Monad Documentation

Release 0.1

Philip Xu

July 20, 2014

Contents

Contents:

CHAPTER 1

monad - a functional python package

1.1 Introduction

1.1.1 What?

Monads in python, with some helpful functions.

1.1.2 How?

```
>>> from monad.decorators import maybe
>>> parse_int = maybe(int)
>>> parse_int(42)
Just(42)
>>> parse_int('42')
Just(42)
>>> parse_int('42.2')
Nothing

>>> parse_float = maybe(float)
>>> parse_float('42.2')
Just(42.2)

>>> from monad.actions import tryout
>>> parse_number = tryout(parse_int, parse_float)
>>> tokens = [2, '0', '4', 'eight', '10.0']
>>> [parse_number(token) for token in tokens]
[Just(2), Just(0), Just(4), Nothing, Just(10.0)]

>>> @maybe
... def reciprocal(n):
...     return 1. / n
>>> reciprocal(2)
Just(0.5)
>>> reciprocal(0)
Nothing

>>> process = parse_number >> reciprocal
>>> process('4')
Just(0.25)
>>> process('0')
Nothing
```

```
>>> [process(token) for token in tokens]
[Just(0.5), Nothing, Just(0.25), Nothing, Just(0.1)]
>>> [parse_number(token) >> reciprocal for token in tokens]
[Just(0.5), Nothing, Just(0.25), Nothing, Just(0.1)]
>>> [parse_number(token) >> reciprocal >> reciprocal for token in tokens]
[Just(2.0), Nothing, Just(4.0), Nothing, Just(10.0)]
```

1.1.3 Why?

Why not.

1.2 Requirements

- CPython >= 2.7

1.3 Installation

Install from PyPI:

```
pip install monad
```

Install from source, download source package, decompress, then `cd` into source directory, run:

```
make install
```

1.4 License

BSD New, see LICENSE for details.

1.5 Links

Documentation: http://monad.readthedocs.org/

Issue Tracker: https://bitbucket.org/pyx/monad/issues/

Source Package @ PyPI: https://pypi.python.org/pypi/monad/

Mercurial Repository @ bitbucket: https://bitbucket.org/pyx/monad/

Git Repository @ Github: https://github.com/pyx/monad/

CHAPTER 2

API Reference

monad - a functional library

2.1 Actions

monad.actions - useful monadic actions.

`monad.actions.`**`either`**(*left_handler*, *right_handler=identity*)

Case analysis for `Either`.

Returns a function that when called with a value of type `Either`, applies either `left_handler` or `right_handler` to that value depending on the type of it. If an incompatible value is passed, a `TypeError` will be raised.

```
>>> def log(v):
...     print('Got Left({})'.format(v))
>>> logger = either(left_handler=log)
>>> logger(Left(1))
Got Left(1)
>>> logger(Right(1))
1
>>> def inc(v):
...     return v + 1
>>> act = either(log, inc)
>>> [act(v) for v in (Left(0), Right(1), Left(2), Right(3))]
Got Left(0)
Got Left(2)
[None, 2, None, 4]
```

`monad.actions.`**`first`**(*sequence*, *default=Nothing*, *predicate=None*)

Iterate over a sequence, return the first `Just`.

If `predicate` is provided, `first` returns the first item that satisfy the `predicate`, the item will be wrapped in a `Just` if it is not already, so that the return value of this function will be an instance of `Maybe` in all circumstances. Returns `default` if no satisfied value in the sequence, `default` defaults to `Nothing`.

```
>>> from monad.types import Just, Nothing
>>> first([Nothing, Nothing, Just(42), Nothing])
Just(42)
>>> first([Just(42), Just(43)])
Just(42)
>>> first([Nothing, Nothing, Nothing])
Nothing
```

```
>>> first([])
Nothing
>>> first([Nothing, Nothing], default=Just(2))
Just(2)
>>> first([False, 0, True], predicate=bool)
Just(True)
>>> first([False, 0, Just(1)], predicate=bool)
Just(1)
>>> first([False, 0, ''], predicate=bool)
Nothing
>>> first(range(100), predicate=lambda x: x > 40 and x % 2 == 0)
Just(42)
>>> first(range(100), predicate=lambda x: x > 100)
Nothing
```

This is basically a customized version of `msum` for `Maybe`, a separate function like this is needed because there is no way to write a generic `msum` in python that cab be evaluated in a non-strict way. The obvious `reduce(operator.add, sequence)`, albeit beautiful, is strict, unless we build up the sequence with generator expressions in-place.

Maybe (pun intended!) implemented as `MonadOr` instead of `MonadPlus` might be more semantically correct in this case.

`monad.actions.`**`tryout`**(**functions*)

Combine functions into one.

Returns a monadic function that when called, will try out functions in `functions` one by one in order, testing the result, stop and return with the first value that is true or the last result.

```
>>> zero = lambda n: 'zero' if n == 0 else False
>>> odd = lambda n: 'odd' if n % 2 else False
>>> even = lambda n: 'even' if n % 2 == 0 else False
>>> test = tryout(zero, odd, even)
>>> test(0)
'zero'
>>> test(1)
'odd'
>>> test(2)
'even'
```

2.2 Decorators

monad.decorators - helpful decorators.

`monad.decorators.`**`failsafe`**(*callable_object=None*, *predicate=None*, *left_on_value=Null*, *left_on_exception=<type 'exceptions.Exception'>*)

Transform a callable into a function returns an `Either`.

```
>>> parse_int = failsafe(int)
>>> parse_int(42)
Right(42)
>>> parse_int(42.0)
Right(42)
>>> parse_int('42')
Right(42)
>>> parse_int('invalid')
Left(ValueError(...))
```

```
>>> parse_pos = failsafe(int, predicate=lambda i: i > 0)
>>> parse_pos('42')
Right(42)
>>> parse_pos('-42')
Left(-42)

>>> parse_nonzero = failsafe(int, left_on_value=0)
>>> parse_nonzero('42')
Right(42)
>>> parse_nonzero('0')
Left(0)

>>> @failsafe(left_on_exception=ZeroDivisionError)
... def safe_div(a, b):
...     return a / b
>>> safe_div(42.0, 2)
Right(21.0)
>>> safe_div(42, 0)
Left(ZeroDivisionError(...))
```

When invoked, this new function returns the return value of decorated function, wrapped in an `Either` monad.

`predicate` should be a false value, or be set to a callable. The default is `None`.

`left_on_value` can be set to any object supporting comparison against return value of the original function. The default is `Null`, which means no checking on the return value.

`left_on_exception` should be a false value, or a type of exception, or a tuple of exceptions. The default is `Exception`, which will suppress most exceptions and return `Left(exception)` instead.

The returned monad will be `Left` if

- `predicate` is set, and `predicate(result_from_decorated_function)` returns true value (not necessarily equal to `True`)
- `left_on_value` is set and the result from decorated function matches it, testing with ==
- `left_on_exception` is set and a compatible exception has been caught, the exception will be suppressed in this case, and the value of exception will be wrapped in a `Left`
- exception `ExtractError` has been caught, this could be the case, for example, trying to extract value from `Nothing`
- any combination of the above

Otherwise, the result will be wrapped in a `Right`.

`monad.decorators.`**`function`**(*callable_object*)

Decorator that wraps a callabe into `Function`.

```
>>> to_int = function(int)
>>> to_int('42')
42
>>> @function
... def puts(msg, times=1):
...     while times > 0:
...         print(msg)
...         times -= 1
>>> puts('Hello, world', 2)
Hello, world
Hello, world
```

monad.decorators.**maybe**(*callable_object=None*, *predicate=None*, *nothing_on_value=Null*, *nothing_on_exception=<type 'exceptions.Exception'>*)

Transform a callable into a function returns a `Maybe`.

```
>>> parse_int = maybe(int)
>>> parse_int(42)
Just(42)
>>> parse_int(42.0)
Just(42)
>>> parse_int('42')
Just(42)
>>> parse_int('invalid')
Nothing

>>> parse_pos = maybe(int, predicate=lambda i: i > 0)
>>> parse_pos('42')
Just(42)
>>> parse_pos('-42')
Nothing

>>> parse_nonzero = maybe(int, nothing_on_value=0)
>>> parse_nonzero('42')
Just(42)
>>> parse_nonzero('0')
Nothing

>>> @maybe(nothing_on_exception=ZeroDivisionError)
... def safe_div(a, b):
...     return a / b
>>> safe_div(42.0, 2)
Just(21.0)
>>> safe_div(42, 0)
Nothing
```

When invoked, this new function returns the return value of decorated function, wrapped in a `Maybe` monad.

`predicate` should be a false value, or be set to a callable. The default is `None`.

`nothing_on_value` can be set to any object supporting comparison against return value of the original function. The default is `Null`, which means no checking on the return value.

`nothing_on_exception` can be a false value, a type of exception, or a tuple of exceptions. The default is `Exception`, which will suppress most exceptions and return `Nothing` instead.

The returned monad will be `Nothing` if

- `predicate` is set, and `predicate(result_from_decorated_function)` returns true value (not necessarily equal to `True`)
- `nothing_on_value` is set and the result from decorated function matches it, testing with ==
- `nothing_on_exception` is set and a compatible exception has been caught, the exception will be suppressed in this case
- exception `ExtractError` has been caught, when trying to extract value from `Nothing`
- any combination of the above

Otherwise, the result will be wrapped in a `Just`.

monad.decorators.**monadic**(*callable_object*)

Decorator that wraps a callabe into `Monadic`.

monad.decorators.**producer**(*function_or_generator=None*, *empty_on_exception=None*)

Transform a callable into a producer that when called, returns `List`.

```
>>> @producer
... def double(a):
...     yield a
...     yield a
>>> List(42) >> double
List(42, 42)

>>> @producer
... def times(a):
...     for b in List(1, 2, 3):
...         yield '{}x{}={}'.format(a, b, a * b)
>>> List(1, 2) >> times
List('1x1=1', '1x2=2', '1x3=3', '2x1=2', '2x2=4', '2x3=6')
```

`function_or_generator` can be a function that returns an iterable, or a generator.

`empty_on_exception` can be a false value, a type of exception, or a tuple of exceptions. The default is `None`, which will not suppress all exceptions except `ExtractError`, in which case, an empty `List` will be returned.

2.3 Exceptions

monad.exceptions - custom exceptions.

exception monad.exceptions.**ExtractError**(*monad*)

Bases: exceptions.Exception

Raised when failed to extract value from monad.

2.4 Common Mixin Classes

monad.mixins - implements common mixin classes.

class monad.mixins.**ContextManager**

Bases: object

Mixin class that support `with` statement for monad.

class monad.mixins.**Ord**

Bases: object

Mixin class that implements rich comparison ordering methods.

2.5 Types

2.5.1 The Null Object

monad.types.null - The Null type.

monad.types.null.**Null** = **Null**

The Null object.

2.5.2 Lazy Sequence

monad.types.lazysequence - a sequence type with lazy evaluation.

class monad.types.lazysequence.**LazySequence**(*iterable*)
Bases: _abcoll.Sequence

Sequence with lazy evaluation.

```
>>> from itertools import count
>>> seq = LazySequence(count())
>>> seq[1]
1
>>> list(seq[3:5])
[3, 4]
>>> list(seq[:20:2])
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

strict
Proxy to self that forces evaluation when accessed.

2.5.3 Functor

monad.types.functor - The Functor Class.

class monad.types.functor.**Functor**(*value*)
Bases: object

The Functor Class.

Defines function fmap, and should satisfy these laws:

```
fmap id  ==  id
fmap (f . g)  ==  fmap f . fmap g
```

fmap(*function*)
The fmap operation.

2.5.4 Applicative Functor

monad.types.applicative - The Applicative Functor Class.

class monad.types.applicative.**Applicative**(*value*)
Bases: monad.types.functor.Functor

The Applicative Functor Class.

Defines the following functions:

- unit which act as constructor, it's called pure in some context.

unit = NotImplemented
The unit.

Maps a value to a value in this type. Also called pure or return depends on context.

2.5.5 Function

monad.types.function - The Function Wrapper.

class monad.types.function.**Function**(*callable_object*)

Bases: object

The Function Wrapper.

Support function composition via * operator.

```
>>> add_1 = Function(lambda n: n + 1)
>>> inc = add_1 * int
>>> inc('42')
43
```

Support function piping via | operator.

```
>>> inc2 = int | add_1 | add_1 | str
>>> inc2('42')
'44'
```

2.5.6 Monadic Function

monad.types.monadic - The Monadic Fuction Wrapper.

class monad.types.monadic.**Monadic**(*callable_object*)

Bases: monad.types.function.Function

The Monadic Function Wrapper.

Implements Kleisli composition operators >> and <<. It is equivalent to (>=>) and (<=<) in haskell.

2.5.7 Monad

monad.types.monad - The Monad Class.

class monad.types.monad.**Monad**(*value*)

Bases: monad.types.applicative.Applicative

The Monad Class.

Implements bind operator >> and inverted bind operator << as syntactic sugar. It is equivalent to (>>=) and (=<<) in haskell, not to be confused with (>>) and (<<) in haskell.

As python treats assignments as statements, there is no way we can overload >>= as a chainable bind, be it directly overloaded through __irshift__, or derived by python itself through __rshift__.

The default implementations of bind, fmap and join are mutual recursive, subclasses should at least either overload bind, or fmap and join, or all of them for better performance.

bind(*function*)

The bind operation.

function is a function that maps from the underlying value to a monadic type, something like signature f :: a -> M a in haskell's term.

The default implementation defines bind in terms of fmap and join.

fmap (*function*)
The fmap operation.

The default implementation defines `fmap` in terms of `bind` and `unit`.

join()
The join operation.

The default implementation defines `join` in terms of `bind` and `identity` function.

unit
The `unit` of monad.

alias of `Monad`

class `monad.types.monad.`**Unit**
Bases: `object`

Descriptor that always return the owner monad, used for `unit`.

2.5.8 Monad Plus

monad.types.monadplus - The MonadPlus Class.

class `monad.types.monadplus.`**MonadPlus**(*value*)
Bases: `monad.types.monad.Monad`

The MonadPlus Class.

Monads that also support choice and failure.

plus(*monad*)
The Associative operation.

zero = NotImplemented
The identity of `plus`.

This property should be a singleton, the following must be `True`:

```
MP.zero is MP.zero
```

It should satisfy the following law, left zero (notice the bind operator is haskell's >>= here):

```
zero >>= f = zero
```

2.5.9 The Identity Monad

monad.types.identity - The Identity Monad.

class `monad.types.identity.`**Identity**(*value*)
Bases: `monad.types.monad.Monad`, `monad.mixins.ContextManager`, `monad.mixins.Ord`

The Identity Monad.

```
>>> Identity(42)
Identity(42)
>>> Identity([1, 2, 3])
Identity([1, 2, 3])
```

Comparison with ==, as long as what's wrapped inside are comparable.

```
>>> Identity(42) == Identity(42)
True
>>> Identity(42) == Identity(24)
False
```

2.5.10 The Maybe Monad

monad.types.maybe - The Maybe Monad.

monad.types.maybe.**Just**
alias of Maybe

class monad.types.maybe.**Maybe**(*value*)
Bases: monad.types.monadplus.MonadPlus, monad.mixins.ContextManager, monad.mixins.Ord

The Maybe Monad.

Representing values/computations that may fail.

```
>>> Just(42)
Just(42)
>>> Just([1, 2, 3])
Just([1, 2, 3])
>>> Just(Nothing)
Just(Nothing)
>>> Just(Just(2))
Just(Just(2))
>>> isinstance(Just(1), Maybe)
True
>>> isinstance(Nothing, Maybe)
True
>>> saving = 100
>>> spend = lambda cost: Nothing if cost > saving else Just(saving - cost)
>>> spend(90)
Just(10)
>>> spend(120)
Nothing
>>> safe_div = lambda a, b: Nothing if b == 0 else Just(a / b)
>>> safe_div(12.0, 6)
Just(2.0)
>>> safe_div(12.0, 0)
Nothing
```

Bind operation with >>

```
>>> inc = lambda n: Just(n + 1) if isinstance(n, int) else Nothing
>>> Just(0)
Just(0)
>>> Just(0) >> inc
Just(1)
>>> Just(0) >> inc >> inc
Just(2)
>>> Just('zero') >> inc
Nothing
```

Comparison with ==, as long as what's wrapped inside are comparable.

```
>>> Just(42) == Just(42)
True
>>> Just(42) == Nothing
False
>>> Nothing == Nothing
True
```

bind (*function*)
The bind operation of `Maybe`.

Applies function to the value if and only if this is a `Just`.

classmethod from_value (*value*)
Wraps `value` in a `Maybe` monad.

Returns a `Just` if the value is evaluated as true. `Nothing` otherwise.

`monad.types.maybe.`**Nothing = Nothing**
The `Maybe` that represents nothing, a singleton, like `None`.

2.5.11 The Either Monad

monad.types.either - The Either Monad.

class `monad.types.either.`**Either** (*value*)
Bases: `monad.types.monad.Monad`, `monad.mixins.ContextManager`, `monad.mixins.Ord`

The Either Monad.

Represents values/computations with two possibilities.

```
>>> Right(42)
Right(42)
>>> Right([1, 2, 3])
Right([1, 2, 3])
>>> Left('Error')
Left('Error')
>>> Right(Left('Error'))
Right(Left('Error'))
>>> isinstance(Right(1), Either)
True
>>> isinstance(Left(None), Either)
True
>>> saving = 100
>>> broke = Left('I am broke')
>>> spend = lambda cost: broke if cost > saving else Right(saving - cost)
>>> spend(90)
Right(10)
>>> spend(120)
Left('I am broke')
>>> safe_div = lambda a, b: Left(str(a) + '/0') if b == 0 else Right(a / b)
>>> safe_div(12.0, 6)
Right(2.0)
>>> safe_div(12.0, 0)
Left('12.0/0')
```

Bind operation with >>

```
>>> inc = lambda n: Right(n + 1) if type(n) is int else Left('Type error')
>>> Right(0)
Right(0)
>>> Right(0) >> inc
Right(1)
>>> Right(0) >> inc >> inc
Right(2)
>>> Right('zero') >> inc
Left('Type error')
```

Comparison with ==, as long as they are the same type and what's wrapped inside are comparable.

```
>>> Left(42) == Left(42)
True
>>> Right(42) == Right(42)
True
>>> Left(42) == Right(42)
False
```

A `Left` is less than a `Right`, or compare the two by the values inside if thay are of the same type.

```
>>> Left(42) < Right(42)
True
>>> Right(0) > Left(100)
True
>>> Left('Error message') > Right(42)
False
>>> Left(100) > Left(42)
True
>>> Right(-2) < Right(-1)
True
```

bind (*function*)
The bind operation of `Either`.

Applies function to the value if and only if this is a `Right`.

unit = <monad.types.monadic.Monadic object at 0x7f6b2aa4c350>

class monad.types.either.**Left** (*value*)
Bases: `monad.types.either.Either`

Left of `Either`.

class monad.types.either.**Right** (*value*)
Bases: `monad.types.either.Either`

Right of `Either`.

2.5.12 The List Monad

monad.types.list - The List Monad.

class monad.types.list.**List** (**items*)
Bases: `monad.types.monadplus.MonadPlus`, `monad.mixins.Ord`, `_abcoll.Sequence`

The List Monad.

Representing nondeterministic computation.

```
>>> List(42)
List(42)
>>> List(1, 2, 3)
List(1, 2, 3)
>>> List([])
List([])
>>> List.from_iterable(range(3))
List(0, 1, 2)
>>> List.from_iterable(n for n in (1, 2, 3) if n % 2 == 0)
List(2)
>>> List(List(2))
List(List(2))
```

Lists are lazy

```
>>> from itertools import count
>>> m = List.from_iterable(count())
>>> m[:5]
List(0, 1, 2, 3, 4)
>>> m[520:524]
List(520, 521, 522, 523)
>>> list(m[1000:1002])
[1000, 1001]
```

Bind operation with >>

```
>>> spawn = lambda cell: List(cell, cell)
>>> spawn('c')
List('c', 'c')
>>> spawn('c') >> spawn
List('c', 'c', 'c', 'c')
>>> grow = lambda cell: List(cell + '~')
>>> grow('o')
List('o~')
>>> grow('o') >> grow >> grow >> grow
List('o~~~~')
>>> generation = lambda cell: grow(cell) + spawn(cell)
>>> first = List('o')
>>> first
List('o')
>>> first >> generation
List('o~', 'o', 'o')
>>> first >> generation >> generation
List('o~~', 'o~', 'o~', 'o~', 'o', 'o', 'o~', 'o', 'o')
```

fmap (*function*)
fmap of List Monad.

classmethod from_iterable (*iterator*)
Creates `List` from iterable.

join()
join of List Monad.

plus (*monad*)
plus operation, concatenates two `List`.

2.6 Utility Functions

monad.utils - utility functions and values.

class monad.utils.**SuppressContextManager**(**exceptions*)
Bases: object

Context manager class that suppress specified exceptions.

monad.utils.**compose**(*f*, *g*)
Function composition.

```
compose(f, g) -> f .  g
```

```
>>> add_2 = lambda a: a + 2
>>> mul_5 = lambda a: a * 5
>>> mul_5_add_2 = compose(add_2, mul_5)
>>> mul_5_add_2(1)
7
>>> add_2_mul_5 = compose(mul_5, add_2)
>>> add_2_mul_5(1)
15
```

monad.utils.**identity**(*a*)
Identity function.

monad.utils.**ignore_exception_set**(**exceptions*)
Helper function for suppress.

monad.utils.**suppress**(**exceptions*)
Context manager that suppress specified exceptions.

```
>>> with suppress(ZeroDivisionError):
...     42 / 0
```

CHAPTER 3

Changelog

- 0.1

  First public release.

CHAPTER 4

Indices and tables

- *genindex*
- *modindex*
- *search*

m