7      """Special interface for types that look like a `Pair`."""
8
9      _laws: ClassVar[Sequence[Law]] = (
10         Law2(_LawSpec.pair_equality_law),
11         Law3(_LawSpec.pair_left_identity_law),
12     )
13
14     @abstractmethod
15     def pair(
16         self: _PairLikeKind,
17         function: Callable[
18             [_FirstType, _SecondType],
19             KindN[_PairLikeKind, _NewFirstType, _NewSecondType, _ThirdType],
20         ],
21     ) -> KindN[_PairLikeKind, _NewFirstType, _NewSecondType, _ThirdType]:
22         """Allows to work with both arguments at the same time."""
23
24     @classmethod
25     @abstractmethod
26     def from_paired(
27         cls: Type[_PairLikeKind],
28         first: _NewFirstType,
29         second: _NewSecondType,
30     ) -> KindN[_PairLikeKind, _NewFirstType, _NewSecondType, _ThirdType]:
31         """Allows to create a PairLikeN from just two values."""
32
33     @classmethod
34     @abstractmethod
35     def from_unpaired(
36         cls: Type[_PairLikeKind],
37         inner_value: _NewFirstType,
38     ) -> KindN[_PairLikeKind, _NewFirstType, _NewFirstType, _ThirdType]:
39         """Allows to create a PairLikeN from just a single object."""

```

The last to do is to call `check_all_laws(Pair, use_init=True)` to generate 10 hypothesis test cases with hundreds real test cases inside.

Here's the final result of our brand new `Pair` type:

```

1  from abc import abstractmethod
2  from typing import (
3      Callable,
4      ClassVar,
5      NoReturn,
6      Sequence,
7      Tuple,
8      Type,
9      TypeVar,
10     final,
11 )
12
13 from returns.contrib.hypothesis.laws import check_all_laws
14 from returns.interfaces import bindable, equable, lashable, swappable
15 from returns.primitives.asserts import assert_equal
16 from returns.primitives.container import BaseContainer, container_equality
17 from returns.primitives.hkt import Kind2, KindN, SupportsKind2, dekind
18 from returns.primitives.laws import Law, Law2, Law3, LawSpecDef, law_definition
19
20 _FirstType = TypeVar('_FirstType')
21 _SecondType = TypeVar('_SecondType')
22 _ThirdType = TypeVar('_ThirdType')
23
24 _NewFirstType = TypeVar('_NewFirstType')
25 _NewSecondType = TypeVar('_NewSecondType')
26
27 _PairLikeKind = TypeVar('_PairLikeKind', bound='PairLikeN')
28
29
30 class _LawSpec(LawSpecDef):
31     @law_definition
32     def pair_equality_law(
33         raw_value: _FirstType,
34         container: 'PairLikeN[_FirstType, _SecondType, _ThirdType]',
35     ) -> None:
36         """Ensures that unpaired and paired constructors work fine."""
37         assert_equal(
38             container.from_unpaired(raw_value),
39             container.from_paired(raw_value, raw_value),
40         )
41
42     @law_definition
43     def pair_left_identity_law(
44         pair: Tuple[_FirstType, _SecondType],
45         container: 'PairLikeN[_FirstType, _SecondType, _ThirdType]',
46         function: Callable[
47             [_FirstType, _SecondType],
48             KindN['PairLikeN', _NewFirstType, _NewSecondType, _ThirdType],
49         ],

```

(continues on next page)

(continued from previous page)

```

50 ) -> None:
51     """Ensures that unpaired and paired constructors work fine."""
52     assert_equal(
53         container.from_paired(*pair).pair(function),
54         function(*pair),
55     )
56
57
58 class PairLikeN(
59     bindable.BindableN[_FirstType, _SecondType, _ThirdType],
60     swappable.SwappableN[_FirstType, _SecondType, _ThirdType],
61     lashable.LashableN[_FirstType, _SecondType, _ThirdType],
62     equable.Equable,
63 ):
64     """Special interface for types that look like a `Pair`."""
65
66     _laws: ClassVar[Sequence[Law]] = (
67         Law2(_LawSpec.pair_equality_law),
68         Law3(_LawSpec.pair_left_identity_law),
69     )
70
71     @abstractmethod
72     def pair(
73         self: _PairLikeKind,
74         function: Callable[
75             [_FirstType, _SecondType],
76             KindN[_PairLikeKind, _NewFirstType, _NewSecondType, _ThirdType],
77         ],
78     ) -> KindN[_PairLikeKind, _NewFirstType, _NewSecondType, _ThirdType]:
79         """Allows to work with both arguments at the same time."""
80
81     @classmethod
82     @abstractmethod
83     def from_paired(
84         cls: Type[_PairLikeKind],
85         first: _NewFirstType,
86         second: _NewSecondType,
87     ) -> KindN[_PairLikeKind, _NewFirstType, _NewSecondType, _ThirdType]:
88         """Allows to create a PairLikeN from just two values."""
89
90     @classmethod
91     @abstractmethod
92     def from_unpaired(
93         cls: Type[_PairLikeKind],
94         inner_value: _NewFirstType,
95     ) -> KindN[_PairLikeKind, _NewFirstType, _NewFirstType, _ThirdType]:
96         """Allows to create a PairLikeN from just a single object."""
97
98
99 PairLike2 = PairLikeN[_FirstType, _SecondType, NoReturn]
100 PairLike3 = PairLikeN[_FirstType, _SecondType, _ThirdType]

```

(continues on next page)

```

102
103 @final
104 class Pair(
105     BaseContainer,
106     SupportsKind2['Pair', _FirstType, _SecondType],
107     PairLike2[_FirstType, _SecondType],
108 ):
109     """
110     A type that represents a pair of something.
111
112     Like to coordinates ``(x, y)`` or two best friends.
113     Or a question and an answer.
114
115     """
116
117     def __init__(
118         self,
119         inner_value: Tuple[_FirstType, _SecondType],
120     ) -> None:
121         """Saves passed tuple as ``._inner_value`` inside this instance."""
122         super().__init__(inner_value)
123
124     # `Equable` part:
125
126     equals = container_equality # we already have this defined for all types
127
128     # `Mappable` part via `BiMappable`:
129
130     def map(
131         self,
132         function: Callable[[_FirstType], _NewFirstType],
133     ) -> 'Pair[_NewFirstType, _SecondType]':
134         """
135         Changes the first type with a pure function.
136
137         >>> assert Pair((1, 2)).map(str) == Pair('1', 2)
138
139         """
140         return Pair((function(self._inner_value[0]), self._inner_value[1]))
141
142     # `BindableN` part:
143
144     def bind(
145         self,
146         function: Callable[
147             [_FirstType],
148             Kind2['Pair', _NewFirstType, _SecondType],
149         ],
150     ) -> 'Pair[_NewFirstType, _SecondType]':
151         """
152         Changes the first type with a function returning another ``Pair``.
153

```

(continues on next page)

(continued from previous page)

```

154     >>> def bindable(first: int) -> Pair[str, str]:
155         ...     return Pair((str(first), ""))
156
157     >>> assert Pair((1, 'b')).bind(bindable) == Pair(('1', ""))
158
159     """
160     return dekind(function(self._inner_value[0]))
161
162     # `AltableN` part via `BiMappableN`:
163
164     def alt(
165         self,
166         function: Callable[[_SecondType], _NewSecondType],
167     ) -> 'Pair[_FirstType, _NewSecondType]':
168         """
169         Changes the second type with a pure function.
170
171         >>> assert Pair((1, 2)).alt(str) == Pair((1, '2'))
172
173         """
174         return Pair((self._inner_value[0], function(self._inner_value[1])))
175
176     # `LashableN` part:
177
178     def lash(
179         self,
180         function: Callable[
181             [_SecondType],
182             Kind2['Pair', _FirstType, _NewSecondType],
183         ],
184     ) -> 'Pair[_FirstType, _NewSecondType]':
185         """
186         Changes the second type with a function returning `Pair`.
187
188         >>> def lashable(second: int) -> Pair[str, str]:
189             ...     return Pair("", str(second))
190
191         >>> assert Pair(('a', 2)).lash(lashable) == Pair("", '2')
192
193         """
194         return dekind(function(self._inner_value[1]))
195
196     # `SwappableN` part:
197
198     def swap(self) -> 'Pair[_SecondType, _FirstType]':
199         """
200         Swaps `Pair` elements.
201
202         >>> assert Pair((1, 2)).swap() == Pair((2, 1))
203
204         """
205         return Pair((self._inner_value[1], self._inner_value[0]))

```

(continues on next page)

```
206
207 # `PairLikeN` part:
208
209 def pair(
210     self,
211     function: Callable[
212         [_FirstType, _SecondType],
213         Kind2['Pair', _NewFirstType, _NewSecondType],
214     ],
215 ) -> 'Pair[_NewFirstType, _NewSecondType]':
216     """
217     Creates a new `Pair` from an existing one via a passed function.
218
219     >>> def min_max(first: int, second: int) -> Pair[int, int]:
220     ...     return Pair((min(first, second), max(first, second)))
221
222     >>> assert Pair((2, 1)).pair(min_max) == Pair((1, 2))
223     >>> assert Pair((1, 2)).pair(min_max) == Pair((1, 2))
224
225     """
226     return dekind(function(self._inner_value[0], self._inner_value[1]))
227
228 @classmethod
229 def from_paired(
230     cls,
231     first: _NewFirstType,
232     second: _NewSecondType,
233 ) -> 'Pair[_NewFirstType, _NewSecondType]':
234     """
235     Creates a new pair from two values.
236
237     >>> assert Pair.from_paired(1, 2) == Pair((1, 2))
238
239     """
240     return Pair((first, second))
241
242 @classmethod
243 def from_unpaired(
244     cls,
245     inner_value: _NewFirstType,
246 ) -> 'Pair[_NewFirstType, _NewFirstType]':
247     """
248     Creates a new pair from a single value.
249
250     >>> assert Pair.from_unpaired(1) == Pair((1, 1))
251
252     """
253     return Pair((inner_value, inner_value))
254
255
256 # Running hypothesis auto-generated tests:
257 check_all_laws(Pair, use_init=True)
```

9.11.7 Step 6: Writing type-tests

Note: You can find all [type-tests](#) here.

The next thing we want is to write a type-test!

What is a type-test? This is a special type of tests for your typing. We run `mypy` on top of tests and use snapshots to assert the result.

We recommend to use `pytest-mypy-plugins`. [Read more](#) about how to use it.

Let's start with a simple test to make sure our `.pair` function works correctly:

Warning: Please, don't use `env:` property the way we do here. We need it since we store our example in `tests/` folder. And we have to tell `mypy` how to find it.

```

1 - case: test_pair_type
2   disable_cache: false
3   env:
4     # We only need this because we store this example in `tests/`
5     # and not in our source code. Please, do not copy this line!
6     - MYPYPATH=./tests/test_examples/test_your_container
7
8     # TODO: remove this config after
9     #   mypy/typedhed/stdlib/unittest/mock.pyi:120:
10    #   error: Class cannot subclass "Any" (has type "Any")
11    # is fixed.
12  mypy_config:
13    disallow_subclassing_any = False
14  main: |
15    # Let's import our `Pair` type we defined earlier:
16    from test_pair4 import Pair
17
18    reveal_type(Pair)
19
20    def function(first: int, second: str) -> Pair[float, bool]:
21      ...
22
23    my_pair: Pair[int, str] = Pair.from_paired(1, 'a')
24    reveal_type(my_pair.pair(function))
25  out: |
26    main:4: note: Revealed type is "def [_FirstType, _SecondType] (inner_value: Tuple[_
27    ↪_FirstType`1, _SecondType`2]) -> test_pair4.Pair[_FirstType`1, _SecondType`2]"
28    main:10: note: Revealed type is "test_pair4.Pair[builtins.float, builtins.bool]"

```

Ok, now, let's try to raise an error by using it incorrectly:

```

1 - case: test_pair_error
2   disable_cache: false
3   env:
4     # We only need this because we store this example in `tests/`
5     # and not in our source code. Please, do not copy this line!

```

(continues on next page)

(continued from previous page)

```

6   - MYPYPATH=./tests/test_examples/test_your_container
7
8   # TODO: remove this config after
9   #     mypy/typedhed/stdlib/unittest/mock.pyi:120:
10  #     error: Class cannot subclass "Any" (has type "Any")
11  # is fixed.
12  mypy_config:
13    disallow_subclassing_any = False
14  main: |
15    # Let's import our `Pair` type we defined earlier:
16    from test_pair4 import Pair
17
18    # Oups! This function has first and second types swapped!
19    def function(first: str, second: int) -> Pair[float, bool]:
20        ...
21
22    my_pair = Pair.from_paired(1, 'a')
23    my_pair.pair(function) # this should and will error
24  out: |
25    main:9: error: Argument 1 to "pair" of "Pair" has incompatible type "Callable[[str,
↪int], Pair[float, bool]]"; expected "Callable[[int, str], KindN[Pair[Any, Any], float,
↪bool, Any]]" [arg-type]

```

9.11.8 Step 7: Reusing code

The last (but not the least!) thing you need to know is that you can reuse all code we already have for this new `Pair` type.

This is because of our *Higher Kinded Types* feature.

So, let's say we want to use native `map_()` pointfree function with our new `Pair` type. Let's test that it will work correctly:

```

1  - case: test_pair_map
2  disable_cache: false
3  env:
4    # We only need this because we store this example in `tests/`
5    # and not in our source code. Please, do not copy this line!
6    - MYPYPATH=./tests/test_examples/test_your_container
7
8    # TODO: remove this config after
9    #     mypy/typedhed/stdlib/unittest/mock.pyi:120:
10   #     error: Class cannot subclass "Any" (has type "Any")
11   # is fixed.
12   mypy_config:
13     disallow_subclassing_any = False
14   main: |
15     from test_pair4 import Pair
16     from returns.pointfree import map_
17
18     my_pair: Pair[int, int] = Pair.from_unpaired(1)
19     reveal_type(my_pair.map(str))

```

(continues on next page)

(continued from previous page)

```

20     reveal_type(map_(str)(my_pair))
21     out: |
22         main:5: note: Revealed type is "test_pair4.Pair[builtins.str, builtins.int]"
23         main:6: note: Revealed type is "test_pair4.Pair[builtins.str, builtins.int]"

```

Yes, it works!

Now you have fully working, typed, documented, lawful, and tested primitive. You can build any other primitive you need for your business logic or infrastructure.

9.12 Pipelines

The main idea behind functional programming is functional composition.

We provide several tools to make composition easy, readable, pythonic, and useful.

Note: Make sure you are familiar with our *Pointfree* tools, because pipelines and pointfree functions are best friends!

9.12.1 flow

`flow` allows to easily compose multiple functions together into a pipeline. It is useful when you already have an instance to compose functions with.

Note: `flow` is the recommended way to write your code with `returns`!

Let's see an example:

```

>>> from returns.pipeline import flow
>>> assert flow(
...     [1, 2, 3],
...     lambda collection: max(collection),
...     lambda max_number: -max_number,
... ) == -3

```

This allows you to write declarative steps that should be performed on an existing value.

Note: Technical note: `flow` has the best type inference mechanism among all other tools we provide here. This happens due to our *mypy plugins*.

You can also use `flow` with pointfree functions and containers:

```

>>> from returns.result import Result, Success, Failure
>>> from returns.pointfree import bind
>>> from returns.pipeline import flow

>>> def regular_function(arg: int) -> float:
...     return float(arg)

```

(continues on next page)

```
>>> def returns_container(arg: float) -> Result[str, ValueError]:
...     if arg != 0:
...         return Success(str(arg))
...     return Failure(ValueError('Wrong arg'))

>>> def also_returns_container(arg: str) -> Result[str, ValueError]:
...     return Success(arg + '!')

>>> assert flow(
...     1, # initial value
...     regular_function, # composes easily
...     returns_container, # also composes easily, but returns a container
...     # So we need to `bind` the next function to allow it to consume
...     # the container from the previous step.
...     bind(also_returns_container),
... ) == Success('1.0!')

>>> # And this will fail:
>>> assert flow(
...     0, # initial value
...     regular_function, # composes easily
...     returns_container, # also composes easily, but returns a container
...     # So we need to `bind` the next function to allow it to consume
...     # the container from the previous step.
...     bind(also_returns_container),
... ).failure().args == ('Wrong arg', )
```

And now let's get to know pipe, it is very similar, but has different usage pattern.

9.12.2 pipe

pipe is an easy way to compose functions together. It is useful when you don't have an instance to compose functions with yet.

Note: pipe requires to use our *mypy plugins*.

Let's see an example.

```
>>> from returns.pipeline import pipe

>>> pipeline = pipe(str, lambda x: x + 'b', str.upper)
>>> assert pipeline(1) == '1B'
```

It might be later used with multiple values:

```
>>> assert pipeline(2) == '2B'
```

It is also might be useful to compose containers together:


```

>>> from returns.pipeline import pipe
>>> from returns.result import Result, Success, Failure
>>> from returns.pointfree import bind

>>> def regular_function(arg: int) -> float:
...     return float(arg)

>>> def returns_container(arg: float) -> Result[str, ValueError]:
...     if arg != 0:
...         return Success(str(arg))
...     return Failure(ValueError('Wrong arg'))

>>> def also_returns_container(arg: str) -> Result[str, ValueError]:
...     return Success(arg + '!')

>>> transaction = pipe(
...     regular_function, # composes easily
...     returns_container, # also composes easily, but returns a container
...     # So we need to `bind` the next function to allow it to consume
...     # the container from the previous step.
...     bind(also_returns_container),
... )
>>> result = transaction(1) # running the pipeline
>>> assert result == Success('1.0!')

```

You might consider `pipe()` as `returns.functions.compose()` on steroids. The main difference is that `compose` takes strictly two arguments (or you might say that it has an arity of two), while `pipe` has infinite possible arguments.

9.12.3 managed

A really common task is to work with something stateful, like database connections or files.

First, you need to acquire some resource, then use it and do your thing, and clear things up and release the acquired resource.

There are several rules here:

1. If the acquiring failed, then do nothing: do not try to use the resource or release it
2. If the resource is acquired, then try to use it and then release it despite of the usage result

In other words, if you cannot open a file, then do nothing. If you opened it, then try to read it. And then always close it.

Let's say you have to read a file's contents:

```

>>> from typing import TextIO
>>> from returns.pipeline import managed, is_successful
>>> from returns.result import ResultE
>>> from returns.io import IOResultE, impure_safe

>>> def read_file(file_obj: TextIO) -> IOResultE[str]:
...     return impure_safe(file_obj.read)() # this will be the final result

>>> def close_file(
...     file_obj: TextIO,

```

(continues on next page)

(continued from previous page)

```

...     file_contents: ResultE[str],
... ) -> IOResultE[None]: # sometimes might require to use `untap`
...     return impure_safe(file_obj.close()) # this value will be dropped

>>> managed_read = managed(read_file, close_file)

>>> read_result = managed_read(
...     impure_safe(lambda filename: open(filename, 'r'))('pyproject.toml'),
... )
>>> assert is_successful(read_result) # file content is inside `IOSuccess`

```

And here's how we recommend to combine managed with other pipeline functions:

```

>>> import tomlkit
>>> from returns.pipeline import flow
>>> from returns.pointfree import bind_result
>>> from returns.result import safe
>>> from returns.io import IOSuccess

>>> @safe
... def parse_toml(file_contents: str) -> dict:
...     return tomlkit.parse(file_contents)

>>> @safe
... def get_project_name(parsed: dict) -> str:
...     return parsed['tool']['poetry']['name']

>>> pipeline_result = flow(
...     'pyproject.toml', # filename we work with
...     impure_safe(lambda filename: open(filename, 'r')),
...     managed_read,
...     bind_result(parse_toml),
...     bind_result(get_project_name),
... )
>>> assert pipeline_result == IOSuccess('returns')

```

Notice a few tricks here:

1. We use managed with and without flow here, both are fine!
2. We have created a managed_read managed function, so we don't need to specify it every time we want to read a file in a functional way
3. We are using impure and pure operations inside the pipeline: this helps us to understand how our app works. Which parts do access the file system and which just work

However, you can still use the imperative approach with with: or try/finally wrapped into @impure_safe decorator, your choice! We don't recommend to mix these two. Stick to one you like the most.

managed can be used with:

- IOResult
- FutureResult
- RequiresContextIOResult

- RequiresContextFutureResult

9.12.4 is_successful

`is_successful` is used to tell whether or not your result is a success. We treat only three types that do not throw as successful ones, basically: `Success`, `IOSuccess`, and `Some`

```
>>> from returns.result import Success, Failure
>>> from returns.pipeline import is_successful

>>> assert is_successful(Success(1)) is True
>>> assert is_successful(Failure('text')) is False
```

9.12.5 Further reading

- fp-ts pipeable
- ZIO Managed

9.12.6 API Reference

flow(*instance*, **functions*)

Allows to compose a value and up to multiple functions that use this value.

All starts with the value itself. Each next function uses the previous result as an input parameter.

We use a custom mypy plugin to make sure types are correct. Otherwise, it is currently impossible to properly type this function.

Currently, `flow` has a hard limit of 21 steps. Because, it is not possible to type it otherwise. We need a hard limit. See: <https://github.com/dry-python/returns/issues/461>

Here's how it should be used:

```
>>> from returns.pipeline import flow

>>> # => executes: str(float(int('1')))
>>> assert flow('1', int, float, str) == '1.0'
```

This function is closely related to `pipe`:

```
>>> from returns.pipeline import pipe
>>> assert flow('1', int, float, str) == pipe(int, float, str)('1')
```

See also:

- <https://stackoverflow.com/a/41585450/4842742>
- <https://github.com/gcanti/fp-ts/blob/master/src/pipeable.ts>

Requires our *mypy plugin*.

Parameters

- **instance** (TypeVar(_InstanceType)) –

- **functions** (TypeVar(_PipelineStepType)) –

Return type

TypeVar(_ReturnType)

pipe(*functions)

Allows to compose a value and up to 7 functions that use this value.

We use a custom mypy plugin to make sure types are correct. Otherwise, it is currently impossible to properly type this function.

Each next function uses the previous result as an input parameter. Here's how it should be used:

```
>>> from returns.pipeline import pipe

>>> # => executes: str(float(int('1')))
>>> assert pipe(int, float, str>('1') == '1.0'
```

This function is closely related to pipe:

```
>>> from returns.pipeline import flow
>>> assert pipe(int, float, str>('1') == flow('1', int, float, str)
```

See also:

- <https://stackoverflow.com/a/41585450/4842742>
- <https://github.com/gcanti/fp-ts/blob/master/src/pipeable.ts>

managed(use, release)

Allows to run managed computation.

Managed computations consist of three steps:

1. **acquire** when we get some initial resource to work with
2. **use** when the main logic is done
3. **release** when we release acquired resource

Let's look at the example:

1. We need to acquire an opened file to read it later
2. We need to use acquired file to read its content
3. We need to release the acquired file in the end

Here's a code example:

```
>>> from returns.pipeline import managed
>>> from returns.io import IOSuccess, IOFailure, impure_safe

>>> class Lock(object):
...     "Example class to emulate state to acquire and release."
...     def __init__(self, default: bool = False) -> None:
...         self.set = default
...     def __eq__(self, lock) -> bool: # we need this for testing
...         return self.set == lock.set
...     def release(self) -> None:
```

(continues on next page)

(continued from previous page)

```

...     self.set = False

>>> pipeline = managed(
...     lambda lock: IOSuccess(lock) if lock.set else IOFailure(False),
...     lambda lock, use_result: impure_safe(lock.release)(),
... )

>>> assert pipeline(IOSuccess(Lock(True))) == IOSuccess(Lock(False))
>>> assert pipeline(IOSuccess(Lock())) == IOFailure(False)
>>> assert pipeline(IOFailure('no lock')) == IOFailure('no lock')

```

See also:

- <https://github.com/gcanti/fp-ts/blob/master/src/IOEither.ts>
- https://zio.dev/docs/datatypes/datatypes_managed

Implementation

This class requires some explanation.

First of all, we modeled this function as a class, so it can be partially applied easily.

Secondly, we used imperative approach of programming inside this class. Functional approach was 2 times slower. And way more complex to read and understand.

Lastly, we try to hide these two things for the end user. We pretend that this is not a class, but a function. We also do not break a functional abstraction for the end user. It is just an implementation detail.

Type inference does not work so well with lambda functions. But, we do not recommend to use this function with lambda functions.

Parameters

- **use** (Callable[[TypeVar(_FirstType)], KindN[TypeVar(_IOResultLikeType, bound= IOResultLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]] –
- **release** (Callable[[TypeVar(_FirstType), Result[TypeVar(_UpdatedType), TypeVar(_SecondType)]], KindN[TypeVar(_IOResultLikeType, bound= IOResultLikeN), None, TypeVar(_SecondType), TypeVar(_ThirdType)]] –

Return type

Kinded[Callable[[KindN[TypeVar(_IOResultLikeType, bound= IOResultLikeN), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], KindN[TypeVar(_IOResultLikeType, bound= IOResultLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

is_successful(container)

Determines if a container was successful or not.

```

>>> from returns.maybe import Some, Nothing
>>> from returns.result import Failure, Success
>>> from returns.io import IOSuccess, IOFailure

>>> assert is_successful(Some(1))

```

(continues on next page)

(continued from previous page)

```
>>> assert not is_successful(Nothing)
>>> assert is_successful(Success(1))
>>> assert not is_successful(Failure(1))

>>> assert is_successful(IOSuccess(1))
>>> assert not is_successful(IOFailure(1))
```

This function can work with containers that are instance of `returns.interfaces.unwrappable.Unwrappable`.

Parameters

container (*Unwrappable*) –

Return type

bool

9.13 Converters

We have several helpers to convert containers from one type to another and back again.

9.13.1 Maybe and Result

We have two converters to work with Result <-> Maybe transformations:

- `maybe_to_result()` that converts Maybe to Result
- `result_to_maybe()` that converts Result to Maybe

That's how they work:

```
>>> from returns.converters import maybe_to_result, result_to_maybe
>>> from returns.maybe import Maybe, Some
>>> from returns.result import Result, Success

>>> result: Result[int, Exception] = Success(1)
>>> maybe: Maybe[int] = result_to_maybe(result)
>>> assert maybe == Some(1)

>>> new_result: Result[int, None] = maybe_to_result(maybe)
>>> assert new_result == Success(1)
```

Take a note, that type changes. Also, take a note that `Success(None)` will be converted to `Nothing`.

9.13.2 flatten

You can also use *flatten* to merge nested containers together:

```
>>> from returns.converters import flatten
>>> from returns.maybe import Some
>>> from returns.result import Success
>>> from returns.io import IO

>>> assert flatten(IO(IO(1))) == IO(1)
>>> assert flatten(Some(Some(1))) == Some(1)
>>> assert flatten(Success(Success(1))) == Success(1)
```

9.13.3 API Reference

flatten(*container*)

Joins two nested containers together.

Please, note that it will not join two Failure for Result case or two Nothing for Maybe case (or basically any two error types) together.

```
>>> from returns.converters import flatten
>>> from returns.io import IO
>>> from returns.result import Failure, Success

>>> assert flatten(IO(IO(1))) == IO(1)

>>> assert flatten(Success(Success(1))) == Success(1)
>>> assert flatten(Failure(Failure(1))) == Failure(Failure(1))
```

See also:

- <https://bit.ly/2sIviUr>

Parameters

container (*KindN*[*TypeVar*(*_BindableKind*, bound=*BindableN*), *KindN*[*TypeVar*(*_BindableKind*, bound=*BindableN*), *TypeVar*(*_FirstType*), *TypeVar*(*_SecondType*), *TypeVar*(*_ThirdType*)], *TypeVar*(*_SecondType*), *TypeVar*(*_ThirdType*)] –

Return type

KindN[*TypeVar*(*_BindableKind*, bound=*BindableN*), *TypeVar*(*_FirstType*), *TypeVar*(*_SecondType*), *TypeVar*(*_ThirdType*)]

result_to_maybe(*result_container*)

Converts Result container to Maybe container.

```
>>> from returns.maybe import Some, Nothing
>>> from returns.result import Failure, Success

>>> assert result_to_maybe(Success(1)) == Some(1)
>>> assert result_to_maybe(Success(None)) == Some(None)
```

(continues on next page)

(continued from previous page)

```
>>> assert result_to_maybe(Failure(1)) == Nothing
>>> assert result_to_maybe(Failure(None)) == Nothing
```

Parameters**result_container** (*Result*[TypeVar(_FirstType), TypeVar(_SecondType)]) –**Return type***Maybe*[TypeVar(_FirstType)]**maybe_to_result**(*maybe_container*)

Converts Maybe container to Result container.

```
>>> from returns.maybe import Some, Nothing
>>> from returns.result import Failure, Success

>>> assert maybe_to_result(Some(1)) == Success(1)
>>> assert maybe_to_result(Some(None)) == Success(None)
>>> assert maybe_to_result(Nothing) == Failure(None)
```

Parameters**maybe_container** (*Maybe*[TypeVar(_FirstType)]) –**Return type***Result*[TypeVar(_FirstType), None]

9.14 Pointfree

This module provides a bunch of primitives to work with containers.

It makes composing functions with containers easier. Sometimes using methods on containers is not very helpful. Container methods are difficult to compose with other functions or methods.

Instead we can use functions that produce the same result but have the reverse semantics.

Usually, this means changing something like $x.f(y)$ to $f(x)(y)$.

Why would anyone need these functions when you can use methods? To create pipelines!

```
from returns.pipeline import pipe
from returns.result import ResultE

def returns_result(arg: int) -> ResultE[int]:
    ...

def works_with_result(arg: int) -> ResultE[int]:
    ...

def finish_work(arg: int) -> ResultE[int]:
    ...

pipe(
    returns_result,
```

(continues on next page)

(continued from previous page)

```
works_with_result, # does not compose! Needs a container for input
finish_work, # does not compose either!
)
```

Without pointfree functions you would probably have to write:

```
returns_result().bind(works_with_result).bind(notifies_user)
```

And you need a way to somehow do this in the pipeline syntax. Remember that pipeline syntax helps make composing functions more readable and pythonic. That's where pointfree functions become really useful.

9.14.1 map

`map_()` is a pointfree alternative to the container method `.map()`.

It lifts a function to work from container to container. `map_(f)` would return `f` lifted to work on a container.

In other words, it modifies the function's signature from: `a -> b` to: `Container[a] -> Container[b]`

Doing this lets us compose regular functions and containers.

```
>>> from returns.pointfree import map_
>>> from returns.maybe import Maybe, Some

>>> def as_int(arg: str) -> int:
...     return ord(arg)

>>> container: Maybe[str] = Some('a')
>>> # We now have two ways to compose container and as_int
>>> # 1. Via ``.map()``:
>>> assert container.map(as_int) == Some(97)
>>> # 2. Or via ``map_()`` , like above but in the reverse order:
>>> assert map_(as_int)(container) == Some(97)
```

This means we can compose functions in a pipeline.

```
>>> from returns.pointfree import map_
>>> from returns.pipeline import flow
>>> from returns.maybe import Maybe, Some, Nothing

>>> def index_of_7(arg: str) -> Maybe[int]:
...     if '7' in arg:
...         return Some(arg.index('7'))
...     return Nothing

>>> def double(num: int) -> int:
...     return num * 2

>>> assert flow(
...     '007',
...     index_of_7, # Some(2)
...     map_(double), # Some(4)
... ) == Some(4)
```

(continues on next page)

```

>>> # Still passes along Nothing
>>> assert flow(
...     '006',
...     index_of_7, # Nothing
...     map_(double), # Nothing
... ) == Nothing

```

9.14.2 bind

Pointfree `bind()` is an alternative to the container method `.bind()`.

It binds a function that returns a container so that it accepts the same container type as input.

In other words, it modifies the function's signature from: `a -> Container[b]` to: `Container[a] -> Container[b]`

Without `bind()` it would be very hard to declaratively compose two entities:

1. Existing containers
2. Existing functions that accept a regular value and return a container

We can compose these entities with `.bind()` when calling it on a container, but how can we do it independently?

```

>>> from returns.pointfree import bind
>>> from returns.maybe import Maybe, Some

>>> def index_of_1(arg: str) -> Maybe[int]:
...     if '1' in arg:
...         return Some(arg.index('1'))
...     return Nothing

>>> container = Some('A1 Steak Sauce')
>>> # We now have two way of composing these entities.
>>> # 1. Via ``.bind``:
>>> assert container.bind(index_of_1) == Some(1)
>>> # 2. Or via the ``bind`` function.
>>> assert bind(index_of_1)(container) == Some(1)
>>> # This produces the same result, but in a different order

```

That's it!

We also have a long list of other `bind_*` functions, like:

- `bind_io` to bind functions returning IO container
- `bind_result` to bind functions returning Result container
- `bind_ioresult` to bind functions returning IOResult container
- `bind_future` to bind functions returning Future container
- `bind_async_future` to bind async functions returning Future container
- `bind_future_result` to bind functions returning FutureResult container
- `bind_async_future_result` to bind async functions returning FutureResult container

- `bind_context` to bind functions returning `RequiresContext` container
- `bind_context_result` to bind functions returning `RequiresContextResult` container
- `bind_context_ioresult` to bind functions returning `RequiresContextIOResult` container
- `bind_async` to bind async functions returning `Future` or `FutureResult`
- `bind_awaitable` to bind async non-container functions

9.14.3 alt

Pointfree `alt()` is an alternative to the container method `.alt()`.

It lifts a function to act on the error contents of a container.

In other words, it modifies the function's signature from: `a -> b` to: `Container[_ , a] -> Container[_ , b]`

You can think of it like `map`, but for the second type of a container.

```
>>> from returns.io import IOFailure, IOSuccess
>>> from returns.pointfree import alt

>>> def half_as_bad(error_code: int) -> float:
...     return error_code / 2

>>> # When acting on a successful state, nothing happens.
>>> assert alt(half_as_bad)(IOSuccess(1)) == IOSuccess(1)

>>> # When acting on a failed state, the result changes
>>> assert alt(half_as_bad)(IOFailure(4)) == IOFailure(2.0)

>>> # This is equivalent to IOFailure(4).alt(half_as_bad)
>>> assert alt(half_as_bad)(IOFailure(4)) == IOFailure(4).alt(half_as_bad)
```

This inverse syntax lets us easily compose functions in a pipeline

```
>>> from returns.io import IOFailure, IOSuccess, IOResult
>>> from returns.pointfree import alt

>>> def always_errors(user_input: str) -> IOResult:
...     return IOFailure(len(user_input))

>>> def twice_as_bad(exit_code: int) -> int:
...     return exit_code * 2

>>> def make_error_message(exit_code: int) -> str:
...     return 'Badness level: {0}'.format(exit_code)

>>> assert flow(
...     '12345',
...     always_errors,
...     alt(twice_as_bad),
...     alt(make_error_message)
... ) == IOFailure('Badness level: 10')
```

9.14.4 lash

Pointfree `lash()` function is an alternative to `.lash()` container method.

It allows better composition by lifting a function that returns a container to act on the failed state of a container.

You can think of it like `bind`, but for the second type of a container.

```
>>> from returns.pointfree import lash
>>> from returns.result import Success, Failure, Result

>>> def always_succeeds(arg: str) -> Result[int, str]:
...     return Success(1)

>>> failed: Result[int, str] = Failure('a')
>>> # We now have two way of composing these entities.
>>> # 1. Via ``.lash``:
>>> assert failed.lash(always_succeeds) == Success(1)
>>> # 2. Or via ``lash`` function, the same but in the inverse way:
>>> assert lash(always_succeeds)(failed) == Success(1)
```

9.14.5 apply

Pointfree `apply` function allows to use `.apply()` container method like a function:

```
>>> from returns.pointfree import apply
>>> from returns.maybe import Some, Nothing

>>> def wow(arg: int) -> str:
...     return chr(arg) + '!'

>>> assert apply(Some(wow))(Some(97)) == Some('a!')
>>> assert apply(Some(wow))(Some(98)) == Some('b!')
>>> assert apply(Some(wow))(Nothing) == Nothing
>>> assert apply(Nothing)(Nothing) == Nothing
```

If you wish to use `apply` inside a pipeline here's how it might look:

```
>>> from returns.pointfree import apply
>>> from returns.pipeline import flow
>>> from returns.maybe import Some, Nothing, Maybe
>>> from typing import Callable

>>> def wow(arg: int) -> str:
...     return chr(arg) + '!'

>>> def my_response(is_excited: bool) -> Maybe[Callable[[int], str]]:
...     if is_excited:
...         return Some(wow)
...     return Nothing

>>> assert flow(
...     Some(97),
```

(continues on next page)

(continued from previous page)

```

...     apply(my_response(True)),
... ) == Some('a!')

>>> assert flow(
...     Nothing,
...     apply(my_response(False)),
... ) == Nothing

```

Or with a function as the first parameter:

```

>>> from returns.pipeline import flow
>>> from returns.curry import curry
>>> from returns.maybe import Some

>>> @curry
... def add_curried(first: int, second: int) -> int:
...     return first + second

>>> assert flow(
...     Some(add_curried),
...     Some(2).apply,
...     Some(3).apply,
... ) == Some(5)

```

9.14.6 compose_result

Sometimes we need to manipulate the inner Result of some containers like IOResult or FutureResult. With `compose_result` we can do this kind of manipulation.

```

>>> from returns.pointfree import compose_result
>>> from returns.io import IOResult, IOSuccess, IOFailure
>>> from returns.result import Result

>>> def cast_to_str(container: Result[float, int]) -> IOResult[str, int]:
...     return IOResult.from_result(container.map(str))

>>> assert compose_result(cast_to_str)(IOSuccess(42.0)) == IOSuccess('42.0')
>>> assert compose_result(cast_to_str)(IOFailure(1)) == IOFailure(1)

```

9.14.7 cond

Sometimes we need to create `SingleFailableN` or `DiverseFailableN` containers (e.g. `Maybe`, `ResultLikeN`) based on a boolean expression, `cond` can help us.

Consider `cond` to be a functional `if`.

See the example below:

```

>>> from returns.pipeline import flow
>>> from returns.pointfree import cond
>>> from returns.result import Result, Failure, Success

```

(continues on next page)

```
>>> def returns_boolean(arg: int) -> bool:
...     return bool(arg)

>>> assert flow(
...     returns_boolean(1),
...     cond(Result, 'success', 'failure')
... ) == Success('success')

>>> assert flow(
...     returns_boolean(0),
...     cond(Result, 'success', 'failure')
... ) == Failure('failure')
```

Example using `cond` with the `Maybe` container:

```
>>> from returns.pipeline import flow
>>> from returns.pointfree import cond
>>> from returns.maybe import Maybe, Some, Nothing

>>> assert flow(
...     returns_boolean(1),
...     cond(Maybe, 'success')
... ) == Some('success')

>>> assert flow(
...     returns_boolean(0),
...     cond(Maybe, 'success')
... ) == Nothing
```

9.14.8 Further reading

- Tacit programming or point-free style
- Pointfree in Haskell

9.14.9 API Reference

`map_(function)`

Lifts function to be wrapped in a container for better composition.

In other words, it modifies the function's signature from: `a -> b` to: `Container[a] -> Container[b]`

This is how it should be used:

```
>>> from returns.io import IO
>>> from returns.pointfree import map_

>>> def example(argument: int) -> float:
...     return argument / 2

>>> assert map_(example)(IO(1)) == IO(0.5)
```

Note, that this function works for all containers with `.map` method. See `returns.primitives.interfaces.mappable.MappableN` for more info.

See also:

- <https://wiki.haskell.org/Lifting>

Parameters

function (Callable[[TypeVar(_FirstType)], TypeVar(_UpdatedType)]) –

Return type

Kinded[Callable[[*KindN*[TypeVar(_MappableKind, bound= *MappableN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_MappableKind, bound= *MappableN*), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

`bind(function)`

Turns function's input parameter from a regular value to a container.

In other words, it modifies the function signature from: `a -> Container[b]` to: `Container[a] -> Container[b]`

Similar to `returns.pointfree.lash()`, but works for successful containers. This is how it should be used:

```
>>> from returns.pointfree import bind
>>> from returns.maybe import Maybe, Some, Nothing

>>> def example(argument: int) -> Maybe[int]:
...     return Some(argument + 1)

>>> assert bind(example)(Some(1)) == Some(2)
>>> assert bind(example)(Nothing) == Nothing
```

Note, that this function works for all containers with `.bind` method. See `returns.primitives.interfaces.bindable.BindableN` for more info.

Parameters

function (Callable[[TypeVar(_FirstType)], *KindN*[TypeVar(_BindableKind, bound= *BindableN*), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]])

–

Return type

Kinded[Callable[[*KindN*[TypeVar(_BindableKind, bound= *BindableN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_BindableKind, bound= *BindableN*), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

`bind_result(function)`

Composes successful container with a function that returns a container.

In other words, it modifies the function's signature from: `a -> Result[b, c]` to: `Container[a, c] -> Container[b, c]`

```
>>> from returns.io import IOSuccess
>>> from returns.context import RequiresContextResult
>>> from returns.result import Result, Success
>>> from returns.pointfree import bind_result
```

(continues on next page)

(continued from previous page)

```

>>> def returns_result(arg: int) -> Result[int, str]:
...     return Success(arg + 1)

>>> bound = bind_result(returns_result)
>>> assert bound(IOSuccess(1)) == IOSuccess(2)
>>> assert bound(RequiresContextResult.from_value(1))(...) == Success(2)

```

Parameters

function (Callable[[TypeVar(_FirstType)], *Result*[TypeVar(_UpdatedType), TypeVar(_SecondType)]] –

Return type

Kinded[Callable[[*KindN*[TypeVar(_ResultLikeKind, bound=*ResultLikeN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_ResultLikeKind, bound=*ResultLikeN*), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

bind_io(function)

Composes successful container with a function that returns a container.

In other words, it modifies the function's signature from: `a -> IO[b]` to: `Container[a, c] -> Container[b, c]`

```

>>> from returns.io import IOSuccess, IOFailure
>>> from returns.io import IO
>>> from returns.pointfree import bind_io

>>> def returns_io(arg: int) -> IO[int]:
...     return IO(arg + 1)

>>> bound = bind_io(returns_io)
>>> assert bound(IO(1)) == IO(2)
>>> assert bound(IOSuccess(1)) == IOSuccess(2)
>>> assert bound(IOFailure(1)) == IOFailure(1)

```

Parameters

function (Callable[[TypeVar(_FirstType, contravariant=True)], *IO*[TypeVar(_UpdatedType, covariant=True)]] –

Return type

Kinded[Callable[[*KindN*[TypeVar(_IOLikeKind, bound=*IOLikeN*), TypeVar(_FirstType, contravariant=True), TypeVar(_SecondType), TypeVar(_ThirdType, contravariant=True)]], *KindN*[TypeVar(_IOLikeKind, bound=*IOLikeN*), TypeVar(_UpdatedType, covariant=True), TypeVar(_SecondType), TypeVar(_ThirdType, contravariant=True)]]]

bind_ioresult(function)

Composes successful container with a function that returns a container.

In other words, it modifies the function's signature from: `a -> IOResult[b, c]` to: `Container[a, c] -> Container[b, c]`


```

>>> from returns.io import IOResult, IOSuccess
>>> from returns.context import RequiresContextIOResult
>>> from returns.pointfree import bind_ioresult

>>> def returns_ioresult(arg: int) -> IOResult[int, str]:
...     return IOSuccess(arg + 1)

>>> bound = bind_ioresult(returns_ioresult)
>>> assert bound(IOSuccess(1)) == IOSuccess(2)
>>> assert bound(
...     RequiresContextIOResult.from_value(1),
... )(...) == IOSuccess(2)

```

Parameters

function (Callable[[TypeVar(_FirstType)], *IOResult*[TypeVar(_UpdatedType), TypeVar(_SecondType)]])

Return type

Kinded[Callable[[*KindN*[TypeVar(_IOResultLikeKind, bound= *IOResultLikeN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_IOResultLikeKind, bound= *IOResultLikeN*), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

bind_future(*function*)

Compose a container and sync function returning Future.

In other words, it modifies the function signature from: `a -> Future[b]` to: `Container[a] -> Container[b]`

Similar to `returns.pointfree.lash()`, but works for successful containers. This is how it should be used:

```

>>> import anyio
>>> from returns.pointfree import bind_future
>>> from returns.future import Future
>>> from returns.io import IO

>>> def example(argument: int) -> Future[int]:
...     return Future.from_value(argument + 1)

>>> assert anyio.run(
...     bind_future(example)(Future.from_value(1)).awaitable,
... ) == IO(2)

```

Note, that this function works for all containers with `.bind_future` method. See `returns.primitives.interfaces.specific.future.FutureLikeN` for more info.

Parameters

function (Callable[[TypeVar(_FirstType)], *Future*[TypeVar(_UpdatedType)]])

Return type

Kinded[Callable[[*KindN*[TypeVar(_FutureKind, bound= *FutureLikeN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_FutureKind, bound= *FutureLikeN*), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

bind_async_future(*function*)

Compose a container and async function returning Future.

In other words, it modifies the function signature from: `a -> Awaitable[Future[b]]` to: `Container[a] -> Container[b]`

This is how it should be used:

```
>>> import anyio
>>> from returns.pointfree import bind_async_future
>>> from returns.future import Future
>>> from returns.io import IO

>>> async def example(argument: int) -> Future[int]:
...     return Future.from_value(argument + 1)

>>> assert anyio.run(
...     bind_async_future(example)(Future.from_value(1)).awaitable,
... ) == IO(2)
```

Note, that this function works for all containers with `.bind_async_future` method. See `returns.primitives.interfaces.specific.future.FutureLikeN` for more info.

Parameters

`function`(Callable[[TypeVar(_FirstType)], Awaitable[Future[TypeVar(_UpdatedType)]]])

–

Return type

`Kinded`[Callable[[KindN[TypeVar(_FutureKind), bound= FutureLikeN], TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], KindN[TypeVar(_FutureKind, bound= FutureLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

bind_future_result(*function*)

Compose a container and async function returning FutureResult.

In other words, it modifies the function signature from: `a -> FutureResult[b, c]` to: `Container[a, c] -> Container[b, c]`

This is how it should be used:

```
>>> import anyio
>>> from returns.pointfree import bind_future_result
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> def example(argument: int) -> FutureResult[int, str]:
...     return FutureResult.from_value(argument + 1)

>>> assert anyio.run(
...     bind_future_result(example)(
...         FutureResult.from_value(1),
...     ).awaitable,
... ) == IOSuccess(2)

>>> assert anyio.run(
...     bind_future_result(example)(
```

(continues on next page)

(continued from previous page)

```

...     FutureResult.from_failure('a'),
...     ).awaitable,
... ) == IOFailure('a')

```

Note, that this function works for all containers with `.bind_async_future` method. See `FutureResultLikeN` for more info.

Parameters

function (Callable[[TypeVar(_FirstType)], `FutureResult`[TypeVar(_UpdatedType), TypeVar(_SecondType)]] –

Return type

`Kinded`[Callable[[`KindN`[TypeVar(_FutureResultKind, bound= `FutureResultLikeN`), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], `KindN`[TypeVar(_FutureResultKind, bound= `FutureResultLikeN`), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

bind_async_future_result (*function*)

Compose a container and async function returning `FutureResult`.

In other words, it modifies the function signature from: `a -> Awaitable[FutureResult[b, c]]` to: `Container[a, c] -> Container[b, c]`

This is how it should be used:

```

>>> import anyio
>>> from returns.pointfree import bind_async_future_result
>>> from returns.future import FutureResult
>>> from returns.io import IOSuccess, IOFailure

>>> async def example(argument: int) -> FutureResult[int, str]:
...     return FutureResult.from_value(argument + 1)

>>> assert anyio.run(
...     bind_async_future_result(example)(
...         FutureResult.from_value(1),
...     ).awaitable,
... ) == IOSuccess(2)

>>> assert anyio.run(
...     bind_async_future_result(example)(
...         FutureResult.from_failure('a'),
...     ).awaitable,
... ) == IOFailure('a')

```

Note, that this function works for all containers with `.bind_async_future` method. See `FutureResultLikeN` for more info.

Parameters

function (Callable[[TypeVar(_FirstType)], Awaitable[`FutureResult`[TypeVar(_UpdatedType), TypeVar(_SecondType)]]] –

Return type

`Kinded`[Callable[[`KindN`[TypeVar(_FutureResultKind, bound= `FutureResultLikeN`), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]],

```
KindN[TypeVar(_FutureResultKind, bound= FutureResultLikeN),
TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]
```

bind_context2(function)

Composes successful container with a function that returns a container.

In other words, it modifies the function's signature from: `a -> RequiresContext[b, c]` to: `Container[a, c] -> Container[b, c]`

```
>>> from returns.pointfree import bind_context2
>>> from returns.context import Reader

>>> def example(argument: int) -> Reader[int, int]:
...     return Reader(lambda deps: argument + deps)

>>> assert bind_context2(example)(Reader.from_value(2))(3) == 5
```

Note, that this function works with only Kind2 containers with `.bind_context` method. See `returns.primitives.interfaces.specific.reader.ReaderLike2` for more info.

Parameters

function(Callable[[TypeVar(_FirstType)], RequiresContext[TypeVar(_UpdatedType), TypeVar(_SecondType)]])

Return type

Kinded[Callable[[*KindN*[TypeVar(_Reader2Kind, bound= ReaderLike2), TypeVar(_FirstType), TypeVar(_SecondType), Any]], *KindN*[TypeVar(_Reader2Kind, bound= ReaderLike2), TypeVar(_UpdatedType), TypeVar(_SecondType), Any]]]

bind_context3(function)

Composes successful container with a function that returns a container.

In other words, it modifies the function's signature from: `a -> RequiresContext[b, c]` to: `Container[a, c] -> Container[b, c]`

```
>>> from returns.context import RequiresContext, RequiresContextResult
>>> from returns.result import Success, Failure
>>> from returns.pointfree import bind_context

>>> def function(arg: int) -> RequiresContext[str, int]:
...     return RequiresContext(lambda deps: len(deps) + arg)

>>> assert bind_context(function)(
...     RequiresContextResult.from_value(2),
... )('abc') == Success(5)
>>> assert bind_context(function)(
...     RequiresContextResult.from_failure(0),
... )('abc') == Failure(0)
```

Note, that this function works with only Kind3 containers with `.bind_context` method. See `returns.primitives.interfaces.specific.reader.ReaderLike3` for more info.

Parameters

function(Callable[[TypeVar(_FirstType)], RequiresContext[TypeVar(_UpdatedType), TypeVar(_ThirdType)]])

Return type

Kinded[Callable[[*KindN*[TypeVar(_Reader3Kind, bound= ReaderLike3),

```
TypeVar(_FirstType),      TypeVar(_SecondType),      TypeVar(_ThirdType)],
KindN[TypeVar(_Reader3Kind, bound= ReaderLike3), TypeVar(_UpdatedType),
TypeVar(_SecondType), TypeVar(_ThirdType)]]]
```

bind_context(function)

Useful alias for `bind_context3()`.

Parameters

function(Callable[[TypeVar(_FirstType)], RequiresContext[TypeVar(_UpdatedType), TypeVar(_ThirdType)]] –

Return type

```
Kinded[Callable[[KindN[TypeVar(_Reader3Kind,      bound= ReaderLike3),
TypeVar(_FirstType),      TypeVar(_SecondType),      TypeVar(_ThirdType)],
KindN[TypeVar(_Reader3Kind, bound= ReaderLike3), TypeVar(_UpdatedType),
TypeVar(_SecondType), TypeVar(_ThirdType)]]]]]
```

modify_env2(function)

Modifies the second type argument of a ReaderLike2.

In other words, it modifies the function's signature from: `a -> b` to: `Container[x, a] -> Container[x, b]`

```
>>> from returns.pointfree import modify_env2
>>> from returns.context import RequiresContext

>>> def multiply(arg: int) -> RequiresContext[int, int]:
...     return RequiresContext(lambda deps: arg * deps)

>>> assert modify_env2(int)(multiply(3))('4') == 12
```

Note, that this function works with only Kind2 containers with `.modify_env` method. See `returns.primitives.interfaces.specific.reader.ReaderLike2` for more info.

Parameters

function (Callable[[TypeVar(_UpdatedType)], TypeVar(_SecondType)]) –

Return type

```
Kinded[Callable[[KindN[TypeVar(_Reader2Kind,      bound= ReaderLike2),
TypeVar(_FirstType), TypeVar(_SecondType), Any]], KindN[TypeVar(_Reader2Kind,
bound= ReaderLike2), TypeVar(_FirstType), TypeVar(_UpdatedType), Any]]]
```

modify_env3(function)

Modifies the third type argument of a ReaderLike3.

In other words, it modifies the function's signature from: `a -> b` to: `Container[x, a] -> Container[x, b]`

```
>>> from returns.pointfree import modify_env
>>> from returns.context import RequiresContextResultE
>>> from returns.result import Success, safe

>>> def divide(arg: int) -> RequiresContextResultE[float, int]:
...     return RequiresContextResultE(safe(lambda deps: arg / deps))

>>> assert modify_env(int)(divide(3))('2') == Success(1.5)
>>> assert modify_env(int)(divide(3))('0').failure()
```

Note, that this function works with only Kind3 containers with `.modify_env` method. See `returns.primitives.interfaces.specific.reader.ReaderLike3` for more info.

Parameters

function (Callable[[TypeVar(_UpdatedType)], TypeVar(_ThirdType)]) –

Return type

Kinded[Callable[[*KindN*[TypeVar(_Reader3Kind, bound= *ReaderLike3*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_Reader3Kind, bound= *ReaderLike3*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_UpdatedType)]]]

modify_env(function)

Useful alias for `modify_env3()`.

Parameters

function (Callable[[TypeVar(_UpdatedType)], TypeVar(_ThirdType)]) –

Return type

Kinded[Callable[[*KindN*[TypeVar(_Reader3Kind, bound= *ReaderLike3*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_Reader3Kind, bound= *ReaderLike3*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_UpdatedType)]]]

bind_context_result(function)

Composes successful container with a function that returns a container.

In other words, it modifies the function's signature from: `a -> ReaderResult[b, c, e]` to: `Container[a, c, e] -> Container[b, c, e]`

```
>>> from returns.pointfree import bind_context_result
>>> from returns.context import ReaderIOResult, ReaderResult
>>> from returns.io import IOSuccess, IOFailure

>>> def example(argument: int) -> ReaderResult[int, str, str]:
...     return ReaderResult.from_value(argument + 1)

>>> assert bind_context_result(example)(
...     ReaderIOResult.from_value(1),
... )(...) == IOSuccess(2)
>>> assert bind_context_result(example)(
...     ReaderIOResult.from_failure('a'),
... )(...) == IOFailure('a')
```

Parameters

function (Callable[[TypeVar(_FirstType)], *RequiresContextResult*[TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]] –

Return type

Kinded[Callable[[*KindN*[TypeVar(_ReaderResultLikeKind, bound= *ReaderResultLikeN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_ReaderResultLikeKind, bound= *ReaderResultLikeN*), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

bind_context_ioresult(function)

Lifts function from `RequiresContextIOResult` for better composition.

In other words, it modifies the function's signature from: `a -> RequiresContextIOResult[env, b, c]` to: `Container[env, a, c] -> Container[env, b, c]`

```
>>> import anyio
>>> from returns.context import (
...     RequiresContextFutureResult,
...     RequiresContextIOResult,
... )
>>> from returns.io import IOSuccess, IOFailure
>>> from returns.pointfree import bind_context_ioresult

>>> def function(arg: int) -> RequiresContextIOResult[str, int, str]:
...     return RequiresContextIOResult(
...         lambda deps: IOSuccess(len(deps) + arg),
...     )

>>> assert anyio.run(bind_context_ioresult(function)(
...     RequiresContextFutureResult.from_value(2),
... )('abc').awaitable) == IOSuccess(5)
>>> assert anyio.run(bind_context_ioresult(function)(
...     RequiresContextFutureResult.from_failure(0),
... )('abc').awaitable) == IOFailure(0)
```

Parameters

function (Callable[[TypeVar(_FirstType)], *RequiresContextIOResult*[TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]] –

Return type

Kinded[Callable[[*KindN*[TypeVar(_ReaderIOResultLikeKind, bound=*ReaderIOResultLikeN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_ReaderIOResultLikeKind, bound=*ReaderIOResultLikeN*), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

`bind_async(function)`

Compose a container and async function returning a container.

In other words, it modifies the function's signature from: `a -> Awaitable[Container[b]]` to: `Container[a] -> Container[b]`

This is how it should be used:

```
>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO
>>> from returns.pointfree import bind_async

>>> async def coroutine(x: int) -> Future[str]:
...     return Future.from_value(str(x + 1))

>>> bound = bind_async(coroutine)(Future.from_value(1))
>>> assert anyio.run(bound.awaitable) == IO('2')
```

Note, that this function works for all containers with `.bind_async` method. See `returns.primitives.interfaces.specific.future.FutureLikeN` for more info.

Parameters

function (Callable[[TypeVar(_FirstType)], Awaitable[KindN[TypeVar(_FutureKind, bound= FutureLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]) –

Return type

Kinded[Callable[[KindN[TypeVar(_FutureKind, bound= FutureLikeN), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], KindN[TypeVar(_FutureKind, bound= FutureLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

bind_awaitable(function)

Composes a container a regular async function.

This function should return plain, non-container value.

In other words, it modifies the function's signature from: `a -> Awaitable[b]` to: `Container[a] -> Container[b]`

This is how it should be used:

```
>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO
>>> from returns.pointfree import bind_awaitable

>>> async def coroutine(x: int) -> int:
...     return x + 1

>>> assert anyio.run(
...     bind_awaitable(coroutine)(Future.from_value(1)).awaitable,
... ) == IO(2)
```

Note, that this function works for all containers with `.bind_awaitable` method. See `returns.primitives.interfaces.specific.future.FutureLikeN` for more info.

Parameters

function (Callable[[TypeVar(_FirstType)], Awaitable[TypeVar(_UpdatedType)]]]) –

Return type

Kinded[Callable[[KindN[TypeVar(_FutureKind, bound= FutureLikeN), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], KindN[TypeVar(_FutureKind, bound= FutureLikeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

bind_optional(function)

Binds a function returning optional value over a container.

In other words, it modifies the function's signature from: `a -> Optional[b]` to: `Container[a] -> Container[b]`

```
>>> from typing import Optional
>>> from returns.pointfree import bind_optional
>>> from returns.maybe import Some, Nothing

>>> def example(argument: int) -> Optional[int]:
...     return argument + 1 if argument > 0 else None
```

(continues on next page)

(continued from previous page)

```
>>> assert bind_optional(example)(Some(1)) == Some(2)
>>> assert bind_optional(example)(Some(0)) == Nothing
>>> assert bind_optional(example)(Nothing) == Nothing
```

Note, that this function works for all containers with `.bind_optional` method. See `returns.primitives.interfaces.specific.maybe._MaybeLikeKind` for more info.

Parameters

function (Callable[[TypeVar(_FirstType)], Optional[TypeVar(_UpdatedType)]])

Return type

Kinded[Callable[[*KindN*[TypeVar(_MaybeLikeKind, bound= *MaybeLikeN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_MaybeLikeKind, bound= *MaybeLikeN*), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

compose_result(function)

Composes inner Result with IOResultLike returning function.

Can be useful when you need an access to both states of the result.

```
>>> from returns.io import IOResult, IOSuccess, IOFailure
>>> from returns.pointfree import compose_result
>>> from returns.result import Result

>>> def modify_string(container: Result[str, str]) -> IOResult[str, str]:
...     return IOResult.from_result(
...         container.map(str.upper).alt(str.lower),
...     )

>>> assert compose_result(modify_string)(
...     IOSuccess('success')
... ) == IOSuccess('SUCCESS')
>>> assert compose_result(modify_string)(
...     IOFailure('FAILURE')
... ) == IOFailure('failure')
```

Parameters

function (Callable[[*Result*[TypeVar(_FirstType), TypeVar(_SecondType)], *KindN*[TypeVar(_IOResultLikeKind, bound= *IOResultLikeN*), TypeVar(_NewFirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]])

Return type

Kinded[Callable[[*KindN*[TypeVar(_IOResultLikeKind, bound= *IOResultLikeN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_IOResultLikeKind, bound= *IOResultLikeN*), TypeVar(_NewFirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]]]

cond(container_type, success_value, error_value=None)

Reduce the boilerplate when choosing paths.

Works with `SingleFailableN` (e.g. `Maybe`) and `DiverseFailableN` (e.g. `Result`).

Example using `cond` with the `Result` container:

```
>>> from returns.pointfree import cond
>>> from returns.result import Failure, Result, Success

>>> assert cond(Result, 'success', 'failure')(True) == Success('success')
>>> assert cond(Result, 'success', 'failure')(False) == Failure('failure')
```

Example using `cond` with the `Maybe` container:

```
>>> from returns.maybe import Maybe, Some, Nothing

>>> assert cond(Maybe, 10.0)(True) == Some(10.0)
>>> assert cond(Maybe, 10.0)(False) == Nothing
```

Parameters

- **container_type** (Union[Type[TypeVar(_SingleFailableKind, *SingleFailableN*)], Type[TypeVar(_DiverseFailableKind, *DiverseFailableN*)]]) – bound=
- **success_value** (TypeVar(_ValueType)) – bound=
- **error_value** (Optional[TypeVar(_ErrorType)]) –

`alt(function)`

Lifts function to be wrapped in a container for better composition.

In other words, it modifies the function's signature from: `a -> b` to: `Container[a] -> Container[b]`

This is how it should be used:

```
>>> from returns.io import IOFailure, IOSuccess
>>> from returns.pointfree import alt

>>> def example(argument: int) -> float:
...     return argument / 2

>>> assert alt(example)(IOSuccess(1)) == IOSuccess(1)
>>> assert alt(example)(IOFailure(4)) == IOFailure(2.0)
```

Note, that this function works for all containers with `.alt` method. See `returns.primitives.interfaces.altable.AltableN` for more info.

Parameters

function (Callable[[TypeVar(_SecondType)], TypeVar(_UpdatedType)]) –

Return type

Kinded[Callable[[*KindN*[TypeVar(_AltableKind, bound=*AltableN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_AltableKind, bound=*AltableN*), TypeVar(_FirstType), TypeVar(_UpdatedType), TypeVar(_ThirdType)]]]

`lash(function)`

Turns function's input parameter from a regular value to a container.

In other words, it modifies the function signature from: `a -> Container[b]` to: `Container[a] -> Container[b]`

Similar to `returns.pointfree.bind()`, but works for failed containers.

This is how it should be used:

```
>>> from returns.pointfree import lash
>>> from returns.result import Success, Failure, Result

>>> def example(argument: int) -> Result[str, int]:
...     return Success(argument + 1)

>>> assert lash(example)(Success('a')) == Success('a')
>>> assert lash(example)(Failure(1)) == Success(2)
```

Note, that this function works for all containers with `.lash` method. See `returns.interfaces.lashable.Lashable` for more info.

Parameters

function (Callable[[TypeVar(_SecondType)], KindN[TypeVar(_LashableKind, bound=*LashableN*), TypeVar(_FirstType), TypeVar(_UpdatedType), TypeVar(_ThirdType)]] –

Return type

Kinded[Callable[[*KindN*[TypeVar(_LashableKind, bound=*LashableN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_LashableKind, bound=*LashableN*), TypeVar(_FirstType), TypeVar(_UpdatedType), TypeVar(_ThirdType)]]]

`unify`(function)

Composes successful container with a function that returns a container.

Similar to `bind()` but has different type. It returns `Result[ValueType, Union[OldErrorType, NewErrorType]]` instead of `Result[ValueType, OldErrorType]`.

So, it can be more useful in some situations. Probably with specific exceptions.

```
>>> from returns.methods import cond
>>> from returns.pointfree import unify
>>> from returns.result import Result, Success, Failure

>>> def bindable(arg: int) -> Result[int, int]:
...     return cond(Result, arg % 2 == 0, arg + 1, arg - 1)

>>> assert unify(bindable)(Success(2)) == Success(3)
>>> assert unify(bindable)(Success(1)) == Failure(0)
>>> assert unify(bindable)(Failure(42)) == Failure(42)
```

Parameters

function (Callable[[TypeVar(_FirstType)], KindN[TypeVar(_DiverseFailableKind, bound=*DiverseFailableN*), TypeVar(_NewFirstType), TypeVar(_NewSecondType), TypeVar(_NewThirdType)]] –

Return type

Kinded[Callable[[*KindN*[TypeVar(_DiverseFailableKind, bound=*DiverseFailableN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]], *KindN*[TypeVar(_DiverseFailableKind, bound=*DiverseFailableN*), TypeVar(_NewFirstType), Union[TypeVar(_SecondType), TypeVar(_NewSecondType)], TypeVar(_NewThirdType)]]]

apply(*container*)

Turns container containing a function into a callable.

In other words, it modifies the function signature from: `Container[a -> b]` to: `Container[a] -> Container[b]`

This is how it should be used:

```
>>> from returns.pointfree import apply
>>> from returns.maybe import Some, Nothing

>>> def example(argument: int) -> int:
...     return argument + 1

>>> assert apply(Some(example))(Some(1)) == Some(2)
>>> assert apply(Some(example))(Nothing) == Nothing
>>> assert apply(Nothing)(Some(1)) == Nothing
>>> assert apply(Nothing)(Nothing) == Nothing
```

Note, that this function works for all containers with `.apply` method. See [returns.interfaces.Applicative](#). [ApplicativeN](#) for more info.

Parameters

container (*KindN*[*TypeVar*(`_ApplicativeKind`), `bound= ApplicativeN`], *Callable*[[*TypeVar*(`_FirstType`)], *TypeVar*(`_UpdatedType`)], *TypeVar*(`_SecondType`), *TypeVar*(`_ThirdType`)] –

Return type

Kinded[[*Callable*[[*KindN*[*TypeVar*(`_ApplicativeKind`), `bound= ApplicativeN`], *TypeVar*(`_FirstType`), *TypeVar*(`_SecondType`), *TypeVar*(`_ThirdType`)]], *KindN*[*TypeVar*(`_ApplicativeKind`), `bound= ApplicativeN`], *TypeVar*(`_UpdatedType`), *TypeVar*(`_SecondType`), *TypeVar*(`_ThirdType`)]]]

9.15 Methods

The following useful methods can be used to interact with interfaces.

9.15.1 cond

Reduce the boilerplate when choosing paths with `DiverseFailableN`. Think of this method as a functional `if` alternative for successful or failed types.

So, this code:

```
>>> from returns.result import Failure, Result, Success

>>> def is_numeric(string: str) -> Result[str, str]:
...     if string.isnumeric():
...         return Success('It is a number')
...     return Failure('It is not a number')
```

Can be replaced with this:

```

>>> from returns.methods import cond
>>> from returns.result import Failure, Result, Success

>>> def is_numeric(string: str) -> Result[str, str]:
...     return cond(
...         Result,
...         string.isnumeric(),
...         'It is a number',
...         'It is not a number',
...     )

>>> assert is_numeric('42') == Success('It is a number')
>>> assert is_numeric('text') == Failure('It is not a number')

```

Why is it helpful? Because `cond` can be easily added into a *Pipelines* of functions.

9.15.2 unwrap_or_failure

Unwraps either a successful or failed value.

```

>>> from returns.io import IO, IOSuccess, IOFailure
>>> from returns.methods import unwrap_or_failure

>>> assert unwrap_or_failure(IOSuccess(1)) == IO(1)
>>> assert unwrap_or_failure(IOFailure('a')) == IO('a')

```

Useful when you have a `ResultLike` value with correctly handled error value, for example with `bimap()`. Here's a full example:

```

>>> from returns.result import Failure, Result, Success
>>> from returns.methods import unwrap_or_failure
>>> from returns.pointfree import bimap

>>> instance: Result[int, str] = Success(1)
>>> error_handled = bimap(lambda inr: inr + 1, lambda _: 0)(instance)
>>> assert isinstance(unwrap_or_failure(error_handled), int)

```

9.15.3 API Reference

`cond(container_type, is_success, success_value, error_value=None)`

Kinded version of `internal_cond()`, use it to infer real return type.

Parameters

- **container_type** (Union[Type[TypeVar(_SingleFailableKind, *SingleFailableN*)], Type[TypeVar(_DiverseFailableKind, *DiverseFailableN*)]]) – bound= bound=
- **is_success** (bool) –
- **success_value** (TypeVar(_ValueType)) –
- **error_value** (Optional[TypeVar(_ErrorType)]) –

`unwrap_or_failure(container)`

Unwraps either successful or failed value.

```
>>> from returns.io import IO, IOSuccess, IOFailure
>>> from returns.methods import unwrap_or_failure

>>> assert unwrap_or_failure(IOSuccess(1)) == IO(1)
>>> assert unwrap_or_failure(IOFailure('a')) == IO('a')
```

Parameters

container (*Unwrappable*[*TypeVar*(*_FirstType*), *TypeVar*(*_SecondType*)])

Return type

Union[*TypeVar*(*_FirstType*), *TypeVar*(*_SecondType*)]

9.16 Do Notation

Note: Technical note: this feature requires *mypy plugin*.

All containers can be easily composed with functions that can take a single argument.

But, what if we need to compose two containers with a function with two arguments? That’s not so easy.

Of course, we can use *Curry* and `.apply` or some imperative code. But, it is not very easy to write and read.

This is why multiple functional languages have a concept of “do-notation”. It allows you to write beautiful imperative code.

9.16.1 Regular containers

Let’s say we have a function called `add` which is defined like this:

```
>>> def add(one: int, two: int) -> int:
...     return one + two
```

And we have two containers: `IO(2)` and `IO(3)`. How can we easily get `IO(5)` in this case?

Luckily, `IO` defines `returns.io.IO.do()` which can help us:

```
>>> from returns.io import IO

>>> assert IO.do(
...     add(first, second)
...     for first in IO(2)
...     for second in IO(3)
... ) == IO(5)
```

Notice, that you don’t have to write any complicated code. Everything is pythonic and readable.

However, we still need to explain what `for` does here. It uses Python’s `__iter__` method which returns an iterable with strictly a single raw value inside.

Warning: Please, don't use `for x in container` outside of `do`-notation. It does not make much sense.

Basically, `for IO(2)` it will return just 2. Then, `IO.do` wraps it into `IO` once again.

Errors

Containers like `Result` and `IOResult` can sometimes represent errors. In this case, `do`-notation expression will return the first found error.

For example:

```
>>> from returns.result import Success, Failure, Result
>>> assert Result.do(
...     first + second
...     for first in Failure('a')
...     for second in Success(3)
... ) == Failure('a')
```

This behavior is consistent with `.map` and other methods.

9.16.2 Async containers

We also support async containers like `Future` and `FutureResult`. It works in a similar way as regular sync containers. But, they require `async` for expressions instead of regular `for` ones. And because of that - they cannot be used outside of `async def` context.

Usage example:

```
>>> import anyio
>>> from returns.future import Future
>>> from returns.io import IO

>>> async def main() -> None:
...     return await Future.do(
...         first + second
...         async for first in Future.from_value(1)
...         async for second in Future.from_value(2)
...     )

>>> assert anyio.run(main) == IO(3)
```

9.16.3 FAQ

Why don't we allow mixing different container types?

One might ask, why don't we allow mixing multiple container types in a single do-notation expression?

For example, this code will not what you expect:

```
>>> from returns.result import Result, Success
>>> from returns.io import IOResult, IOSuccess

>>> assert Result.do(
...     first + second
...     for first in Success(2)
...     for second in IOSuccess(3) # Notice the IO part here
... ) == Success(5)
```

This code will raise a mypy error at `for second in IOSuccess(3)` part:

```
Invalid type supplied in do-notation: expected "returns.result.Result[Any, Any]", got
↪"returns.io.IOSuccess[builtins.int*]"
```

Notice, that the IO part is gone in the final result. This is not right. And we can't track this in any manner. So, we require all containers to have the same type.

The code above must be rewritten as:

```
>>> from returns.result import Success
>>> from returns.io import IOResult, IOSuccess

>>> assert IOResult.do(
...     first + second
...     for first in IOResult.from_result(Success(2))
...     for second in IOSuccess(3)
... ) == IOSuccess(5)
```

Now, it is correct. IO part is safe, the final result is correct. And mypy is happy.

Why don't we allow if conditions in generator expressions?

At the moment, using if conditions inside generator expressions passed into `.do` method is not allowed. Why?

Because if the if condition will return False, we will have an empty iterable and `StopIteration` will be thrown.

```
>>> from returns.io import IO

>>> IO.do(
...     first + second
...     for first in IO(2)
...     for second in IO(3)
...     if second > 10
... )
Traceback (most recent call last):
...
StopIteration
```


It will raise:

```
Using "if" conditions inside a generator is not allowed
```

Instead, use conditions and checks inside your logic, not inside your generator.

Why do we require a literal expression in do-notation?

This code will work in runtime, but will raise a mypy error:

```
>>> from returns.result import Result, Success
>>> expr = (
...     first + second
...     for first in Success(2)
...     for second in Success(3)
... )
>>>
>>> assert Result.do(expr) == Success(5)
```

It raises:

```
Literal generator expression is required, not a variable or function call
```

This happens, because of mypy's plugin API. We need the whole expression to make sure it is correct. We cannot use variables and function calls in its place.

9.16.4 Further reading

- [Do notation in Haskell](#)

9.17 Helper functions

We feature several helper functions to make your developer experience better.

9.17.1 compose

We also ship an utility function to compose two different functions together.

```
>>> from returns.functions import compose
>>> bool_after_int = compose(int, bool)
>>> assert bool_after_int('1') is True
>>> assert bool_after_int('0') is False
```

Composition is also type-safe. The only limitation is that we only support functions with one argument and one return to be composed.

Only works with regular functions (not async).

Warning: `compose` might fail to infer types for some functions. There are several problems: `lambda` and generic functions. In these cases `mypy` will fail to infer the types of the resulting function.

In this case, use `pipe` it does the same thing, but has pretty good type inference. Or use manual annotations with `Callable` type.

9.17.2 identity

We also ship `returns.functions.identity()` function to help you with the composition.

Identity function is a simple concept: it just returns its argument. If you wonder why do we need this function, please read below:

- [Practical Usage of Identity Function \(JS\)](#)
- [Using Identity Functions \(Scala\)](#)

9.17.3 tap and untap

We need `tap()` function to easily compose values with functions that does not return. For example you sometimes need to `print()` values inside your *pipe*:

```
>>> from returns.functions import tap
>>> result = tap(print)(1) # will print and return 1
1
>>> assert result == 1
```

You can also use the `untap` function to turn any function's return type to `None` and still do its thing:

```
>>> from returns.functions import tap, untap
>>> result = untap(tap(print))(1) # will print and return None
1
>>> assert result is None
```

This is also sometimes helpful for a typed function composition.

9.17.4 raise_exception

Sometimes you really want to reraise an exception from `Failure[Exception]` due to some existing API (or a dirty hack).

We allow you to do that with ease!

```
from returns.functions import raise_exception

def create_account_and_user(username: str) -> ...:
    """
    Creates new Account-User pair.

    Imagine, that you need to reraise ValidationErrors due to existing API.
```

(continues on next page)

(continued from previous page)

```

"""
return _validate_user(
    username,
).alt(
    # What happens here is interesting, since you do not let your
    # unwrap to fail with UnwrapFailedError, but instead
    # allows you to reraise a wrapped exception.
    # In this case `ValidationError` will be thrown
    # before `UnwrapFailedError`
    raise_exception,
)
def _validate_user(username: str) -> Result['User', ValidationError]:
    ...

```

Use this with caution. We try to remove exceptions from our code base. Original proposal is [here](#).

9.17.5 not_

With `not_` helper function we can easily deny a function returns. It supports functions with one or more arguments.

```

>>> from typing import List
>>> from returns.functions import compose, not_

>>> def is_even(number: int) -> bool:
...     return number % 2 == 0

>>> def number_is_in_list(number: int, list_: List[int]) -> bool:
...     return number in list_

>>> assert not_(is_even)(2) is False
>>> assert not_(number_is_in_list)(1, [2, 3, 4]) is True
>>> assert compose(int, not_(is_even))("1") is True

```

9.17.6 API Reference

`identity`(*instance*)

Function that returns its argument.

```

>>> assert identity(1) == 1
>>> assert identity([1, 2, 3]) == [1, 2, 3]

```

This function is really helpful for some composition. It is also useful for “do nothing” use-case.

See also:

- https://en.wikipedia.org/wiki/Identity_function
- <https://stackoverflow.com/a/21506571/4842742>

Parameters

instance (TypeVar(*_FirstType*)) –

Return type

TypeVar(_FirstType)

compose(*first*, *second*)

Allows function composition.

Works as: `second . first` or `first() |> second()`. You can read it as “second after first”.

```
>>> assert compose(float, int>('123.5') == 123
```

We can only compose functions with one argument and one return. Type checked.

Parameters

- **first** (Callable[[TypeVar(_FirstType)], TypeVar(_SecondType)]) –
- **second** (Callable[[TypeVar(_SecondType)], TypeVar(_ThirdType)]) –

Return type

Callable[[TypeVar(_FirstType)], TypeVar(_ThirdType)]

tap(*function*)

Allows to apply some function and return an argument, instead of a result.

Is useful for composing functions with side-effects like `print()`, `logger.log()`, etc.

```
>>> assert tap(print)(1) == 1
1
>>> assert tap(lambda _: 1)(2) == 2
```

See also:

- <https://github.com/dry-python/returns/issues/145>

Parameters**function** (Callable[[TypeVar(_FirstType)], Any]) –**Return type**

Callable[[TypeVar(_FirstType)], TypeVar(_FirstType)]

untap(*function*)

Allows to apply some function and always return None as a result.

Is useful for composing functions that do some side effects and return some nonsense.

Is the kind of a reverse of the `tap` function.

```
>>> def strange_log(arg: int) -> int:
...     print(arg)
...     return arg

>>> assert untap(strange_log)(2) is None
2
>>> assert untap(tap(lambda _: 1))(2) is None
```

See also:

- <https://github.com/dry-python/returns/issues/145>

Parameters

function (Callable[[TypeVar(_FirstType)], Any]) –

Return type

Callable[[TypeVar(_FirstType)], None]

raise_exception(*exception*)

Helper function to raise exceptions as a function.

It might be required as a compatibility tool for existing APIs. That's how it can be used:

```
>>> from returns.result import Failure, Result
>>> # Some operation result:
>>> user: Result[int, ValueError] = Failure(ValueError('boom'))

>>> # Here we unwrap internal exception and raise it:
>>> user.alt(raise_exception)
Traceback (most recent call last):
...
ValueError: boom
```

See also:

- <https://github.com/dry-python/returns/issues/56>

Parameters

exception (Exception) –

Return type

NoReturn

not_(function)

Denies the function returns.

```
>>> from returns.result import Result, Success, Failure

>>> def is_successful(result_container: Result[float, int]) -> bool:
...     return isinstance(result_container, Result.success_type)

>>> assert not_(is_successful)(Success(1.0)) is False
>>> assert not_(is_successful)(Failure(1)) is True
```

Parameters

function (Callable[[ParamSpec(_FuncParams)], bool]) –

Return type

Callable[[ParamSpec(_FuncParams)], bool]

9.18 Curry

This module is dedicated to partial application.

We support two types of partial application: `@curry` and `partial`.

`@curry` is a new concept for most Python developers, but Python already has a great tool to use partial application: `functools.partial`

The only problem with it is the lack of typing. Let's see what problems do we solve with this module.

Warning: This module requires *our mypy plugin* to be present. Without it we will fallback to the original behaviour.

9.18.1 Partial

Here's how typing works there:

```
from functools import partial

def some_function(first: int, second: int) -> float:
    return first / second

reveal_type(partial(some_function, 1))
# => functools.partial[builtins.float*]
# => Which is really: `def (*Any, **Any) -> builtins.float`
```

And compare it with our solution:

```
from returns.curry import partial

def some_function(first: int, second: int) -> float:
    return first / second

reveal_type(partial(some_function, 1))
# => def (second: builtins.int) -> builtins.float*
# => Which is fair!
```

Note: We still use `functools.partial` inside. We just improve the typings.

Generics

One more problem is generics support in `functools.partial`. Here's the comparison:

```
from functools import partial
from typing import List, TypeVar

T = TypeVar('T')

x: List[int]
```

(continues on next page)

(continued from previous page)

```
def some_function(first: List[T], second: int) -> T:
    return first[second]

reveal_type(partial(some_function, x))
# => functools.partial[T`-1]
# => Which is broken!
```

And our solution works fine:

```
from returns.curry import partial

reveal_type(partial(some_function, x))
# => def (second: builtins.int) -> builtins.int*
```

We also work with complex generic with multiple arguments or with multiple generics.

The only known problem is that passing explicit generic like `[1, 2, 3]` will resolve in `List[Any]`. Because mypy won't be able to infer this type for some reason.

The reasonable work-around is to pass annotated variables like in the example above.

Types and Instances

We can also work with types and instances. Because they are callable too!

```
from returns.curry import partial

class Test(object):
    def __init__(self, arg: int) -> None:
        self.arg = arg

    def __call__(self, other: int) -> int:
        return self.arg + other

reveal_type(partial(Test, 1)) # N: Revealed type is 'def () -> ex.Test'
reveal_type(partial(Test(1), 1)) # N: Revealed type is 'def () -> builtins.int'
```

No differences with regular callables at all.

Overloads

We also support working with `@overload` definitions. It also looks the same way:

```
from typing import overload
from returns.curry import partial

@overload
def test(a: int, b: str) -> str:
    ...

@overload
```

(continues on next page)

(continued from previous page)

```

def test(a: int) -> int:
    ...

@overload
def test(a: str) -> None: # won't match!
    ...

def test(a, b=None):
    ...

reveal_type(partial(test, 1)) # N: Revealed type is 'Overload(def (b: builtins.str) ->
↳builtins.str, def () -> builtins.int)'

```

From this return type you can see that we work with all matching cases and discriminate unmatching ones.

9.18.2 @curry

curry allows to provide only a subset of arguments to a function. And it won't be called until all the required arguments are provided.

In contrast to `partial` which works on the calling stage, `@curry` works best when defining a new function.

```

>>> from returns.curry import curry

>>> @curry
... def function(first: int, second: str) -> bool:
...     return len(second) > first

>>> assert function(1)('a') is False
>>> assert function(1, 'a') is False
>>> assert function(2)('abc') is True
>>> assert function(2, 'abc') is True

```

Take a note, that providing invalid arguments will raise `TypeError`:

```

>>> function(1, 2, 3)
Traceback (most recent call last):
...
TypeError: too many positional arguments

>>> function(a=1)
Traceback (most recent call last):
...
TypeError: got an unexpected keyword argument 'a'

```

This is really helpful when working with `.apply()` method of containers.

Warning: We recommend using `partial` instead of `@curry` when possible because it's much faster.

Typing

@curry functions are also fully typed with our custom mypy plugin.

Let's see how types do look like for a curried function:

```
>>> from returns.curry import curry

>>> @curry
... def zero(a: int, b: float, *, kw: bool) -> str:
...     return str(a - b) if kw else ''

>>> assert zero(1)(0.3)(kw=True) == '0.7'
>>> assert zero(1)(0.3, kw=False) == ''

# If we will reveal the type it would be quite big:

reveal_type(zero)

# Overload(
#   def (a: builtins.int) -> Overload(
#     def (b: builtins.float, *, kw: builtins.bool) -> builtins.str,
#     def (b: builtins.float) -> def (*, kw: builtins.bool) -> builtins.str
#   ),
#   def (a: builtins.int, b: builtins.float) -> def (*, kw: builtins.bool)
#     -> builtins.str,
#   def (a: builtins.int, b: builtins.float, *, kw: builtins.bool)
#     -> builtins.str
# )
```

It reveals to us that there are 4 possible way to call this function. And we type all of them with `overload` type.

When you provide any arguments, you discriminate some overloads and choose more specific path:

```
reveal_type(zero(1, 2.0))
# By providing this set of arguments we have chosen this path:
#
#   def (a: builtins.int, b: builtins.float) -> def (*, kw: builtins.bool)
#     -> builtins.str,
#
# And the revealed type would be:
#
#   def (*, kw: builtins.bool) -> builtins.str
#
```

It works with functions, instance, class, and static methods, including generics. See `Limitations` in the API Reference.

9.18.3 FAQ

Why don't you support `*` and `**` arguments?

When you use `partial(some, *my_args)` or `partial(some, **my_args)` or both of them at the same time, we fallback to the default return type. The same happens with `curry`. Why?

There are several problems:

- Because mypy cannot not infer what arguments are there inside this `my_args` variable
- Because `curry` cannot know when to stop accepting `*args` and `**kwargs`
- And there are possibly other problems!

Our advice is not to use `*args` and `*kwargs` with `partial` and `curry`.

But, it is still possible, but in this case we will fallback to `Any`.

9.18.4 Further reading

- `functools.partial`
- Currying
- `@curry` decorator

9.18.5 API Reference

`partial`(*func*, **args*, ***kwargs*)

Typed partial application.

It is just a `functools.partial` wrapper with better typing support.

We use a custom mypy plugin to make sure types are correct. Otherwise, it is currently impossible to properly type this function.

```
>>> from returns.curry import partial

>>> def sum_two_numbers(first: int, second: int) -> int:
...     return first + second

>>> sum_with_ten = partial(sum_two_numbers, 10)
>>> assert sum_with_ten(2) == 12
>>> assert sum_with_ten(-5) == 5
```

See also:

- <https://docs.python.org/3/library/functools.html#functools.partial>

Parameters

- **`func`** (`Callable[... , TypeVar(_ReturnType)]`) –
- **`args`** (`Any`) –
- **`kwargs`** (`Any`) –

Return type

Callable[... , TypeVar(_ReturnType)]

curry(function)

Typed currying decorator.

Currying is a conception from functional languages that does partial applying. That means that if we pass one argument in a function that gets 2 or more arguments, we'll get a new function that remembers all previously passed arguments. Then we can pass remaining arguments, and the function will be executed.

`partial()` function does a similar thing, but it does partial application exactly once. `curry` is a bit smarter and will do partial application until enough arguments passed.

If wrong arguments are passed, `TypeError` will be raised immediately.

We use a custom mypy plugin to make sure types are correct. Otherwise, it is currently impossible to properly type this function.

```
>>> from returns.curry import curry

>>> @curry
... def divide(number: int, by: int) -> float:
...     return number / by

>>> divide(1) # doesn't call the func and remembers arguments
<function divide at ...>
>>> assert divide(1)(by=10) == 0.1 # calls the func when possible
>>> assert divide(1)(10) == 0.1 # calls the func when possible
>>> assert divide(1, by=10) == 0.1 # or call the func like always
```

Here are several examples with wrong arguments:

```
>>> divide(1, 2, 3)
Traceback (most recent call last):
...
TypeError: too many positional arguments

>>> divide(a=1)
Traceback (most recent call last):
...
TypeError: got an unexpected keyword argument 'a'
```

Limitations:

- It is kinda slow. Like 100 times slower than a regular function call.
- It does not work with several builtins like `str`, `int`, and possibly other C defined callables
- `*args` and `**kwargs` are not supported and we use `Any` as a fallback
- Support of arguments with default values is very limited, because we cannot be totally sure which case we are using: with the default value or without it, be careful
- We use a custom mypy plugin to make types correct, otherwise, it is currently impossible
- It might not work as expected with `curried Klass().method`, it might generate invalid method signature (looks like a bug in mypy)
- It is probably a bad idea to `curry` a function with lots of arguments, because you will end up with lots of overload functions, that you won't be able to understand. It might also be slow during the typecheck

- Currying of `__init__` does not work because of the bug in `mypy`: <https://github.com/python/mypy/issues/8801>

We expect people to use this tool responsibly when they know that they are doing.

See also:

- <https://en.wikipedia.org/wiki/Currying>
- <https://stackoverflow.com/questions/218025/>

Parameters

`function` (Callable[... , TypeVar(_ReturnType)]) –

Return type

Callable[... , TypeVar(_ReturnType)]

9.19 Trampolines

Python does not support TCO (tail call optimization), so any recursion-based algorithms become dangerous.

We cannot be sure that they won't cause `RecursionError` on deeply nested data.

Here's why we need trampolines: they allow to replicate tail call optimization by wrapping function calls into `returns.trampolines.Trampoline` objects, making recursion-based function *always* safe.

Example:

```
>>> from typing import Union, List
>>> from returns.trampolines import Trampoline, trampoline

>>> @trampoline
... def accumulate(
...     numbers: List[int],
...     acc: int = 0,
... ) -> Union[int, Trampoline[int]]:
...     if not numbers:
...         return acc
...     number = number = numbers.pop()
...     return Trampoline(accumulate, numbers, acc + number)

>>> assert accumulate([1, 2]) == 3
>>> assert accumulate([1, 2, 3]) == 6
```

The following function is still fully type-safe: - `Trampoline` object uses `ParamSpec` to be sure that passed arguments are correct - Final return type of the function is narrowed to contain only an original type (without `Trampoline` implementation detail)

9.19.1 API Reference

final class Trampoline(*func*, /, **args*, ***kwargs*)

Bases: `Generic[_ReturnType]`

Represents a wrapped function call.

Primitive to convert recursion into an actual object.

Parameters

- **func** (`Callable[[ParamSpec(_FuncParams)], TypeVar(_ReturnType)]`) –
- **args** (`ParamSpecArgs`) –
- **kwargs** (`ParamSpecKwargs`) –

func

args

kwargs

trampoline(*func*)

Convert functions using recursion to regular functions.

Trampolines allow to unwrap recursion into a regular `while` loop, which does not raise any `RecursionError` ever.

Since python does not have TCO (tail call optimization), we have to provide this helper.

This is done by wrapping real function calls into `returns.trampolines.Trampoline` objects:

```
>>> from typing import Union
>>> from returns.trampolines import Trampoline, trampoline

>>> @trampoline
... def get_factorial(
...     for_number: int,
...     current_number: int = 0,
...     acc: int = 1,
... ) -> Union[int, Trampoline[int]]:
...     assert for_number >= 0
...     if for_number <= current_number:
...         return acc
...     return Trampoline(
...         get_factorial,
...         for_number,
...         current_number=current_number + 1,
...         acc=acc * (current_number + 1),
...     )

>>> assert get_factorial(0) == 1
>>> assert get_factorial(3) == 6
>>> assert get_factorial(4) == 24
```

See also:

- eli.thegreenplace.net/2017/on-recursion-continuations-and-trampolines

- https://en.wikipedia.org/wiki/Tail_call

Parameters

```
func (Callable[[ParamSpec(_FuncParams)], Union[TypeVar(_ReturnType),
Trampoline[TypeVar(_ReturnType)]]]) -
```

Return type

```
Callable[[ParamSpec(_FuncParams)], TypeVar(_ReturnType)]
```

9.20 Primitive types

We have several utility types that we use for our containers, that can also help end users as well.

9.20.1 Fold

You can use all power of declarative loops in your app with Fold.

```
>>> from returns.iterables import Fold
>>> from returns.io import IO

>>> items = [IO(1), IO(2), IO(3)]
>>> assert Fold.loop(
...     items,
...     IO(''),
...     lambda num: lambda text: text + str(num),
... ) == IO('123')
```

There are also other helpful methods as well. See *returns.iterables.AbstractFold*.

We also ship *AbstractFold*, where you can change how `loop` (or any other) method works. For example, for performance reasons.

Let's say you have a big number of *RequiresContext* instances and you want to do the same thing string concatenation we have shown above.

You might face recursion problems with it:

```
>>> import sys
>>> from returns.context import Reader
>>> from returns.iterables import Fold

>>> items = [Reader.from_value(num) for num in range(sys.getrecursionlimit())]
>>> Fold.loop(items, Reader.from_value(0), lambda x: lambda y: x + y)(...)
Traceback (most recent call last):
...
RecursionError: ...
```

So, let's change how it works for this specific type:

```
>>> from returns.iterables import AbstractFold

>>> class ContextAwareFold(AbstractFold):
...     @classmethod
```

(continues on next page)

(continued from previous page)

```

...     def _loop(cls, iterable, acc, function, concat, deps=None):
...         wrapped = acc.from_value(function)
...         for current in iterable:
...             assert isinstance(current, Reader)
...             acc = Reader.from_value(concat(current, acc, wrapped)(deps))
...         return acc

```

Note: Don't forget to add typing annotations to your real code! This is just an example.

And now let's test that it works without recursion:

```

>>> items = [Reader.from_value(num) for num in range(sys.getrecursionlimit())]
>>> assert ContextAwareFold.loop(
...     items, Reader.from_value(0), lambda x: lambda y: x + y,
... )(...) == sum(range(sys.getrecursionlimit()))

```

And no error will be produced! We now don't use recursion inside. Consider this way of doing things as a respected hack.

9.20.2 Immutable

This class is useful when you need to make some instances immutable (like *our containers are immutable*).

9.20.3 API Reference

Iterables

class AbstractFold

Bases: object

A collection of different helpers to write declarative `Iterable` actions.

Allows to work with iterables.

Implementation

`AbstractFold` and `Fold` types are special. They have double definition for each method: public and protected ones. Why?

Because you cannot override `@kinded` method due to a mypy bug. So, there are two opportunities for us here:

1. Declare all method as `@final` and do not allow to change anything
2. Use delegation to protected unkinded methods

We have chosen the second way! Here's how it works:

1. Public methods are `@kinded` for better typing and cannot be overridden
2. Protected methods are unkinded and can be overridden in subclassing

Now, if you need to make a change into our implementation, then you can subclass `Fold` or `AbstractFold` and then change an implementation of any unkinded protected method.

final classmethod `loop(iterable, acc, function)`

Allows to make declarative loops for any `ApplicativeN` subtypes.

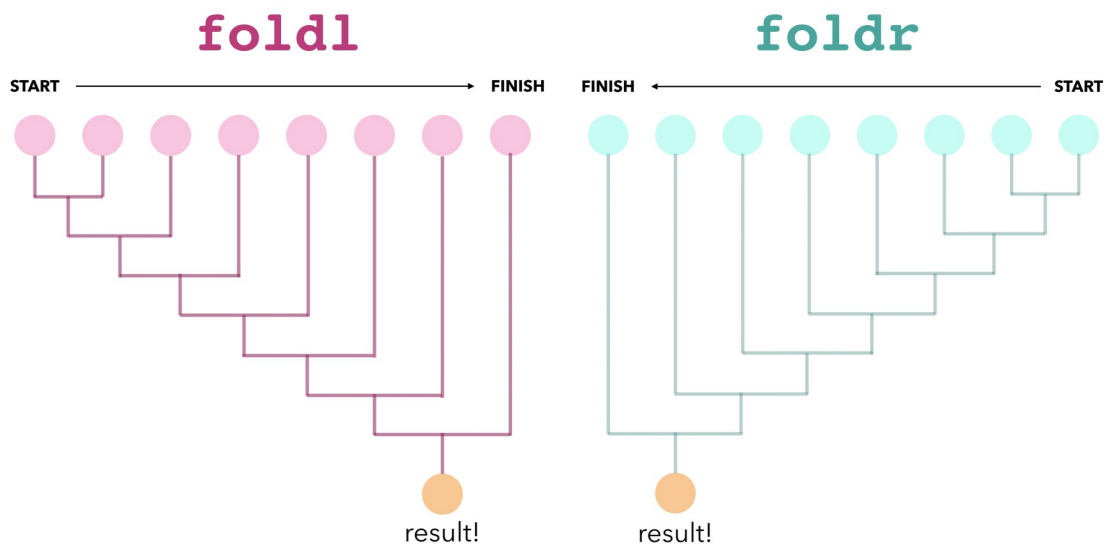
Quick example:

```
>>> from typing import Callable
>>> from returns.maybe import Some
>>> from returns.iterables import Fold

>>> def sum_two(first: int) -> Callable[[int], int]:
...     return lambda second: first + second

>>> assert Fold.loop(
...     [Some(1), Some(2), Some(3)],
...     Some(10),
...     sum_two,
... ) == Some(16)
```

Looks like `foldl` in some other languages with some more specifics. See: https://philippschwarz.dev/fpilluminated/?page_id=348#bwg3/137



Is also quite similar to `reduce`.

Public interface for `_loop` method. Cannot be modified directly.

Parameters

- **iterable** (`Iterable[KindN[TypeVar(_ApplicativeKind, bound= ApplicativeN), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]]`) –
- **acc** (`(KindN[TypeVar(_ApplicativeKind, bound= ApplicativeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)])`) –
- **function** (`Callable[[TypeVar(_FirstType)], Callable[[TypeVar(_UpdatedType), TypeVar(_UpdatedType)]]]`) –

Return type

`KindN[TypeVar(_ApplicativeKind, bound= ApplicativeN), TypeVar(_UpdatedType), TypeVar(_SecondType), TypeVar(_ThirdType)]`

final classmethod collect(*iterable*, *acc*)

Transforms an iterable of containers into a single container.

Quick example for regular containers:

```
>>> from returns.io import IO
>>> from returns.iterables import Fold

>>> items = [IO(1), IO(2)]
>>> assert Fold.collect(items, IO(())) == IO((1, 2))
```

If container can have failed values, then this strategy fails on any existing failed like type.

It is enough to have even a single failed value in iterable for this type to convert the whole operation result to be a failure. Let's see how it works:

```
>>> from returns.result import Success, Failure
>>> from returns.iterables import Fold

>>> empty = []
>>> all_success = [Success(1), Success(2), Success(3)]
>>> has_failure = [Success(1), Failure('a'), Success(3)]
>>> all_failures = [Failure('a'), Failure('b')]

>>> acc = Success() # empty tuple

>>> assert Fold.collect(empty, acc) == Success()
>>> assert Fold.collect(all_success, acc) == Success((1, 2, 3))
>>> assert Fold.collect(has_failure, acc) == Failure('a')
>>> assert Fold.collect(all_failures, acc) == Failure('a')
```

If that's now what you need, check out `collect_all()` to force collect all non-failed values.

Public interface for `_collect` method. Cannot be modified directly.

Parameters

- **iterable** (Iterable[KindN[TypeVar(_ApplicativeKind, bound= *ApplicativeN*), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]] –
- **acc** (KindN[TypeVar(_ApplicativeKind, bound= *ApplicativeN*), Tuple[TypeVar(_FirstType), ..., TypeVar(_SecondType), TypeVar(_ThirdType)]] –

Return type

KindN[TypeVar(_ApplicativeKind, bound= *ApplicativeN*), Tuple[TypeVar(_FirstType), ..., TypeVar(_SecondType), TypeVar(_ThirdType)]]

final classmethod collect_all(*iterable*, *acc*)

Transforms an iterable of containers into a single container.

This method only works with `FailableN` subtypes, not just any `ApplicativeN` like `collect()`.

Strategy to extract all successful values even if there are failed values.

If there's at least one successful value and any amount of failed values, we will still return all collected successful values.

We can return failed value for this strategy only in a single case: when default element is a failed value.

Let's see how it works:

```

>>> from returns.result import Success, Failure
>>> from returns.iterables import Fold

>>> empty = []
>>> all_success = [Success(1), Success(2), Success(3)]
>>> has_failure = [Success(1), Failure('a'), Success(3)]
>>> all_failures = [Failure('a'), Failure('b')]

>>> acc = Success(()) # empty tuple

>>> assert Fold.collect_all(empty, acc) == Success(())
>>> assert Fold.collect_all(all_success, acc) == Success((1, 2, 3))
>>> assert Fold.collect_all(has_failure, acc) == Success((1, 3))
>>> assert Fold.collect_all(all_failures, acc) == Success(())
>>> assert Fold.collect_all(empty, Failure('c')) == Failure('c')

```

If that's now what you need, check out `collect()` to collect only successful values and fail on any failed ones.

Public interface for `_collect_all` method. Cannot be modified directly.

Parameters

- **iterable** (`Iterable[KindN[TypeVar(_FailableKind, bound= FailableN), TypeVar(_FirstType), TypeVar(_SecondType), TypeVar(_ThirdType)]]`) –
- **acc** (`KindN[TypeVar(_FailableKind, bound= FailableN), Tuple[TypeVar(_FirstType), ..., TypeVar(_SecondType), TypeVar(_ThirdType)]]`) –

Return type

`KindN[TypeVar(_FailableKind, bound= FailableN), Tuple[TypeVar(_FirstType), ..., TypeVar(_SecondType), TypeVar(_ThirdType)]]`

class Fold

Bases: `AbstractFold`

Concrete implementation of `AbstractFold` of end users.

Use it by default.

Types

class Immutable

Bases: `object`

Helper type for objects that should be immutable.

When applied, each instance becomes immutable. Nothing can be added or deleted from it.

```

>>> from returns.primitives.types import Immutable
>>> class MyModel(Immutable):
...     ...

>>> model = MyModel()
>>> model.prop = 1
Traceback (most recent call last):

```

(continues on next page)

(continued from previous page)

```
...
returns.primitives.exceptions.ImmutableStateError
```

See `returns.primitives.container.BaseContainer` for examples.

Exceptions

exception `UnwrapFailedError(container)`

Bases: `Exception`

Raised when a container can not be unwrapped into a meaningful value.

Parameters

`container` (*Unwrappable*) –

`halted_container`

exception `ImmutableStateError`

Bases: `AttributeError`

Raised when a container is forced to be mutated.

It is a subclass of `AttributeError` for two reasons:

1. It seems kinda reasonable to expect `AttributeError` on attribute modification
2. It is used inside `typing.py` this way, we do have several typing features that requires that behaviour

See: <https://github.com/dry-python/returns/issues/394>

9.21 Development tooling

9.21.1 Tracing Failures

Sometimes we want to trace where the `Failure` has occurred in our system, `returns` provide a way to trace those failures. By default tracing is disabled.

The trace is accessible by `trace` property that is available for `Result`, `IOResult` containers. It's basically a list containing all `inspect.FrameInfo` objects from the call stack when the `Failure` was originally created.

To enable it you can use `collect_traces`. See examples below:

You can use it as a context manager:

```
>>> from inspect import FrameInfo
>>> from returns.result import Failure, Result
>>> from returns.primitives.tracing import collect_traces
>>> def get_failure(argument: str) -> Result[str, str]:
...     return Failure(argument)
>>> non_traced_failure = get_failure('Normal Failure')
>>> with collect_traces():
...     traced_failure = get_failure('Traced Failure')
```

(continues on next page)

(continued from previous page)

```

>>> assert non_traced_failure.trace is None
>>> assert isinstance(traced_failure.trace, list)
>>> assert all(isinstance(trace_line, FrameInfo) for trace_line in traced_failure.trace)

>>> for trace_line in traced_failure.trace:
...     print(f"{trace_line.filename}:{trace_line.lineno} in `{trace_line.function}`")
...
/returns/returns/result.py:529 in `Failure`
/example_folder/example.py:5 in `get_failure`
/example_folder/example.py:1 in ``

```

Or as a decorator:

```

>>> from inspect import FrameInfo

>>> from returns.io import IOFailure, IOResult
>>> from returns.result import Failure, Result
>>> from returns.primitives.tracing import collect_traces

>>> @collect_traces
... def traced_function(value: str) -> IOResult[str, str]:
...     return IOFailure(value)

>>> non_traced_failure = Failure('Normal Failure')
>>> traced_failure = traced_function('Traced Failure')

>>> assert non_traced_failure.trace is None
>>> assert isinstance(traced_failure.trace, list)
>>> assert all(isinstance(trace_line, FrameInfo) for trace_line in traced_failure.trace)

>>> for trace_line in traced_failure.trace:
...     print(f"{trace_line.filename}:{trace_line.lineno} in `{trace_line.function}`")
...
/returns/returns/result.py:525 in `Failure`
/returns/returns/io.py:852 in `IOFailure`
/example_folder/example.py:7: in `traced_function`
/usr/lib/python3.8/contextlib.py:75 in `inner`
/example_folder/example.py:1 in ``

```

Warning: Activating trace can make your program noticeably slower if it has many points where `Failure` is often created.

Warning: `collect_traces` is not thread safe, beware to use it with threading!

Warning: Traces are meant to be used during development only.

9.21.2 API Reference

`collect_traces`(*function=None*)

Context Manager/Decorator to active traces collect to the Failures.

```
>>> from inspect import FrameInfo
>>> from returns.io import IOResult
>>> from returns.result import Result
>>> from returns.primitives.tracing import collect_traces

>>> with collect_traces():
...     traced_failure = Result.from_failure('Traced Failure')
>>> non_traced_failure = IOResult.from_failure('Non Traced Failure')

>>> assert non_traced_failure.trace is None
>>> assert isinstance(traced_failure.trace, list)
>>> assert all(isinstance(trace_line, FrameInfo) for trace_line in traced_failure.
↳trace)

>>> for trace_line in traced_failure.trace:
...     print(
...         '{0}:{1} in `{2}`'.format(
...             trace_line.filename,
...             trace_line.lineno,
...             trace_line.function,
...         ),
...     )
...
/returns/returns/result.py:525 in `Failure`
/returns/returns/result.py:322 in `from_failure`
/example_folder/example.py:1 in ``
```

Parameters

function (Optional[TypeVar(_FunctionType, bound= Callable)]) –

Return type

Union[TypeVar(_FunctionType, bound= Callable), ContextManager[None]]

9.22 mypy plugin

We provide a custom mypy plugin to fix existing issues, provide new awesome features, and improve type-safety of things developers commonly use.

9.22.1 Installation

You will need to install mypy separately. It is not bundled with returns.

To install any mypy plugin add it to the plugins section of the config file (setup.cfg or mypy.ini):

```
[mypy]
plugins =
    returns.contrib.mypy.returns_plugin
```

Or in pyproject.toml:

```
[tool.mypy]
plugins = ["returns.contrib.mypy.returns_plugin"]
```

We recommend to always add our plugin as the first one in chain.

9.22.2 Configuration

You can have a look at the suggested mypy configuration in our own repository.

You can also use nitpick tool to enforce the same mypy configuration for all your projects.

We recommend to use our own setup. Add this to your pyproject.toml:

```
[tool.nitpick]
style = "https://raw.githubusercontent.com/wemake-services/wemake-python-styleguide/
↪master/styles/mypy.toml"
```

And use flake8 to lint that configuration defined in the setup matches yours. This will allow to keep them in sync with the upstream.

9.22.3 Supported features

- kind feature adds Higher Kinded Types (HKT) support
- curry feature allows to write typed curried functions
- partial feature allows to write typed partial application
- flow feature allows to write better typed functional pipelines with flow function
- pipe feature allows to write better typed functional pipelines with pipe function
- do-notation feature allows using *Do Notation*

9.22.4 Further reading

- [Mypy official docs](#)
- [Mypy plugins docs](#)

9.22.5 API Reference

Plugin definition

TYPED_PARTIAL_FUNCTION: Final = 'returns.curry.partial'

Used for typed partial function.

TYPED_CURRY_FUNCTION: Final = 'returns.curry.curry'

Used for typed curry decorator.

TYPED_FLOW_FUNCTION: Final = 'returns._internal.pipeline.flow.flow'

Used for typed flow call.

TYPED_PIPE_FUNCTION: Final = 'returns._internal.pipeline.pipe.pipe'

Used for typed pipe call.

TYPED_KINDN: Final = 'returns.primitives.hkt.KindN'

Used for HKT emulation.

DO_NOTATION_METHODS: Final = ('returns.io.IO.do', 'returns.maybe.Maybe.do',
'returns.future.Future.do', 'returns.result.Result.do', 'returns.io.IOResult.do',
'returns.future.FutureResult.do')

Used for *Do Notation*.

Custom mypy plugin to solve the temporary problem with python typing.

Important: we don't do anything ugly here. We only solve problems of the current typing implementation.

mypy API docs are here: https://mypy.readthedocs.io/en/latest/extending_mypy.html

We use `pytest-mypy-plugins` to test that it works correctly, see: <https://github.com/mkurnikov/pytest-mypy-plugins>

plugin(*version*)

Plugin's public API and entrypoint.

Parameters

version (str) –

Return type

Type[Plugin]

Kind

attribute_access(ctx)

Ensures that attribute access to KindN is correct.

In other words:

```
from typing import TypeVar
from returns.primitives.hkt import KindN
from returns.interfaces.mappable import MappableN

_MappableType = TypeVar('_MappableType', bound=MappableN)

kind: KindN[_MappableType, int, int, int]
reveal_type(kind.map) # will work correctly!
```

Parameters

ctx (AttributeContext) –

Return type

Type

dekind(ctx)

Infers real type behind Kind form.

Basically, it turns Kind[IO, int] into IO[int]. The only limitation is that it works with only Instance type in the first type argument position.

So, dekind(KindN[T, int]) will fail.

Parameters

ctx (FunctionContext) –

Return type

Type

kinded_signature(ctx)

Returns the internal function wrapped as Kinded[def].

Works for Kinded class when `__call__` magic method is used. See [returns.primitives.hkt.Kinded](#) for more information.

Parameters

ctx (MethodSigContext) –

Return type

CallableType

kinded_call(ctx)

Reveals the correct return type of Kinded.`__call__` method.

Turns `-> KindN[I, t1, t2, t3]` into `-> I[t1, t2, t3]`.

Also strips unused type arguments for KindN, so: `- KindN[IO, int, <nothing>, <nothing>]` will be `IO[int]` - `KindN[Result, int, str, <nothing>]` will be `Result[int, str]`

It also processes nested KindN with recursive strategy.

See [returns.primitives.hkt.Kinded](#) for more information.

Parameters**ctx** (MethodContext) –**Return type**

Type

kinded_get_descriptor(*ctx*)

Used to analyze @kinded method calls.

We do this due to __get__ descriptor magic.

Parameters**ctx** (MethodContext) –**Return type**

Type

Curry**analyze**(*ctx*)

Returns proper type for curried functions.

Parameters**ctx** (FunctionContext) –**Return type**

Type

Partial**analyze**(*ctx*)This hook is used to make typed currying a thing in *returns* project.

This plugin is a temporary solution to the problem. It should be later replaced with the official way of doing things. One day functions will have better API and we plan to submit this plugin into mypy core plugins, so it would not be required.

Internally we just reduce the original function's argument count. And drop some of them from function's signature.

Parameters**ctx** (FunctionContext) –**Return type**

Type

Flow**analyze**(*ctx*)

Helps to analyze flow function calls.

By default, mypy cannot infer and check this function call:

```
>>> from returns.pipeline import flow
>>> assert flow(
...     1,
```

(continues on next page)

```
...     lambda x: x + 1,
...     lambda y: y / 2,
... ) == 1.0
```

But, this plugin can! It knows all the types for all lambda functions in the pipeline. How?

1. We use the first passed parameter as the first argument to the first passed function
2. We use parameter + function to check the call and reveal types of current pipeline step
3. We iterate through all passed function and use previous return type as a new parameter to call current function

Parameters

ctx (FunctionContext) –

Return type

Type

Pipe

Typing pipe functions requires several phases.

It is pretty obvious from its usage:

1. When we pass a sequence of functions we have to reduce the final callable type, it is require to match the callable protocol. And at this point we also kinda try to check that all pipeline functions do match, but this is impossible to do 100% correctly at this point, because generic functions don't have a type argument to infer the final result
2. When we call the function, we need to check for two things. First, we check that passed argument fits our instance requirement. Second, we check that pipeline functions match. Now we have all arguments to do the real inference.
3. We also need to fix generic in method signature. It might be broken, because we add new generic arguments and return type. So, it is safe to reattach generic back to the function.

Here's when it works:

```
>>> from returns.pipeline import pipe

>>> def first(arg: int) -> bool:
...     return arg > 0
>>> def second(arg: bool) -> str:
...     return 'bigger' if arg else 'not bigger'

>>> pipeline = pipe(first, second) # `analyzed` is called
>>> assert pipeline(1) == 'bigger' # `signature and `infer` are called
>>> assert pipeline(0) == 'not bigger' # `signature and `infer` again
```

analyze(ctx)

This hook helps when we create the pipeline from sequence of funcs.

Parameters

ctx (FunctionContext) –

Return type

Type

infer(*ctx*)

This hook helps when we finally call the created pipeline.

Parameters

ctx (MethodContext) –

Return type

Type

signature(*ctx*)

Helps to fix generics in method signature.

Parameters

ctx (MethodSigContext) –

Return type

CallableType

Do notation**analyze**(*ctx*)

Used to handle validation and error types in *Do Notation*.

What it does?

1. For all types we ensure that only a single container type is used in a single do-notation. We don't allow mixing them.
2. For types with error types (like `Result`), it infers what possible errors types can we have. The result is a `Union` of all possible errors.
3. Ensures that expression passed into `.do` method is literal.
4. Checks that default value is provided if generator expression has `if` conditions inside.

Parameters

ctx (MethodContext) –

Return type

Type

9.23 pytest plugin

We use special pytest plugin to improve the testing side of this project.

For example: it is a popular request to ensure that your container does have its error pass handled. Because otherwise, developers might forget to do it properly. It is impossible to fix with types, but is really simple to check with tests.

9.23.1 Installation

You will need to install `pytest` separately.

9.23.2 Usage

There's no need to install anything special. `pytest` will automatically find and use this plugin.

To use it in your tests, request `returns` fixture like so:

```
def test_my_container(returns):
    ...
```

`assert_equal`

We have a special helper to compare containers' equality.

It might be an easy task for two `Result` or `Maybe` containers, but it is not very easy for two `ReaderResult` or `FutureResult` instances.

Take a look:

```
>>> from returns.result import Result
>>> from returns.context import Reader

>>> assert Result.from_value(1) == Result.from_value(1)
>>> Reader.from_value(1) == Reader.from_value(1)
False
```

So, we can use `assert_equal()` or `returns.assert_equal` method from our `pytest` fixture:

```
>>> from returns.result import Success
>>> from returns.context import Reader
>>> from returns.contrib.pytest import ReturnsAsserts

>>> def test_container_equality(returns: ReturnsAsserts):
...     returns.assert_equal(Success(1), Success(1))
...     returns.assert_equal(Reader.from_value(1), Reader.from_value(1))

>>> # We only run these tests manually, because it is a doc example:
>>> returns_fixture = getfixture('returns')
>>> test_container_equality(returns_fixture)
```

`is_error_handled`

The first helper we define is `is_error_handled` function. It tests that containers do handle error track.

```
>>> from returns.result import Failure, Success
>>> from returns.contrib.pytest import ReturnsAsserts

>>> def test_error_handled(returns: ReturnsAsserts):
...     assert not returns.is_error_handled(Failure(1))
```

(continues on next page)

(continued from previous page)

```

...     assert returns.is_error_handled(
...         Failure(1).lash(lambda _: Success('default value')),
...     )

>>> # We only run these tests manually, because it is a doc example:
>>> returns_fixture = getfixture('returns')
>>> test_error_handled(returns_fixture)

```

We recommend to unit test big chunks of code this way. This is helpful for big pipelines where you need at least one error handling at the very end.

This is how it works internally:

- Methods like `fix` and `lash` mark errors inside the container as handled
- Methods like `map` and `alt` just copies the error handling state from the old container to a new one, so there's no need to re-handle the error after these methods
- Methods like `bind` create new containers with unhandled errors

Note: We use monkeypatching of containers inside tests to make this check possible. They are still purely functional inside. It does not affect production code.

assert_trace

Sometimes we have to know if a container is created correctly in a specific point of our flow.

`assert_trace` helps us to check exactly this by identifying when a container is created and looking for the desired function.

```

>>> from returns.result import Result, Success, Failure
>>> from returns.contrib.pytest import ReturnsAsserts

>>> def desired_function(arg: str) -> Result[int, str]:
...     if arg.isnumeric():
...         return Success(int(arg))
...     return Failure("{} is not a number".format(arg))

>>> def test_if_failure_is_created_at_convert_function(
...     returns: ReturnsAsserts,
... ):
...     with returns.assert_trace(Failure, desired_function):
...         Success('not a number').bind(desired_function)

>>> def test_if_success_is_created_at_convert_function(
...     returns: ReturnsAsserts,
... ):
...     with returns.assert_trace(Success, desired_function):
...         Success('42').bind(desired_function)

>>> # We only run these tests manually, because it is a doc example:
>>> returns_fixture = getfixture('returns')

```

(continues on next page)

```
>>> test_if_failure_is_created_at_convert_function(returns_fixture)
>>> test_if_success_is_created_at_convert_function(returns_fixture)
```

markers

We also ship a bunch of pre-defined markers with returns:

- `returns_lawful` is used to mark all tests generated by our *hypothesis plugin*

9.23.3 Further reading

- [pytest docs](#)

9.23.4 API Reference

final class ReturnsAsserts(*errors_handled*)

Bases: object

Class with helpers assertions to check containers.

Parameters

errors_handled (Dict[int, Any]) –

static assert_equal(*first, second, *, deps=None, backend='asyncio'*)

Can compare two containers even with extra calling and awaiting.

Parameters

backend (str) –

Return type

None

is_error_handled(*container*)

Ensures that container has its error handled in the end.

Return type

bool

static assert_trace(*trace_type, function_to_search*)

Ensures that a given function was called during execution.

Use it to determine where the failure happened.

Parameters

- **trace_type** (TypeVar(_ReturnsResultType, bound= Union[ResultLikeN, Callable[... , ResultLikeN]]) –

- **function_to_search** (TypeVar(_FunctionType, bound= Callable)) –

Return type

Iterator[None]

pytest_configure(*config*)

Hook to be executed on import.

We use it define custom markers.

Return type

None

returns()

Returns class with helpers assertions to check containers.

Return type

Iterator[ReturnsAsserts]

assert_equal(*first, second, *, deps=None, backend='asyncio'*)

Custom assert function to compare two any containers.

The important note here is that this `assert` should probably used in tests. Not real application code.

It will call all Reader based containers and `await` all Future based ones.

It also works recursively. For example, `ReaderFutureResult` will be called and then awaited.

You can specify different dependencies to call your containers. And different backends to `await` then using `anyio`.

By the way, `anyio` should be installed separately.

Parameters

backend (str) –

Return type

None

9.24 hypothesis plugin

We provide several extra features for Hypothesis users. And encourage to use it together with `returns`.

9.24.1 Installation

You will need to install `hypothesis` separately. It is not bundled with `returns`.

We also require `anyio` package for this plugin to work with async laws.

9.24.2 hypothesis entrypoint

We support a `hypothesis` entrypoint that is executed on `hypothesis` import.

There we are registering all our containers as strategies. So, you don't have to. Example:

```
from returns.result import Result
from hypothesis import strategies as st

assert st.from_type(Result).example()
```

This is a convenience thing only.

9.24.3 strategy_from_container

We provide a utility function to create hypothesis strategy from any container.

You can use it to easily register your own containers.

```
from hypothesis import strategies as st
from returns.contrib.hypothesis.containers import strategy_from_container

st.register_type_strategy(
    YourContainerClass,
    strategy_from_container(YourContainerClass),
)
```

You can also pass `use_init` keyword argument if you wish to use `__init__` method to instantiate your containers. Turned off by default. Example:

```
st.register_type_strategy(
    YourContainerClass,
    strategy_from_container(YourContainerClass, use_init=True),
)
```

Or you can write your own hypothesis strategy. It is also fine.

9.24.4 check_all_laws

We also provide a very powerful mechanism of checking defined container laws. It works in a combination with “Laws as Values” feature we provide in the core.

```
from returns.contrib.hypothesis.laws import check_all_laws
from your.module import YourCustomContainer

check_all_laws(YourCustomContainer)
```

This one line of code will generate ~100 tests for all defined law in both `YourCustomContainer` and all its super types, including our internal ones.

We also provide a way to configure the checking process with `settings_kwargs`:

```
check_all_laws(YourCustomContainer, settings_kwargs={'max_examples': 500})
```

This will increase the number of generated test to 500. We support all kwargs from `@settings`, see [@settings docs](#).

You can also change how hypothesis creates instances of your container. By default, we use `.from_value`, `.from_optional`, and `.from_failure` if we are able to find them.

But, you can also pass types without these methods, but with `__init__` defined:

```
from typing import Callable, TypeVar, final
from returns.interfaces.mappable import Mappable1
from returns.primitives.container import BaseContainer
from returns.primitives.hkt import SupportsKind1

_ValueType = TypeVar('_ValueType')
_NewValueType = TypeVar('_NewValueType')
```

(continues on next page)

(continued from previous page)

```

@final
class Number(
    BaseContainer,
    SupportsKind1['Number', _ValueType],
    Mappable1[_ValueType],
):
    def __init__(self, inner_value: _ValueType) -> None:
        super().__init__(inner_value)

    def map(
        self,
        function: Callable[[_ValueType], _NewValueType],
    ) -> 'Number[_NewValueType]':
        return Number(function(self._inner_value))

# We want to allow `__init__` method to be used:
check_all_laws(Number, use_init=True)

```

As you see, we don't support any from methods here. But, `__init__` would be used to generate values thanks to `use_init=True`.

By default, we don't allow to use `__init__`, because there are different complex types like `Future`, `ReaderFutureResult`, etc that have complex `__init__` signatures. And we don't want to mess with them.

Warning::

Checking laws is not compatible with `pytest-xdist`, because we use a lot of global mutable state there. Please, use `returns_lawful` marker to exclude them from `pytest-xdist` execution plan.

9.24.5 Further reading

- [Projects Extending hypothesis](#)

9.24.6 API Reference

Types we have already registered for you

Used to register all our types as hypothesis strategies.

See: <https://hypothesis.readthedocs.io/en/latest/strategies.html>

But, beware that we only register concrete types here, interfaces won't be registered!

```

REGISTERED_TYPES: Sequence[Type[Lawful]] = (<class 'returns.result.Result'>, <class
'returns.maybe.Maybe'>, <class 'returns.io.IO'>, <class 'returns.io.IOResult'>, <class
'returns.future.Future'>, <class 'returns.future.FutureResult'>, <class
'returns.context.requires_context.RequiresContext'>, <class
'returns.context.requires_context_result.RequiresContextResult'>, <class
'returns.context.requires_context_ioresult.RequiresContextIOResult'>, <class
'returns.context.requires_context_future_result.RequiresContextFutureResult'>)

```

Our types that we register in hypothesis to be working with `st.from_type`

DSL to register custom containers

strategy_from_container(*container_type*, *, *use_init=False*)

Creates a strategy from a container type.

Basically, containers should not support `__init__` even when they have one. Because, that can be very complex: for example `FutureResult` requires `Awaitable[Result[a, b]]` as an `__init__` value.

But, custom containers pass `use_init` if they are not an instance of `ApplicativeN` and do not have a working `.from_value` method. For example, pure `MappableN` can do that.

We also try to resolve generic arguments. So, `Result[_ValueType, Exception]` will produce any value for success cases and only exceptions for failure cases.

Parameters

- **container_type** (type[*Lawful*]) –
- **use_init** (bool) –

Return type

Callable[[type], SearchStrategy]

DSL to define laws

law_definition

Special alias to define laws as functions even inside a class

class Law(*function*)

Bases: *Immutable*

Base class for all laws. Does not have an attached signature.

Should not be used directly. Use `Law1`, `Law2` or `Law3` instead.

definition: Callable

Function used to define this law.

property name: str

Returns a name of the given law. Basically a name of the function.

final class Law1(*function*)

Bases: *Law*, `Generic[_TypeArgType1, _ReturnType]`

Law definition for functions with a single argument.

Parameters

function (Callable[[TypeVar(_TypeArgType1)], TypeVar(_ReturnType)]) –

final class Law2(*function*)

Bases: *Law*, `Generic[_TypeArgType1, _TypeArgType2, _ReturnType]`

Law definition for functions with two arguments.

Parameters

function (Callable[[TypeVar(_TypeArgType1), TypeVar(_TypeArgType2)], TypeVar(_ReturnType)]) –

final class Law3(*function*)

Bases: *Law*, *Generic*[_TypeArgType1, _TypeArgType2, _TypeArgType3, _ReturnType]

Law definition for functions with three argument.

Parameters

function (Callable[[*TypeVar*(_TypeArgType1), *TypeVar*(_TypeArgType2), *TypeVar*(_TypeArgType3)], *TypeVar*(_ReturnType)]) –

class Lawful

Bases: *Generic*[_Caps]

Base class for all lawful classes.

Allows to smartly collect all defined laws from all parent classes.

final classmethod laws()

Collects all laws from all parent classes.

Algorithm: `:rtype: Dict[Type[Lawful], Sequence[Law]]`

1. First, we collect all unique parents in `__mro__`
2. Then we get the laws definition from each of them
3. Then we structure them in a `type: its_laws` way

class LawSpecDef

Bases: *object*

Base class for all collection of laws aka *LawSpecs*.

Plugin internals

check_all_laws(*container_type*, *, *settings_kwargs*=None, *use_init*=False)

Function to check all defined mathematical laws in a specified container.

Should be used like so:

```
from returns.contrib.hypothesis.laws import check_all_laws
from returns.io import IO

check_all_laws(IO)
```

You can also pass different *hypothesis* settings inside:

```
check_all_laws(IO, {'max_examples': 100})
```

Note: Cannot be used inside doctests because of the magic we use inside.

See also:

- <https://sobolevn.me/2021/02/make-tests-a-part-of-your-app>
- <https://mmhaskell.com/blog/2017/3/13/obey-the-type-laws>

Parameters

- **container_type** (Type[*Lawful*]) –
- **settings_kwargs** (Optional[Dict[str, Any]]) –
- **use_init** (bool) –

Return type

None

container_strategies(*container_type*, *, *settings*)

Registers all types inside a container to resolve to a correct strategy.

For example, let's say we have `Result` type. It is a subtype of `ContainerN`, `MappableN`, `BindableN`, etc. When we check this type, we need `MappableN` to resolve to `Result`.

Can be used independently from other functions.

Parameters

- **container_type** (Type[*Lawful*]) –
- **settings** (*_Settings*) –

Return type

Iterator[None]

maybe_register_container(*container_type*, *, *use_init*)

Temporary registers a container if it is not registered yet.

Parameters

- **container_type** (Type[*Lawful*]) –
- **use_init** (bool) –

Return type

Iterator[None]

pure_functions()

Context manager to resolve all `Callable` as pure functions.

It is not a default in hypothesis.

Return type

Iterator[None]

type_vars()

Our custom `TypeVar` handling.

There are several noticeable differences: `:rtype: Iterator[None]`

1. We add mutable types to the tests: like `list` and `dict`
2. We ensure that values inside strategies are self-equal, for example, `nan` does not work for us

9.25 Version history

We follow Semantic Versions since the 1.0.0 release. Versions before 1.0.0 are `Over`-based: incremental in minor, bugfixes only are patches. See `Over`.

9.25.1 0.22.0

Features

- *Breaking*: Drops python3.7 support
- Now requires `mypy>=1.5`
- Adds trampolines support

9.25.2 0.21.0

Features

- Now requires `mypy>=1.4`
- Adds `[compatible-mypy]` extra for installation

9.25.3 0.20.1

Bugfixes

- Fixed HKT `to_str` example in documentation
- Fixed backward deserialization compatibility for `BaseContainer`

9.25.4 0.20.0

Features

- Now requires `mypy>=1.2`

Bugfixes

- Fixes a problem with `do-notation` and type aliases
- Fixes custom pickle protocol to handle `None` values gracefully
- Removes broken `drylabs.io` link in README
- Revises `pointfree` documentation

9.25.5 0.19.0 aka The Do Notation

Features

- Adds do notation
- Adds attempt decorator

Misc

- Check `__slots__` correctness with `slotscheck``

9.25.6 0.18.0

New Year Release!

Features

- Now requires `typing_extensions>=4.0`
- Now requires `mypy>=0.930`
- Removes plugin for `@safe`, `@maybe`, `@future`, etc. Because we now use `ParamSpec` type to properly type decorators

Bugfixes

- Fixes `__slots__` not being set properly in containers and their base classes
- Fixes patching of containers in pytest plugin not undone after each test

9.25.7 0.17.0

Features

- Enables Pattern Matching support for `Result` containers
- Enables Pattern Matching support for `Maybe` container
- Enables Pattern Matching support for `IOResult` container
- Improves `hypothesis` plugin, now we detect when type cannot be constructed and give a clear error message
- Adds the option to pass what exceptions `@safe` will handle

9.25.8 0.16.0

Features

- Makes `_Nothing` a singleton
- Refactor `flow` function to be faster

Bugfixes

- Fixes that `assert_trace` was not catching containers from `@safe`-wrapped functions

Misc

- Fixes typos in documentation

9.25.9 0.15.0 aka The initial HKT release

Features

- Adds Higher Kinded Types partial support
- **Breaking:** drops `python3.6` support
- **Breaking:** makes our `mypy` plugin not optional, but required!
- **Breaking:** changes all `RequiresContext`-based type arguments order, previously we used to specify `_EnvType` as the first type argument, now it is the last one. This is done to respect new HKT rules
- **Breaking:** renames `.rescue` to `.lash`
- **Breaking:** removes all old interfaces from `primitives/interfaces.py`, use new typeclasses instead
- **Breaking:** `Maybe` is fully reworked to be lawful
- **Breaking:** removes `value_or` pointfree method, because it is impossible to express with HKT
- **Breaking:** removes `.value_or`, `.unwrap`, and `.failure` methods from `FutureResult` and `RequiresContext`-based types, because we do require these methods to raise an exception on failure, but these methods were lazy and did not raise the required exception
- **Breaking:** changes how `is_successful` is typed: now we allow any `Unwrappable` interface instances there, including custom ones
- **Breaking:** changes `UnwrapFailedError` constructor, now it does accept an `Unwrappable` instance instead of a `BaseContainer`
- **Breaking:** removes `.fix` method from all containers, also removes `fix` pointfree function
- **Breaking:** Removes `coalesce` function, because it is impossible to properly type it
- **Breaking:** Removes all `Context*` based types with `.ask()` method, use new `.ask()` methods on the `Reader`-based containers
- **Breaking:** Now `Future` and `FutureResult` can be awaited multiple times
- **Breaking:** Removes `.unify()` method from several containers, use `unify()` pointfree function instead
- **Breaking:** Removes `.from_iterable` method from all containers, instead adds better `iterables` support, we now have `returns.iterables` module with `Fold` helper

- **Breaking:** Renames property `empty` to `no_args` of all `RequiresContext`-based classes
- Adds new public interfaces: see `returns.interfaces`
- Adds `methods` package with several helpful things inside
- Adds `FutureSuccess` and `FutureFailure` unit functions to be similar to `Result` and `IOResult`
- Adds `.swap` method to `Result`, `IOResult`, `FutureResult`, and other result based containers
- Adds `.modify_env` method to all `RequiresContext*` types
- Adds `.rescue` to `Maybe`
- Adds `.equals` methods to types that can be compared directly: `Result`, `Maybe`, `IO`, `IOResult`
- Adds missing `from_requires_context_future_result` to `RequiresContext`
- Adds `.from_optional` and `.bind_optional` to `Maybe` container
- Adds `__slots__` to `UnwrapFailedError` with `halted_container`
- Changes `flatten` to work with `KindN` and any possible container
- Adds a helper to test traces to our `pytest` plugin
- Adds `cond` function to `pointfree` and `methods` packages
- Adds `compose_result` HKT method and `pointfree` function
- Adds `unify` HKT `pointfree` function
- Adds `bimap` `pointfree` function
- Adds `unwrap_or_failure` function to `methods` package
- Adds `collect_trace` helper function for better development experience
- Adds `hypothesis` integration and pre-defined “monad laws as values”
- Adds `assert_equal` method to our `pytest` plugin

Bugfixes

- **Breaking:** fixes serious typing issue and changes how `flow` works
- **Breaking:** fixes serious typing issue and changes how `pipe` works, now it has a hard limit of 20 parameters
- Fixes that `RequiresContextFutureResult` was not supported by `pytest` plugin
- Fixes incorrect `partial` behaviour in an edge case, #618
- Fixes that `.apply` method of `IOResult` was working incorrectly, it was returning `IOFailure(2)` as a result of `IOFailure(1).apply(IOFailure(2))`
- Fixes bug that `safe(tap(...))` was revealing invalid types sometimes

Misc

- Adds a lot of new typetests
- Checks that now all math laws are checked for all types
- Changes docs structure, adds new Interfaces, HKT, and Methods pages
- Changed `__str__` method in `BaseContainer` class to `__repr__` method
- Adds Quickstart guide

9.25.10 0.14.0

Features

- **Breaking:** renames mypy plugin from `decorator_plugin` to `returns_plugin` because of a complete rewrite and lots of new features
- **Breaking:** changes `@safe`, `@impure`, `impure_safe`, `@maybe` semantics: they do not work with `async` functions anymore; now you are forced to use `Future` and its helpers to work with `async` functions
- **Breaking:** renames `Maybe.new` to `Maybe.from_value`. Because all our other containers support this protocol. Only `Maybe` was different, sorry for that!
- **Breaking:** renames `.from_success()` to `.from_value()`, there's no need in two separate methods
- **Breaking:** renames `.from_successful_io()` to `.from_io()`, there's no need in two separate methods
- **Breaking:** renames `.from_successful_context()` to `.from_context()`, there's no need in two separate methods
- **Breaking:** since we now support `.apply()` method, there's no more need in `*_squash` converters, they are removed
- **Breaking:** renamed `Instanceable` to `Applicative`
- **Breaking:** changes `.from_io` and `.from_failed_io` of `IOResult` to return `Any` instead of `NoReturn` unfilled type
- **Breaking:** removes `.lift` and `.lift_*` methods from all containers, use `map_`, `bind_result`, `bind_io`, and other pointfree helpers instead
- **Breaking:** removes `@pipeline` function. It was a mistake: it does not work with mixed container types, it does not type failures properly, it does not work with `IO` and `Future`, it enforces to write imperative code in a functional codebase. Use `flow` instead
- Adds typed `partial` and `curry` mypy plugins!
- Adds typed `flow` plugin, now it can accept any number of arguments, it now also has **excellent** type inference
- Adds typed `pipe` plugin, now it can accept any number of arguments, it now also has good type inference
- Adds managed pipeline function that is useful for working with stateful computations
- Adds typed `map_`, `fix`, and `alt` pointfree functions
- Adds typed `bind_result`, `bind_io`, `bind_ioresult`, `bind_context`, `bind_context_result`, `bind_future`, `bind_async`, and `bind_awaitable` pointfree functions
- Adds typed `bind_async_future` and `bind_async_future_result` pointfree functions
- Adds typed `unify` pointfree function

- Adds typed apply pointfree function
- Adds typed value_or pointfree function
- Adds pytest plugin with the ability to tests error handling
- Adds Future container to easily work with async functions
- Adds FutureResult container to easily work with async function that might fail
- Adds RequiresContextFutureResult container
- Adds ReaderFutureResult alias for RequiresContextFutureResult
- Adds RequiresContextFutureResultE and ReaderFutureResultE aliases
- Adds Future, FutureResult and RequiresContextFutureResult support for all existing pointfree functions
- Adds bind_io method to IOResult
- Adds bind_io method to RequiresContextIOResult
- Adds or_else method to Maybe
- Adds .from_io and .from_failed_io to RequiresContextIOResult
- Syncs naming in from_* methods, now all parameters are named inner_value
- Adds not_ composition helper
- Adds flatten support for Future, FutureResult and RequiresContextFutureResult
- Adds __copy__ and __deepcopy__ magic methods to Immutable class
- Speeds up is_successful function
- Makes all Context context helpers abstract, so you cannot create new instances of this class, also adds __slots__ to these classes
- Improves RequiresContext* types with NoDeps where it is logically true

Bugfixes

- Fixes that @safe decorator was generating incorrect signatures for functions with Any
- Fixes that .rescue() of RequiresContextResult was returning Any
- Fixes that .rescue() of RequiresContextIOResult was returning Any
- Fixes that RequiresContextResult and RequiresContextIOResult were not final
- Fixes that ImmutableStateError was not a subclass of AttributeError
- Fixes that IOResult was not showing str representation of wrapped inner_value

Misc

- Replaces `pytest-asyncio` with `anyio` plugin, now we test compatibility with any IO stack: `asyncio`, `trio`, `curio`
- Updates lots of dependencies
- Adds lots of new tests
- Updates lots of docs
- Removes “IO marker” name from docs in favor for “IO container”, it is not special at all. Why would we call it differently?

9.25.11 0.13.0

Features

- **Breaking:** renames `join` to `flatten`, sorry!
- **Breaking:** renames `box` to `bind` and moves it to `returns.pointfree`
- **Breaking:** removes `Maybe.rescue` and `Maybe.fix` methods
- **Breaking:** renames `io_squash` to `squash_io` and moves it to `returns.converters`
- **Breaking:** moves all interfaces from `returns.primitives.container` to `returns.primitives.interfaces`
- Adds `rescue pointfree` function
- Adds `ResultE` alias for `Result[... , Exception]`
- Adds `RequiresContext` container and `Context` helper class
- Adds `RequiresContext` support for `bind pointfree` function
- Adds `RequiresContext` support for `flatten` function
- Adds `RequiresContextResult` container
- Adds `RequiresContextResultE` alias
- Adds `ReaderResult` and `ReaderResultE` aliases for `RequiresContextResult[... , ... , Exception]`
- Adds `RequiresContextResult` support for `bind` and `rescue`
- Adds `RequiresContextResult` support for `flatten`
- Adds `IOResult` helper to work better with `IO[Result[a, b]]`
- Adds `IOResultE` alias for `IOResult[a, Exception]`
- Adds `IOResult` support for `bind`
- Adds `IOResult` support for `flatten`
- Adds `IOResult` support for `@pipeline`
- Adds `IOResult` support for `coalesce`
- Adds `IOResult` support for `is_successful`
- Adds `RequiresContextIOResult` container
- Adds `RequiresContextIOResultE` alias

- Adds `ReaderIOResult` and `ReaderIOResultE` aliases for `RequiresContextIOResult[... , ..., Exception]`
- Adds `RequiresContextIOResult` support for `bind` and `rescue`
- Adds `RequiresContextIOResult` support for `flatten`
- Adds `Result.lift`, `Maybe.lift`, `RequiresContext.lift`, and `RequiresContextResult.lift` functions in addition to `IO.lift`
- Adds `Immutable` primitive type
- Adds `Unitable` protocol and `.from_success()` and `.from_failure()` methods for all `Result` related classes
- Adds `Instanceable` protocol and `.from_value()` method for `IO` and `RequiresContext`
- Adds `flow` function, which is similar to `pipe`
- Adds swap converter for `Result` and `IOResult`
- Adds `squash_context` function to squash `RequiresContext` similar to `IO`

Bugfixes

- Now `Success` and `Failure` (both `io` and `pure`) return `Any` and not `NoReturn`
- Fixes how `flatten` works, also adds more tests and docs about `Failure` case
- Fixes `Unwrappable` type being parametrized with only one `TypeVar`
- Changes `Success` and `Failure` to return `Any` instead of `NoReturn`

Misc

- Updates poetry version in `travis`
- Improves `pipe` docs with `lambda` and `Generic` problem
- Improves docs in several places
- Now examples in docs tries to be docstests where possible
- Changes how tests are checked with `mypy` in CI

9.25.12 0.12.0

Features

- **Breaking:** now `@pipeline` requires a container type when created: `@pipeline(Result)` or `@pipeline(Maybe)`
- `Maybe` and `Result` now has `success_type` and `failure_type` aliases
- Adds `Result.unify` utility method for better error type composition
- We now support `dry-python/classes` as a first-class citizen
- Adds `io_squash` to squash several `IO` containers into one container with a tuple inside, currently works with 9 containers max at a time
- Adds `untap` function which does convert return type to `None`

Bugfixes

- Fixes that containers were not usable with multiprocessing
- Changes the inheritance order, now `BaseContainer` is the first child
- Fixes that `Nothing` had incorrect docstrings

Misc

- Now generated package is protected
- Updates poetry to 1.0

9.25.13 0.11.0

Features

- **Breaking:** now `pipe()` does not require argument to be the first value, instead it is required to use: `pipe(f1, f2, f3, f4)(value)`
- **Breaking:** dropped everything from `returns/__init__.py`, because we now have quite a lot of stuff
- **Breaking:** dropped support of zero argument functions for `Nothing.fix`
- **Breaking:** dropped support of zero argument functions for `Nothing.rescue`
- Maybe now has `.failure()` to match the same API as `Result`
- Adds `identity` function
- Adds `tap` function
- Now `pipe` allows to pipe 8 steps
- Adds `coalesce_result` and `coalesce_maybe` converters

Bugfixes

- Fixes that code inside `.fix` and `.rescue` of `Maybe` might be called twice

Misc

- Now all methods have doctests
- Updates docs about `Success` and `_Success`, `Failure` and `_Failure`
- Updates docs about `@pipeline`
- Typechecks async functions and decorators inside `typesafety/` tests

9.25.14 0.10.0

Features

- **Breaking:** python>=3.7, <=3.7.2 are not supported anymore, because of a bug inside typing module
- **Breaking:** Now bind does not change the type of an error
- **Breaking:** Now rescue does not change the type of a value
- **Breaking:** Renames map_failure to alt
- Adds box() function with the ability to box function for direct container composition like: a -> Container[b] to Container[a] -> Container[b]
- Adds IO.lift() function to lift a -> a to IO[a] -> IO[a]
- Adds pipe() function to pipeline.py
- Adds __hash__() magic methods to all containers

Bugfixes

- Changes Any to NoReturn in Success and Failure
- Now all type parameters in Result, Maybe, and IO are covariant

Misc

- Massive docs rewrite
- Updates mypy version
- Updates wemake-python-styleguide and introduces nitpick
- Updates pytest-plugin-mypy, all tests now use yaml

9.25.15 0.9.0

Features

- Provides a bunch of primitive interfaces to write your own containers
- Adds .map_failure() method
- Adds flatten() function to join nested containers

Bugfixes

- Fixes type of Maybe.fix and Maybe.rescue to work with both lambda: 1 and lambda _: 1

Misc

- Improves README

9.25.16 0.8.0

Features

- Reintroduces the `Maybe` container, typed!
- Introduces converters from one type to another
- Adds `mypy` plugin to type decorators
- Complete rewrite of `Result` types
- Partial API change, now `Success` and `Failure` are not types, but functions
- New internal types introduced: `FixableContainer` and `ValueUnwrapContainer`

Bugfixes

- Fixes issue when you could return IO container from `Result.bind`
- Fixes `@pipeline` return type

Misc

- Reapplied all types to `.py` files
- Improved docs about IO and `Container` concept
- Adds docs about container composition
- Moves from Alpha to Beta

9.25.17 0.7.0

Features

- Adds IO container
- Adds `unsafe` module with unsafe functions
- Changes how functions are located inside the project

Bugfixes

- Fixes container type in `@pipeline`
- Now `is_successful` is public
- Now `raise_exception` is public

Misc

- Changes how `str()` function works for container types
- Total rename to “container” in the source code

9.25.18 Version 0.6.0

Features

- `safe` and `pipeline` now supports `asyncio`
- `is_successful` now returns `Literal` types if possible

9.25.19 Version 0.5.0

Features

- Adds `compose` helper function
- Adds public API to `import returns`
- Adds `raise_exception` helper function
- Adds full traceback to `.unwrap()`

Misc

- Updates multiple dev-dependencies, including `mypy`
- Now search in the docs is working again
- Relicenses this project to BSD
- Fixes copyright notice in the docs

9.25.20 Version 0.4.0 aka Goodbye, containers!

Features

- Moves all types to `.pyi` files
- Renames all classes according to new naming pattern
- **HUGE** improvement of types
- Renames `fmap` to `map`

- Renames `do_notation` to `pipeline`, moves it to `functions.py`
- Renames `ebind` to `rescue`
- Renames `efmap` to `fix`
- Renames `container` to `Container`
- Removes `Maybe` container, since typing does not have `NonNullable` type

9.25.21 Version 0.3.1

Bugfixes

- Adds `py.typed` file to be PEP561 compatible

9.25.22 Version 0.3.0, Renamed to returns

The project is renamed to `returns` and moved to `dry-python.org`.

Features

- Adds `.pyi` files for all modules, to enable `mypy` support for 3rd party users

9.25.23 Version 0.2.0

Features

- Adds `Maybe` container
- Adds immutability and `__slots__` to all containers
- Adds methods to work with failures
- Adds `safe` decorator to convert exceptions to `Result` container
- Adds `is_successful()` function to detect if your result is a success
- Adds `failure()` method to unwrap values from failed containers

Bugfixes

- Changes the type of `.bind` method for `Success` container
- Changes how equality works, so now `Failure(1) != Success(1)`
- Changes how new instances created on unused methods

Misc

- Improves docs

9.25.24 Version 0.1.1

Bugfixes

- Changes how PyPI renders package's page

Misc

- Improves README with new badges and installation steps

9.25.25 Version 0.1.0

Initial release. Featuring only `Result` and `do_notation`.

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

r

returns.context.requires_context, 164
returns.context.requires_context_future_result, 192
returns.context.requires_context_ioresult, 179
returns.context.requires_context_result, 170
returns.contrib.hypothesis._entrypoint, 293
returns.contrib.hypothesis.containers, 294
returns.contrib.hypothesis.laws, 295
returns.contrib.mypy._consts, 283
returns.contrib.mypy._features.curry, 285
returns.contrib.mypy._features.do_notation, 287
returns.contrib.mypy._features.flow, 285
returns.contrib.mypy._features.kind, 284
returns.contrib.mypy._features.partial, 285
returns.contrib.mypy._features.pipe, 286
returns.contrib.mypy.returns_plugin, 283
returns.contrib.pytest.plugin, 290
returns.converters, 235
returns.curry, 270
returns.functions, 263
returns.future, 131
returns.interfaces.altable, 55
returns.interfaces.applicative, 53
returns.interfaces.bimappable, 56
returns.interfaces.bindable, 52
returns.interfaces.container, 59
returns.interfaces.equable, 50
returns.interfaces.failable, 60
returns.interfaces.lashable, 57
returns.interfaces.mappable, 51
returns.interfaces.specific.future, 70
returns.interfaces.specific.future_result, 73
returns.interfaces.specific.io, 68
returns.interfaces.specific.ioresult, 69
returns.interfaces.specific.maybe, 64
returns.interfaces.specific.reader, 74
returns.interfaces.specific.reader_future_result, 81
returns.interfaces.specific.reader_ioresult, 79
returns.interfaces.specific.reader_result, 78
returns.interfaces.specific.result, 66
returns.interfaces.swappable, 57
returns.interfaces.unwrappable, 58
returns.io, 113
returns.iterables, 275
returns.maybe, 87
returns.pipeline, 233
returns.primitives.asserts, 291
returns.primitives.container, 28
returns.primitives.exceptions, 279
returns.primitives.hkt, 36
returns.primitives.laws, 294
returns.primitives.tracing, 281
returns.primitives.types, 278
returns.result, 98
returns.trampolines, 273
returns.unsafe, 126

Symbols

- `_DiverseFailableLawSpec` (class in `returns.interfaces.failable`), 62
 - `_FailableLawSpec` (class in `returns.interfaces.failable`), 60
 - `_LawSpec` (class in `returns.interfaces.altable`), 55
 - `_LawSpec` (class in `returns.interfaces.applicative`), 53
 - `_LawSpec` (class in `returns.interfaces.container`), 59
 - `_LawSpec` (class in `returns.interfaces.equable`), 50
 - `_LawSpec` (class in `returns.interfaces.mappable`), 51
 - `_LawSpec` (class in `returns.interfaces.specific.maybe`), 64
 - `_LawSpec` (class in `returns.interfaces.specific.reader`), 77
 - `_LawSpec` (class in `returns.interfaces.specific.reader_future_result`), 81
 - `_LawSpec` (class in `returns.interfaces.specific.reader_ioresult`), 80
 - `_LawSpec` (class in `returns.interfaces.specific.reader_result`), 78
 - `_LawSpec` (class in `returns.interfaces.swappable`), 57
 - `_SingleFailableLawSpec` (class in `returns.interfaces.failable`), 61
 - `__abstractmethods__` (BaseContainer attribute), 28
 - `__annotations__` (BaseContainer attribute), 28
 - `__eq__()` (BaseContainer method), 28
 - `__getstate__()` (BaseContainer method), 28
 - `__hash__()` (BaseContainer method), 28
 - `__init__()` (BaseContainer method), 28
 - `__module__` (BaseContainer attribute), 29
 - `__repr__()` (BaseContainer method), 28
 - `__setstate__()` (BaseContainer method), 28
 - `__slots__` (BaseContainer attribute), 28
 - `_laws` (AltableN attribute), 55
 - `_laws` (ApplicativeN attribute), 54
 - `_laws` (ContainerN attribute), 60
 - `_laws` (DiverseFailableN attribute), 63
 - `_laws` (Equable attribute), 50
 - `_laws` (FailableN attribute), 61
 - `_laws` (MappableN attribute), 51
 - `_laws` (MaybeLikeN attribute), 65
 - `_laws` (ReaderBased2 attribute), 77
 - `_laws` (ReaderFutureResultBasedN attribute), 82
 - `_laws` (ReaderIOResultBasedN attribute), 80
 - `_laws` (ReaderResultBasedN attribute), 79
 - `_laws` (SingleFailableN attribute), 62
 - `_laws` (SwappableN attribute), 57
- ## A
- `AbstractFold` (class in `returns.iterables`), 275
 - `alt()` (AltableN method), 56
 - `alt()` (Failure method), 103
 - `alt()` (FutureResult method), 145
 - `alt()` (in module `returns.pointfree`), 254
 - `alt()` (IOResult method), 120
 - `alt()` (RequiresContextFutureResult method), 203
 - `alt()` (RequiresContextIOResult method), 185
 - `alt()` (RequiresContextResult method), 174
 - `alt()` (Result method), 100
 - `alt()` (Success method), 103
 - `alt_short_circuit_law()` (`_DiverseFailableLawSpec` static method), 63
 - `Altable2` (in module `returns.interfaces.altable`), 56
 - `Altable3` (in module `returns.interfaces.altable`), 56
 - `AltableN` (class in `returns.interfaces.altable`), 55
 - `analyze()` (in module `returns.contrib.mypy._features.curry`), 285
 - `analyze()` (in module `returns.contrib.mypy._features.do_notation`), 287
 - `analyze()` (in module `returns.contrib.mypy._features.flow`), 285
 - `analyze()` (in module `returns.contrib.mypy._features.partial`), 285
 - `analyze()` (in module `returns.contrib.mypy._features.pipe`), 286
 - `Applicative1` (in module `returns.interfaces.applicative`), 55
 - `Applicative2` (in module `returns.interfaces.applicative`), 55
 - `Applicative3` (in module `returns.interfaces.applicative`), 55
 - `ApplicativeN` (class in `returns.interfaces.applicative`), 54

- apply() (*ApplicativeN* method), 54
 apply() (*Failure* method), 103
 apply() (*Future* method), 133
 apply() (*FutureResult* method), 140
 apply() (*in module returns.pointfree*), 255
 apply() (*IO* method), 114
 apply() (*IOResult* method), 118
 apply() (*Maybe* method), 88
 apply() (*RequiresContext* method), 165
 apply() (*RequiresContextFutureResult* method), 194
 apply() (*RequiresContextIOResult* method), 181
 apply() (*RequiresContextResult* method), 172
 apply() (*Result* method), 99
 apply() (*Some* method), 92
 apply() (*Success* method), 104
 apply_short_circuit_law() (*_DiverseFail-
ableLawSpec* static method), 63
 apply_short_circuit_law() (*_SingleFail-
ableLawSpec* static method), 61
 args (*Trampoline* attribute), 273
 ask() (*ReaderLike2* class method), 75
 ask() (*ReaderLike3* class method), 76
 ask() (*RequiresContext* class method), 167
 ask() (*RequiresContextFutureResult* class method), 205
 ask() (*RequiresContextIOResult* class method), 187
 ask() (*RequiresContextResult* class method), 176
 asking_law() (*_LawSpec* static method), 77, 79–81
 assert_equal() (*in module returns.primitives.asserts*),
291
 assert_equal() (*ReturnsAsserts* static method), 290
 assert_trace() (*ReturnsAsserts* static method), 290
 associative_law() (*_LawSpec* static method), 51, 55,
59
 async_identity() (*in module returns.future*), 131
 AsyncFuture1 (*in module re-
turns.interfaces.specific.future*), 72
 AsyncFuture2 (*in module re-
turns.interfaces.specific.future*), 72
 AsyncFuture3 (*in module re-
turns.interfaces.specific.future*), 72
 asyncify() (*in module returns.future*), 137
 attempt() (*in module returns.result*), 105
 attribute_access() (*in module re-
turns.contrib.mypy._features.kind*), 284
 awaitable() (*AwaitableFutureN* method), 72
 awaitable() (*Future* method), 132
 awaitable() (*FutureResult* method), 139
 AwaitableFutureN (*class in re-
turns.interfaces.specific.future*), 72
- ## B
- BaseContainer (*class in returns.primitives.container*),
28
 BiMappable2 (*in module returns.interfaces.bimappable*),
56
 BiMappable3 (*in module returns.interfaces.bimappable*),
56
 BiMappableN (*class in returns.interfaces.bimappable*),
56
 bind() (*BindableN* method), 52
 bind() (*Failure* method), 103
 bind() (*Future* method), 133
 bind() (*FutureResult* method), 141
 bind() (*in module returns.pointfree*), 243
 bind() (*IO* method), 114
 bind() (*IOFailure* method), 125
 bind() (*IOResult* method), 119
 bind() (*IOSuccess* method), 125
 bind() (*Maybe* method), 88
 bind() (*RequiresContext* method), 165
 bind() (*RequiresContextFutureResult* method), 195
 bind() (*RequiresContextIOResult* method), 181
 bind() (*RequiresContextResult* method), 172
 bind() (*Result* method), 99
 bind() (*Some* method), 92
 bind() (*Success* method), 103
 bind_async() (*Future* method), 134
 bind_async() (*FutureLikeN* method), 71
 bind_async() (*FutureResult* method), 141
 bind_async() (*in module returns.pointfree*), 251
 bind_async() (*RequiresContextFutureResult* method),
195
 bind_async_context_future_result() (*Reader-
FutureResultLikeN* method), 81
 bind_async_context_future_result() (*Re-
quiresContextFutureResult* method), 196
 bind_async_future() (*Future* method), 134
 bind_async_future() (*FutureLikeN* method), 71
 bind_async_future() (*FutureResult* method), 144
 bind_async_future() (*in module returns.pointfree*),
245
 bind_async_future() (*RequiresContextFutureResult*
method), 202
 bind_async_future_result() (*FutureResult*
method), 142
 bind_async_future_result() (*FutureResultLikeN*
method), 73
 bind_async_future_result() (*in module re-
turns.pointfree*), 247
 bind_async_future_result() (*RequiresContextFu-
tureResult* method), 202
 bind_awaitable() (*Future* method), 134
 bind_awaitable() (*FutureLikeN* method), 71
 bind_awaitable() (*FutureResult* method), 142
 bind_awaitable() (*in module returns.pointfree*), 252
 bind_awaitable() (*RequiresContextFutureResult*
method), 196

- bind_context() (in module returns.pointfree), 249
 bind_context() (ReaderLike2 method), 75
 bind_context() (ReaderLike3 method), 76
 bind_context() (RequiresContext method), 166
 bind_context() (RequiresContextFutureResult method), 198
 bind_context() (RequiresContextIOResult method), 183
 bind_context() (RequiresContextResult method), 174
 bind_context2() (in module returns.pointfree), 248
 bind_context3() (in module returns.pointfree), 248
 bind_context_future_result() (ReaderFutureResultLikeN method), 81
 bind_context_future_result() (RequiresContextFutureResult method), 195
 bind_context_ioresult() (in module returns.pointfree), 250
 bind_context_ioresult() (ReaderIOResultLikeN method), 79
 bind_context_ioresult() (RequiresContextFutureResult method), 199
 bind_context_ioresult() (RequiresContextIOResult method), 182
 bind_context_result() (in module returns.pointfree), 250
 bind_context_result() (ReaderResultLikeN method), 78
 bind_context_result() (RequiresContextFutureResult method), 198
 bind_context_result() (RequiresContextIOResult method), 183
 bind_context_result() (RequiresContextResult method), 173
 bind_future() (Future method), 134
 bind_future() (FutureLikeN method), 70
 bind_future() (FutureResult method), 144
 bind_future() (in module returns.pointfree), 245
 bind_future() (RequiresContextFutureResult method), 201
 bind_future_result() (FutureResult method), 141
 bind_future_result() (FutureResultLikeN method), 73
 bind_future_result() (in module returns.pointfree), 246
 bind_future_result() (RequiresContextFutureResult method), 201
 bind_io() (Future method), 135
 bind_io() (FutureResult method), 143
 bind_io() (in module returns.pointfree), 244
 bind_io() (IO method), 115
 bind_io() (IOFailure method), 125
 bind_io() (IOLikeN method), 68
 bind_io() (IOResult method), 120
 bind_io() (IOSuccess method), 125
 bind_io() (RequiresContextFutureResult method), 200
 bind_io() (RequiresContextIOResult method), 184
 bind_ioresult() (FutureResult method), 143
 bind_ioresult() (in module returns.pointfree), 244
 bind_ioresult() (IOFailure method), 125
 bind_ioresult() (IOResult method), 119
 bind_ioresult() (IOResultLikeN method), 69
 bind_ioresult() (IOSuccess method), 125
 bind_ioresult() (RequiresContextFutureResult method), 200
 bind_ioresult() (RequiresContextIOResult method), 185
 bind_optional() (in module returns.pointfree), 252
 bind_optional() (Maybe method), 89
 bind_optional() (MaybeLikeN method), 65
 bind_optional() (Some method), 92
 bind_optional_short_circuit_law() (_LawSpec static method), 64
 bind_result() (Failure method), 103
 bind_result() (FutureResult method), 142
 bind_result() (in module returns.pointfree), 243
 bind_result() (IOFailure method), 125
 bind_result() (IOResult method), 119
 bind_result() (IOSuccess method), 125
 bind_result() (RequiresContextFutureResult method), 197
 bind_result() (RequiresContextIOResult method), 182
 bind_result() (RequiresContextResult method), 173
 bind_result() (Result method), 100
 bind_result() (ResultLikeN method), 66
 bind_result() (Success method), 104
 bind_short_circuit_law() (_DiverseFailableLawSpec static method), 62
 bind_short_circuit_law() (_LawSpec static method), 64
 bind_short_circuit_law() (_SingleFailableLawSpec static method), 61
 Bindable1 (in module returns.interfaces.bindable), 52
 Bindable2 (in module returns.interfaces.bindable), 53
 Bindable3 (in module returns.interfaces.bindable), 53
 BindableN (class in returns.interfaces.bindable), 52
- ## C
- CallableReader2 (class in returns.interfaces.specific.reader), 75
 CallableReader3 (class in returns.interfaces.specific.reader), 77
 check_all_laws() (in module returns.contrib.hypothesis.laws), 295
 collect() (AbstractFold class method), 276
 collect_all() (AbstractFold class method), 277
 collect_traces() (in module returns.primitives.tracing), 281
 compose() (in module returns.functions), 264

compose_result() (*FutureResult* method), 146
 compose_result() (in module *returns.pointfree*), 253
 compose_result() (*IOResult* method), 122
 compose_result() (*IOResultLikeN* method), 69
 compose_result() (*RequiresContextFutureResult* method), 204
 compose_result() (*RequiresContextIOResult* method), 186
 composition_law() (*_LawSpec* static method), 54
 cond() (in module *returns.methods*), 257
 cond() (in module *returns.pointfree*), 253
 Container1 (in module *returns.interfaces.container*), 60
 Container2 (in module *returns.interfaces.container*), 60
 Container3 (in module *returns.interfaces.container*), 60
 container_equality() (in module *returns.primitives.container*), 29
 container_strategies() (in module *returns.contrib.hypothesis.laws*), 296
 ContainerN (class in *returns.interfaces.container*), 60
 Contextable (class in *returns.interfaces.specific.reader*), 74
 curry() (in module *returns.curry*), 271

D

definition (*Law* attribute), 294
 dekind() (in module *returns.contrib.mypy._features.kind*), 284
 dekind() (in module *returns.primitives.hkt*), 38
 DiverseFailable2 (in module *returns.interfaces.failable*), 64
 DiverseFailable3 (in module *returns.interfaces.failable*), 64
 DiverseFailableN (class in *returns.interfaces.failable*), 63
 do() (*Future* class method), 135
 do() (*FutureResult* class method), 146
 do() (*IO* class method), 115
 do() (*IOResult* class method), 122
 do() (*Maybe* class method), 89
 do() (*Result* class method), 100
 DO_NOTATION_METHODS (in module *returns.contrib.mypy._consts*), 283
 double_swap_law() (*_LawSpec* static method), 57

E

empty (*Maybe* attribute), 87
 empty (*SingleFailableN* property), 62
 Equable (class in *returns.interfaces.equable*), 50
 equals() (*Equable* method), 51
 equals() (*IO* method), 113
 equals() (*IOResult* method), 118
 equals() (*Maybe* method), 88
 equals() (*Result* method), 98

F

Failable2 (in module *returns.interfaces.failable*), 61
 Failable3 (in module *returns.interfaces.failable*), 61
 FailableN (class in *returns.interfaces.failable*), 61
 Failure (class in *returns.result*), 102
 failure() (*Failure* method), 103
 failure() (*IOResult* method), 121
 failure() (*Maybe* method), 91
 failure() (*Result* method), 101
 failure() (*Some* method), 92
 failure() (*Success* method), 104
 failure() (*Unwrappable* method), 58
 failure_type (*IOResult* attribute), 117
 failure_type (*Maybe* attribute), 87
 failure_type (*Result* attribute), 98
 flatten() (in module *returns.converters*), 235
 flow() (in module *returns.pipeline*), 231
 Fold (class in *returns.iterables*), 278
 from_context() (*ReaderLike2* class method), 75
 from_context() (*ReaderLike3* class method), 76
 from_context() (*RequiresContext* class method), 168
 from_context() (*RequiresContextFutureResult* class method), 210
 from_context() (*RequiresContextIOResult* class method), 190
 from_context() (*RequiresContextResult* class method), 177
 from_failed_context() (*ReaderResultLikeN* class method), 78
 from_failed_context() (*RequiresContextFutureResult* class method), 210
 from_failed_context() (*RequiresContextIOResult* class method), 190
 from_failed_context() (*RequiresContextResult* class method), 177
 from_failed_future() (*FutureResult* class method), 148
 from_failed_future() (*FutureResultLikeN* class method), 73
 from_failed_future() (*RequiresContextFutureResult* class method), 208
 from_failed_io() (*FutureResult* class method), 149
 from_failed_io() (*IOResult* class method), 123
 from_failed_io() (*IOResultLikeN* class method), 69
 from_failed_io() (*RequiresContextFutureResult* class method), 207
 from_failed_io() (*RequiresContextIOResult* class method), 188
 from_failure() (*DiverseFailableN* class method), 63
 from_failure() (*FutureResult* class method), 151
 from_failure() (*IOResult* class method), 124
 from_failure() (*RequiresContextFutureResult* class method), 212

- from_failure() (*RequiresContextIOResult class method*), 191
 from_failure() (*RequiresContextResult class method*), 178
 from_failure() (*Result class method*), 102
 from_future() (*Future class method*), 136
 from_future() (*FutureLikeN class method*), 71
 from_future() (*FutureResult class method*), 147
 from_future() (*RequiresContextFutureResult class method*), 208
 from_future_result() (*Future class method*), 137
 from_future_result() (*FutureResult class method*), 148
 from_future_result() (*FutureResultLikeN class method*), 73
 from_future_result() (*RequiresContextFutureResult class method*), 209
 from_future_result_context() (*ReaderFutureResultLikeN class method*), 81
 from_future_result_context() (*RequiresContextFutureResult class method*), 208
 from_io() (*Future class method*), 136
 from_io() (*FutureResult class method*), 149
 from_io() (*IO class method*), 115
 from_io() (*IOLikeN class method*), 68
 from_io() (*IOResult class method*), 123
 from_io() (*RequiresContextFutureResult class method*), 206
 from_io() (*RequiresContextIOResult class method*), 188
 from_ioresult() (*FutureResult class method*), 149
 from_ioresult() (*IO class method*), 116
 from_ioresult() (*IOResult class method*), 124
 from_ioresult() (*IOResultLikeN class method*), 69
 from_ioresult() (*RequiresContextFutureResult class method*), 207
 from_ioresult() (*RequiresContextIOResult class method*), 189
 from_ioresult_context() (*ReaderIOResultLikeN class method*), 80
 from_ioresult_context() (*RequiresContextFutureResult class method*), 211
 from_ioresult_context() (*RequiresContextIOResult class method*), 189
 from_optional() (*Maybe class method*), 92
 from_optional() (*MaybeLikeN class method*), 65
 from_requires_context_future_result() (*RequiresContext class method*), 169
 from_requires_context_ioresult() (*RequiresContext class method*), 169
 from_requires_context_result() (*RequiresContext class method*), 168
 from_result() (*FutureResult class method*), 150
 from_result() (*IOResult class method*), 123
 from_result() (*RequiresContextFutureResult class method*), 206
 from_result() (*RequiresContextIOResult class method*), 188
 from_result() (*RequiresContextResult class method*), 176
 from_result() (*Result class method*), 102
 from_result() (*ResultLikeN class method*), 67
 from_result_context() (*ReaderResultLikeN class method*), 78
 from_result_context() (*RequiresContextFutureResult class method*), 211
 from_result_context() (*RequiresContextIOResult class method*), 191
 from_result_context() (*RequiresContextResult class method*), 177
 from_typecast() (*FutureResult class method*), 147
 from_typecast() (*IOResult class method*), 123
 from_typecast() (*RequiresContextFutureResult class method*), 209
 from_typecast() (*RequiresContextIOResult class method*), 190
 from_typecast() (*RequiresContextResult class method*), 176
 from_value() (*ApplicativeN class method*), 54
 from_value() (*Future class method*), 136
 from_value() (*FutureResult class method*), 150
 from_value() (*IO class method*), 115
 from_value() (*IOResult class method*), 124
 from_value() (*Maybe class method*), 91
 from_value() (*RequiresContext class method*), 168
 from_value() (*RequiresContextFutureResult class method*), 212
 from_value() (*RequiresContextIOResult class method*), 191
 from_value() (*RequiresContextResult class method*), 178
 from_value() (*Result class method*), 102
 func (*Trampoline attribute*), 273
 Future (*class in returns.future*), 132
 future() (*in module returns.future*), 137
 future_safe() (*in module returns.future*), 152
 FutureBased1 (*in module returns.interfaces.specific.future*), 72
 FutureBased2 (*in module returns.interfaces.specific.future*), 72
 FutureBased3 (*in module returns.interfaces.specific.future*), 72
 FutureBasedN (*class in returns.interfaces.specific.future*), 72
 FutureFailure() (*in module returns.future*), 151
 FutureLike1 (*in module returns.interfaces.specific.future*), 71
 FutureLike2 (*in module returns.interfaces.specific.future*), 71

- FutureLike3 (in module *returns.interfaces.specific.future*), 72
- FutureLikeN (class in *returns.interfaces.specific.future*), 70
- FutureResult (class in *returns.future*), 138
- FutureResultBased2 (in module *returns.interfaces.specific.future_result*), 74
- FutureResultBased3 (in module *returns.interfaces.specific.future_result*), 74
- FutureResultBasedN (class in *returns.interfaces.specific.future_result*), 74
- FutureResultE (in module *returns.future*), 152
- FutureResultLike2 (in module *returns.interfaces.specific.future_result*), 74
- FutureResultLike3 (in module *returns.interfaces.specific.future_result*), 74
- FutureResultLikeN (class in *returns.interfaces.specific.future_result*), 73
- FutureSuccess() (in module *returns.future*), 151
- ## H
- halted_container (*UnwrapFailedError* attribute), 279
- homomorphism_law() (*_LawSpec* static method), 53
- ## I
- identity() (in module *returns.functions*), 263
- identity_law() (*_LawSpec* static method), 51, 53, 55
- Immutable (class in *returns.primitives.types*), 278
- ImmutableStateError, 279
- impure() (in module *returns.io*), 116
- impure_safe() (in module *returns.io*), 125
- infer() (in module *returns.contrib.mypy._features.pipe*), 287
- interchange_law() (*_LawSpec* static method), 53
- IO (class in *returns.io*), 113
- IOBased1 (in module *returns.interfaces.specific.io*), 68
- IOBased2 (in module *returns.interfaces.specific.io*), 68
- IOBased3 (in module *returns.interfaces.specific.io*), 69
- IOBasedN (class in *returns.interfaces.specific.io*), 68
- IOFailure (class in *returns.io*), 124
- IOLike1 (in module *returns.interfaces.specific.io*), 68
- IOLike2 (in module *returns.interfaces.specific.io*), 68
- IOLike3 (in module *returns.interfaces.specific.io*), 68
- IOLikeN (class in *returns.interfaces.specific.io*), 68
- IOResult (class in *returns.io*), 116
- IOResultBased2 (in module *returns.interfaces.specific.ioresult*), 70
- IOResultBased3 (in module *returns.interfaces.specific.ioresult*), 70
- IOResultBasedN (class in *returns.interfaces.specific.ioresult*), 70
- IOResultE (in module *returns.io*), 125
- IOResultLike2 (in module *returns.interfaces.specific.ioresult*), 70
- IOResultLike3 (in module *returns.interfaces.specific.ioresult*), 70
- IOResultLikeN (class in *returns.interfaces.specific.ioresult*), 69
- IOSuccess (class in *returns.io*), 125
- is_error_handled() (*ReturnsAsserts* method), 290
- is_successful() (in module *returns.pipeline*), 233
- ## K
- Kind1 (in module *returns.primitives.hkt*), 37
- Kind2 (in module *returns.primitives.hkt*), 37
- Kind3 (in module *returns.primitives.hkt*), 37
- Kinded (class in *returns.primitives.hkt*), 38
- kinded() (in module *returns.primitives.hkt*), 38
- kinded_call() (in module *returns.contrib.mypy._features.kind*), 284
- kinded_get_descriptor() (in module *returns.contrib.mypy._features.kind*), 285
- kinded_signature() (in module *returns.contrib.mypy._features.kind*), 284
- KindN (class in *returns.primitives.hkt*), 36
- kwargs (*Trampoline* attribute), 273
- ## L
- lash() (*Failure* method), 103
- lash() (*FutureResult* method), 145
- lash() (in module *returns.pointfree*), 254
- lash() (*IOFailure* method), 125
- lash() (*IOResult* method), 120
- lash() (*IOSuccess* method), 125
- lash() (*LashableN* method), 57
- lash() (*Maybe* method), 89
- lash() (*RequiresContextFutureResult* method), 204
- lash() (*RequiresContextIOResult* method), 186
- lash() (*RequiresContextResult* method), 175
- lash() (*Result* method), 100
- lash() (*Some* method), 92
- lash() (*Success* method), 104
- lash_short_circuit_law() (*_FailableLawSpec* static method), 60
- lash_short_circuit_law() (*_LawSpec* static method), 65
- Lashable2 (in module *returns.interfaces.lashable*), 58
- Lashable3 (in module *returns.interfaces.lashable*), 58
- LashableN (class in *returns.interfaces.lashable*), 57
- Law (class in *returns.primitives.laws*), 294
- Law1 (class in *returns.primitives.laws*), 294
- Law2 (class in *returns.primitives.laws*), 294
- Law3 (class in *returns.primitives.laws*), 294
- law_definition (in module *returns.primitives.laws*), 294
- Lawful (class in *returns.primitives.laws*), 295
- laws() (*Lawful* class method), 295
- LawSpecDef (class in *returns.primitives.laws*), 295

`left_identity_law()` (*LawSpec* static method), 59
`loop()` (*AbstractFold* class method), 275

M

`managed()` (in module *returns.pipeline*), 232
`map()` (*Failure* method), 103
`map()` (*Future* method), 132
`map()` (*FutureResult* method), 139
`map()` (*IO* method), 114
`map()` (*IOResult* method), 118
`map()` (*MappableN* method), 52
`map()` (*Maybe* method), 88
`map()` (*RequiresContext* method), 165
`map()` (*RequiresContextFutureResult* method), 194
`map()` (*RequiresContextIOResult* method), 180
`map()` (*RequiresContextResult* method), 171
`map()` (*Result* method), 98
`map()` (*Some* method), 92
`map()` (*Success* method), 103
`map_()` (in module *returns.pointfree*), 242
`map_short_circuit_law()` (*DiverseFailableLawSpec* static method), 62
`map_short_circuit_law()` (*LawSpec* static method), 64
`map_short_circuit_law()` (*SingleFailableLawSpec* static method), 61
Mappable1 (in module *returns.interfaces.mappable*), 52
Mappable2 (in module *returns.interfaces.mappable*), 52
Mappable3 (in module *returns.interfaces.mappable*), 52
MappableN (class in *returns.interfaces.mappable*), 51
Maybe (class in *returns.maybe*), 87
`maybe()` (in module *returns.maybe*), 92
`maybe_register_container()` (in module *returns.contrib.hypothesis.laws*), 296
`maybe_to_result()` (in module *returns.converters*), 236
MaybeBased2 (in module *returns.interfaces.specific.maybe*), 66
MaybeBased3 (in module *returns.interfaces.specific.maybe*), 66
MaybeBasedN (class in *returns.interfaces.specific.maybe*), 66
MaybeLike2 (in module *returns.interfaces.specific.maybe*), 66
MaybeLike3 (in module *returns.interfaces.specific.maybe*), 66
MaybeLikeN (class in *returns.interfaces.specific.maybe*), 65
`modify_env()` (in module *returns.pointfree*), 250
`modify_env()` (*ReaderLike2* method), 75
`modify_env()` (*ReaderLike3* method), 76
`modify_env()` (*RequiresContext* method), 166
`modify_env()` (*RequiresContextFutureResult* method), 205

`modify_env()` (*RequiresContextIOResult* method), 187
`modify_env()` (*RequiresContextResult* method), 175
`modify_env2()` (in module *returns.pointfree*), 249
`modify_env3()` (in module *returns.pointfree*), 249
module
 returns.context.requires_context, 164
 returns.context.requires_context_future_result, 192
 returns.context.requires_context_ioresult, 179
 returns.context.requires_context_result, 170
 returns.contrib.hypothesis._entrypoint, 293
 returns.contrib.hypothesis.containers, 294
 returns.contrib.hypothesis.laws, 295
 returns.contrib.mypy._consts, 283
 returns.contrib.mypy._features.curry, 285
 returns.contrib.mypy._features.do_notation, 287
 returns.contrib.mypy._features.flow, 285
 returns.contrib.mypy._features.kind, 284
 returns.contrib.mypy._features.partial, 285
 returns.contrib.mypy._features.pipe, 286
 returns.contrib.mypy.returns_plugin, 283
 returns.contrib.pytest.plugin, 290
 returns.converters, 235
 returns.curry, 270
 returns.functions, 263
 returns.future, 131
 returns.interfaces.altable, 55
 returns.interfaces.applicative, 53
 returns.interfaces.bimappable, 56
 returns.interfaces.bindable, 52
 returns.interfaces.container, 59
 returns.interfaces.equable, 50
 returns.interfaces.failable, 60
 returns.interfaces.lashable, 57
 returns.interfaces.mappable, 51
 returns.interfaces.specific.future, 70
 returns.interfaces.specific.future_result, 73
 returns.interfaces.specific.io, 68
 returns.interfaces.specific.ioresult, 69
 returns.interfaces.specific.maybe, 64
 returns.interfaces.specific.reader, 74
 returns.interfaces.specific.reader_future_result, 81
 returns.interfaces.specific.reader_ioresult, 79
 returns.interfaces.specific.reader_result, 78

returns.interfaces.specific.result, 66
 returns.interfaces.swappable, 57
 returns.interfaces.unwrappable, 58
 returns.io, 113
 returns.iterables, 275
 returns.maybe, 87
 returns.pipeline, 233
 returns.primitives.asserts, 291
 returns.primitives.container, 28
 returns.primitives.exceptions, 279
 returns.primitives.hkt, 36
 returns.primitives.laws, 294
 returns.primitives.tracing, 281
 returns.primitives.types, 278
 returns.result, 98
 returns.trampolines, 273
 returns.unsafe, 126

N

name (*Law property*), 294
 no_args (*ReaderLike2 property*), 75
 no_args (*ReaderLike3 property*), 76
 no_args (*RequiresContext attribute*), 165
 no_args (*RequiresContextFutureResult attribute*), 193
 no_args (*RequiresContextIOResult attribute*), 180
 no_args (*RequiresContextResult attribute*), 171
 NoDeps (*in module returns.context.requires_context*), 164
 not_() (*in module returns.functions*), 265
 Nothing (*in module returns.maybe*), 92

O

or_else_call() (*Maybe method*), 90
 or_else_call() (*MaybeBasedN method*), 66
 or_else_call() (*Some method*), 92

P

partial() (*in module returns.curry*), 270
 pipe() (*in module returns.pipeline*), 232
 plugin() (*in module returns.contrib.mypy.returns_plugin*), 283
 pure_functions() (*in module returns.contrib.hypothesis.laws*), 296
 purity_law() (*_LawSpec static method*), 77, 78
 pytest_configure() (*in module returns.contrib.pytest.plugin*), 290

R

raise_exception() (*in module returns.functions*), 265
 Reader (*in module returns.context.requires_context*), 170
 ReaderBased2 (*class in returns.interfaces.specific.reader*), 77
 ReaderFutureResult (*in module returns.context.requires_context_future_result*), 213

ReaderFutureResultBased3 (*in module returns.interfaces.specific.reader_future_result*), 82
 ReaderFutureResultBasedN (*class in returns.interfaces.specific.reader_future_result*), 82
 ReaderFutureResultE (*in module returns.context.requires_context_future_result*), 213
 ReaderFutureResultLike3 (*in module returns.interfaces.specific.reader_future_result*), 81
 ReaderFutureResultLikeN (*class in returns.interfaces.specific.reader_future_result*), 81
 ReaderIOResult (*in module returns.context.requires_context_ioresult*), 192
 ReaderIOResultBased3 (*in module returns.interfaces.specific.reader_ioresult*), 80
 ReaderIOResultBasedN (*class in returns.interfaces.specific.reader_ioresult*), 80
 ReaderIOResultE (*in module returns.context.requires_context_ioresult*), 192
 ReaderIOResultLike3 (*in module returns.interfaces.specific.reader_ioresult*), 80
 ReaderIOResultLikeN (*class in returns.interfaces.specific.reader_ioresult*), 79
 ReaderLike2 (*class in returns.interfaces.specific.reader*), 75
 ReaderLike3 (*class in returns.interfaces.specific.reader*), 76
 ReaderResult (*in module returns.context.requires_context_result*), 178
 ReaderResultBased3 (*in module returns.interfaces.specific.reader_result*), 79
 ReaderResultBasedN (*class in returns.interfaces.specific.reader_result*), 79
 ReaderResultE (*in module returns.context.requires_context_result*), 179
 ReaderResultLike3 (*in module returns.interfaces.specific.reader_result*), 78
 ReaderResultLikeN (*class in returns.interfaces.specific.reader_result*), 78
 reflexive_law() (*_LawSpec static method*), 50
 REGISTERED_TYPES (*in module returns.contrib.hypothesis._entrypoint*), 293
 RequiresContext (*class in returns.context.requires_context*), 164

RequiresContextFutureResult (class in *returns.context.requires_context_future_result*), 192
 RequiresContextFutureResultE (in module *returns.context.requires_context_future_result*), 213
 RequiresContextIOResult (class in *returns.context.requires_context_ioresult*), 179
 RequiresContextIOResultE (in module *returns.context.requires_context_ioresult*), 192
 RequiresContextResult (class in *returns.context.requires_context_result*), 170
 RequiresContextResultE (in module *returns.context.requires_context_result*), 178
 Result (class in *returns.result*), 98
 result_to_maybe() (in module *returns.converters*), 235
 ResultBased2 (in module *returns.interfaces.specific.result*), 67
 ResultBased3 (in module *returns.interfaces.specific.result*), 67
 ResultBasedN (class in *returns.interfaces.specific.result*), 67
 ResultE (in module *returns.result*), 104
 ResultLike2 (in module *returns.interfaces.specific.result*), 67
 ResultLike3 (in module *returns.interfaces.specific.result*), 67
 ResultLikeN (class in *returns.interfaces.specific.result*), 66
 returns() (in module *returns.contrib.pytest.plugin*), 291
 returns.context.requires_context module, 164
 returns.context.requires_context_future_result module, 192
 returns.context.requires_context_ioresult module, 179
 returns.context.requires_context_result module, 170
 returns.contrib.hypothesis._entrypoint module, 293
 returns.contrib.hypothesis.containers module, 294
 returns.contrib.hypothesis.laws module, 295
 returns.contrib.mypy._consts module, 283
 returns.contrib.mypy._features.curry module, 285
 returns.contrib.mypy._features.do_notation module, 287
 returns.contrib.mypy._features.flow module, 285
 returns.contrib.mypy._features.kind module, 284
 returns.contrib.mypy._features.partial module, 285
 returns.contrib.mypy._features.pipe module, 286
 returns.contrib.mypy.returns_plugin module, 283
 returns.contrib.pytest.plugin module, 290
 returns.converters module, 235
 returns.curry module, 270
 returns.functions module, 263
 returns.future module, 131
 returns.interfaces.alttable module, 55
 returns.interfaces.applicative module, 53
 returns.interfaces.bimappable module, 56
 returns.interfaces.bindable module, 52
 returns.interfaces.container module, 59
 returns.interfaces.equable module, 50
 returns.interfaces.failable module, 60
 returns.interfaces.lashable module, 57
 returns.interfaces.mappable module, 51
 returns.interfaces.specific.future module, 70
 returns.interfaces.specific.future_result module, 73
 returns.interfaces.specific.io module, 68
 returns.interfaces.specific.ioresult module, 69
 returns.interfaces.specific.maybe module, 64
 returns.interfaces.specific.reader module, 74
 returns.interfaces.specific.reader_future_result module, 81
 returns.interfaces.specific.reader_ioresult module, 79
 returns.interfaces.specific.reader_result

- module, 78
 - returns.interfaces.specific.result
 - module, 66
 - returns.interfaces.swappable
 - module, 57
 - returns.interfaces.unwrappable
 - module, 58
 - returns.io
 - module, 113
 - returns.iterables
 - module, 275
 - returns.maybe
 - module, 87
 - returns.pipeline
 - module, 233
 - returns.primitives.asserts
 - module, 291
 - returns.primitives.container
 - module, 28
 - returns.primitives.exceptions
 - module, 279
 - returns.primitives.hkt
 - module, 36
 - returns.primitives.laws
 - module, 294
 - returns.primitives.tracing
 - module, 281
 - returns.primitives.types
 - module, 278
 - returns.result
 - module, 98
 - returns.trampolines
 - module, 273
 - returns.unsafe
 - module, 126
 - ReturnsAsserts (class in returns.contrib.pytest.plugin), 290
 - right_identity_law() (*_LawSpec* static method), 59
- S**
- safe() (in module returns.result), 104
 - signature() (in module returns.contrib.mypy._features.pipe), 287
 - SingleFailable2 (in module returns.interfaces.failable), 62
 - SingleFailable3 (in module returns.interfaces.failable), 62
 - SingleFailableN (class in returns.interfaces.failable), 62
 - Some (class in returns.maybe), 92
 - strategy_from_container() (in module returns.contrib.hypothesis.containers), 294
 - Success (class in returns.result), 103
 - success_type (*IOResult* attribute), 117
 - success_type (*Maybe* attribute), 87
 - success_type (*Result* attribute), 98
 - SupportsKind1 (in module returns.primitives.hkt), 37
 - SupportsKind2 (in module returns.primitives.hkt), 37
 - SupportsKind3 (in module returns.primitives.hkt), 38
 - SupportsKindN (class in returns.primitives.hkt), 37
 - swap() (*Failure* method), 103
 - swap() (*FutureResult* method), 139
 - swap() (*IOResult* method), 118
 - swap() (*RequiresContextFutureResult* method), 193
 - swap() (*RequiresContextIOResult* method), 180
 - swap() (*RequiresContextResult* method), 171
 - swap() (*Result* method), 98
 - swap() (*Success* method), 104
 - swap() (*SwappableN* method), 57
 - Swappable2 (in module returns.interfaces.swappable), 57
 - Swappable3 (in module returns.interfaces.swappable), 57
 - SwappableN (class in returns.interfaces.swappable), 57
 - symmetry_law() (*_LawSpec* static method), 50
- T**
- tap() (in module returns.functions), 264
 - trace (*IOResult* property), 118
 - trace (*Result* property), 98
 - Trampoline (class in returns.trampolines), 273
 - trampoline() (in module returns.trampolines), 273
 - transitivity_law() (*_LawSpec* static method), 50
 - type_vars() (in module returns.contrib.hypothesis.laws), 296
 - TYPED_CURRY_FUNCTION (in module returns.contrib.mypy._consts), 283
 - TYPED_FLOW_FUNCTION (in module returns.contrib.mypy._consts), 283
 - TYPED_KINDN (in module returns.contrib.mypy._consts), 283
 - TYPED_PARTIAL_FUNCTION (in module returns.contrib.mypy._consts), 283
 - TYPED_PIPE_FUNCTION (in module returns.contrib.mypy._consts), 283
- U**
- unify() (in module returns.pointfree), 255
 - unit_structure_law() (*_LawSpec* static method), 65
 - unsafe_perform_io() (in module returns.unsafe), 126
 - untap() (in module returns.functions), 264
 - unwrap() (*Failure* method), 103
 - unwrap() (*IOResult* method), 121
 - unwrap() (*Maybe* method), 91
 - unwrap() (*Result* method), 101
 - unwrap() (*Some* method), 92
 - unwrap() (*Success* method), 104
 - unwrap() (*Unwrappable* method), 58

`unwrap_or_failure()` (*in module returns.methods*),
257

`UnwrapFailedError`, 279

`Unwrappable` (*class in returns.interfaces.unwrappable*),
58

`UnwrappableResult` (*class in re-*
turns.interfaces.specific.result), 67

V

`value_or()` (*Failure method*), 103

`value_or()` (*IOResult method*), 121

`value_or()` (*Maybe method*), 90

`value_or()` (*Result method*), 101

`value_or()` (*Some method*), 92

`value_or()` (*Success method*), 104