Functors, Applicatives, And Monads In Pictures

Edit New page

Jump to bottom

mustafardestroyer edited this page on Dec 2, 2019 · 27 revisions

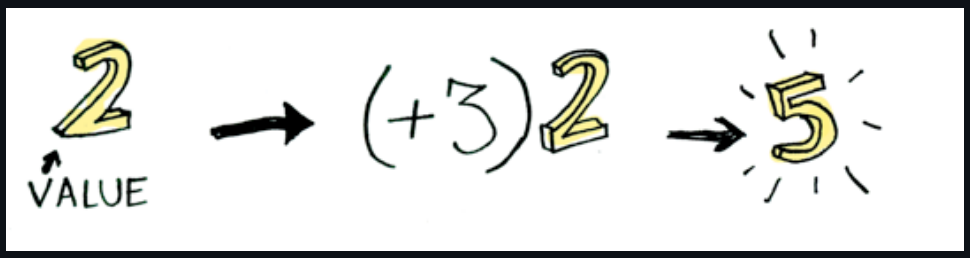
> This is a translation of Functors, Applicatives, And Monads In Pictures from Haskell into Python. Hopefully this should make the article much easier to understand for people who don't know Haskell. All code samples uses the Python OSlash library (Python 3 only).

How to make the examples work with Python:

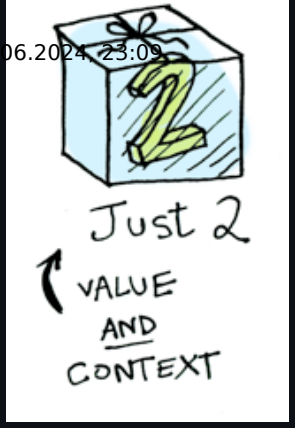
```
$ pip3 install oslash
python3
>>> from oslash import *
```

Here's a simple value:

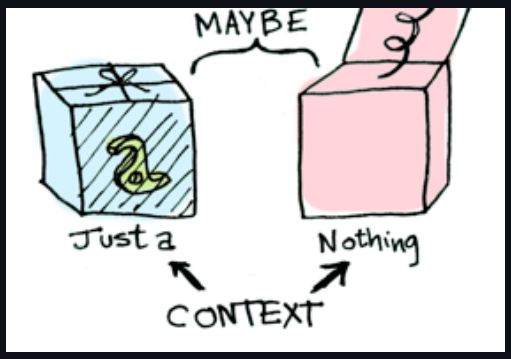
And we know how to apply a function to this value:



Simple enough. Lets extend this by saying that any value can be in a context. For now you can think of a context as a box that you can put a value in:



Now when you apply a function to this value, you'll get different results **depending on the context**. This is the idea that Functors, Applicatives, Monads, Arrows etc are all based on. The Maybe data type defines two related contexts:

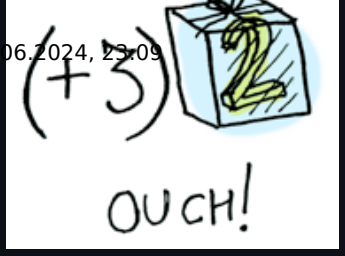



```
class Maybe(Monad, Monoid, Applicative, Functor, metaclass=ABCMeta):
    """The Maybe type encapsulates an optional value. A value of type Maybe a
    either contains a value of (represented as Just a), or it is empty
    (represented as Nothing). Using Maybe is a good way to deal with errors or
    exceptional cases without resorting to drastic measures such as error.
    """
    ...

class Just(Maybe):
    """Represents a value of type Maybe that contains a value (represented as
    Just a).
    """
    ...

class Nothing(Maybe):
    """Represents an empty Maybe that holds nothing (in which case it has the
    value of Nothing).
    """
    ...
```

In a second we'll see how function application is different when something is a Just a versus a Nothing. First let's talk about Functors!

Functors

When a value is wrapped in a context, you can't apply a normal function to it:



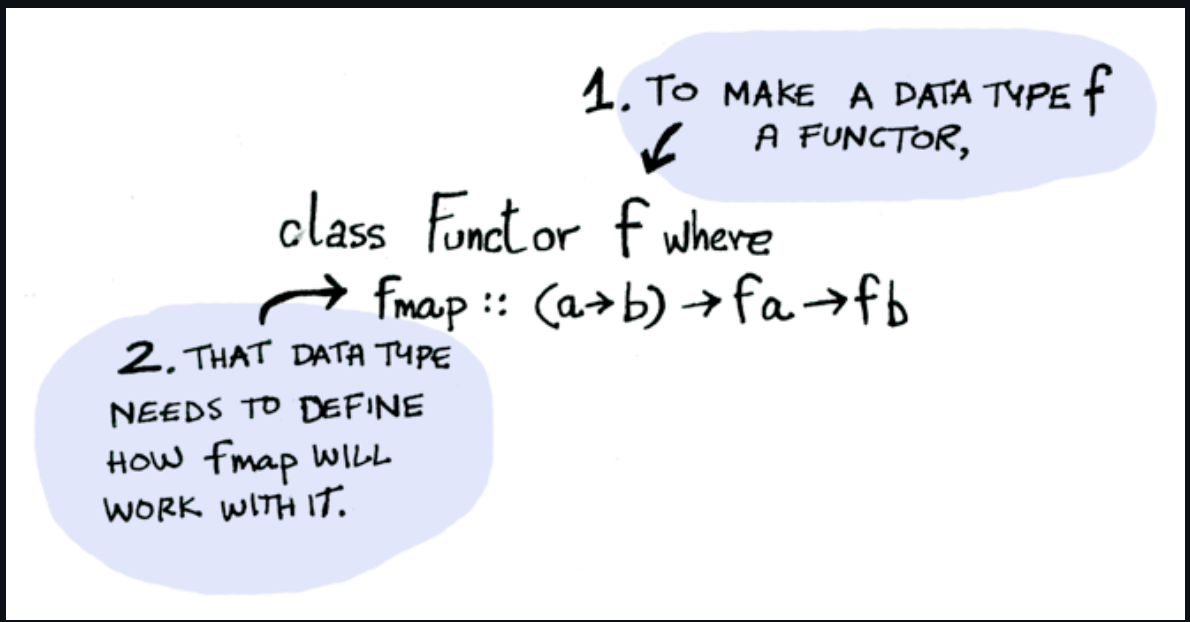
This is where `map` comes in (`fmap` in Haskell). `map` is from the street, `map` is hip to contexts. `map` knows how to apply functions to values that are wrapped in a context. For example, suppose you want to apply `(+3)` to `Just 2`. Use `map`:

```
>>> Just(2).map(lambda x: x+3)
Just 5
```

Bam! `map` shows us how it's done! But how does `map` know how to apply the function?

Just what is a Functor, really?

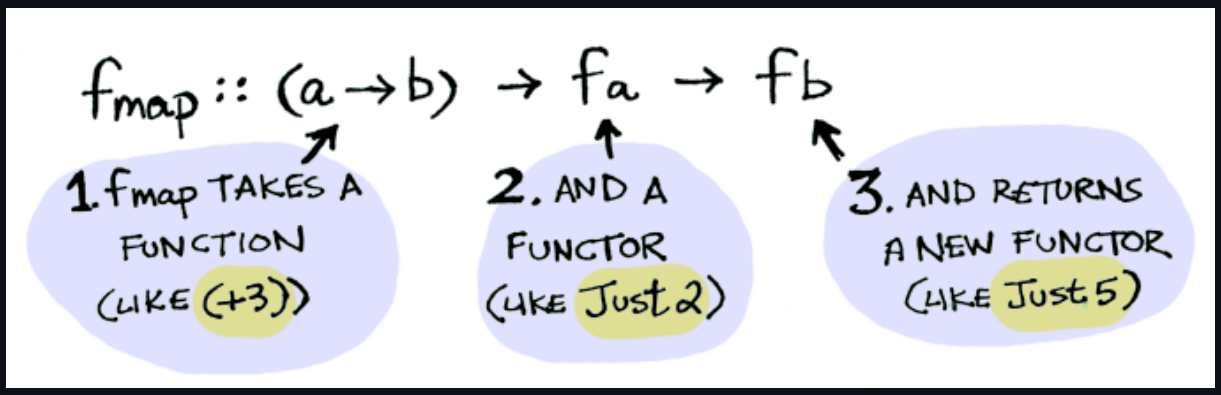
Functor is an Abstract Base Class (typeclass in Haskell). Here's the definition:



In Python:

```
class Functor(metaclass=ABCMeta):
    @abstractmethod
    def map(self, func) -> "Functor":
        return NotImplemented
```

A Functor is any data type that defines how `map` applies to it. Here's how `map` works:



So we can do this:

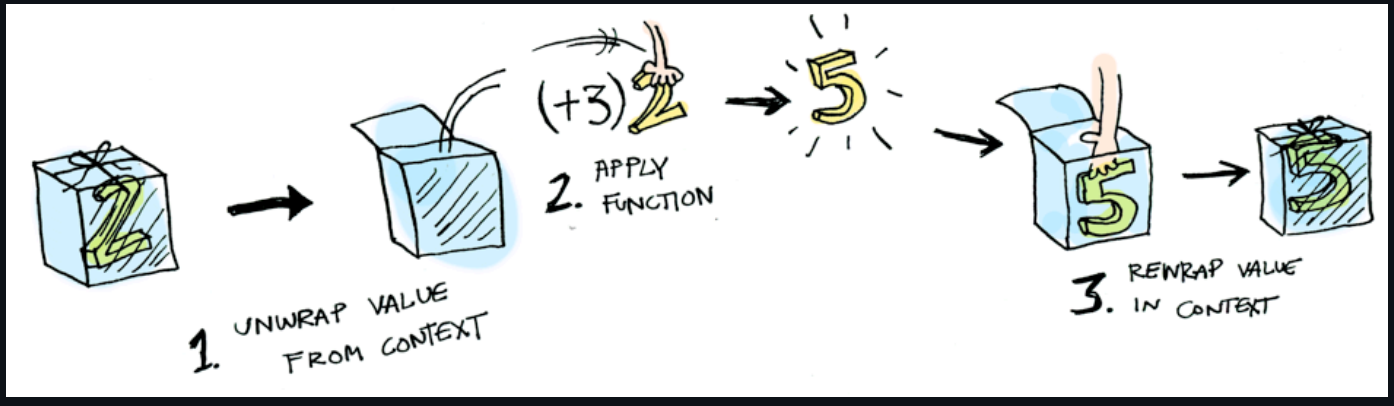
```
>>> Just(2).map(lambda x: x+3)
Just 5
```

And `map` magically applies this function, because `Maybe` is a `Functor`. It specifies how `fmap` applies to `Just`s and `Nothing`s:

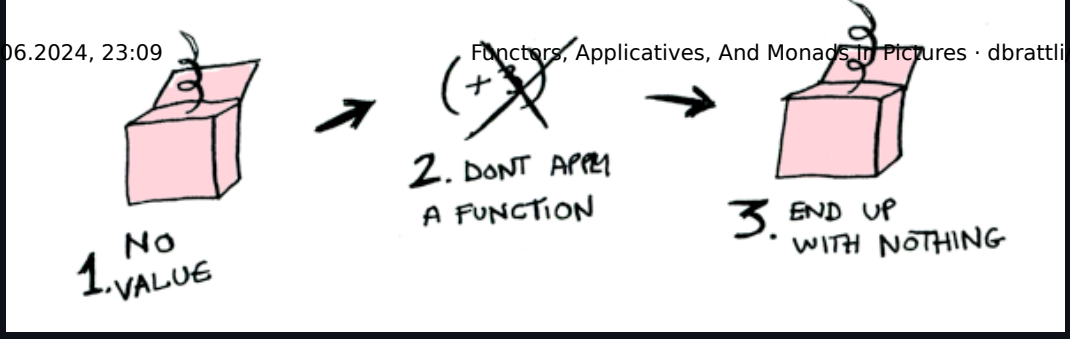
```
class Just(Maybe):
    def map(self, mapper) -> Maybe:
        result = ...
        return Just(result)
```

```
class Nothing(Maybe):
    def map(self, _) -> Maybe:
        return Nothing()
```

Here's what is happening behind the scenes when we write `Just(2).map(lambda x: x+3)`:



So then you're like, alright `map`, please apply `lambda x: x+3` to a Nothing?




```
>>> Nothing().map(lambda x: x+3)
Nothing
```

Bill O'Reilly being totally ignorant about the Maybe functor

Like Morpheus in the Matrix, `map` knows just what to do; you start with `Nothing`, and you end up with `Nothing`! `map` is zen. Now it makes sense why the `Maybe` data type exists. For example, here's how you work with a database record in a language without `Maybe`:

```
post = Post.find_by_id(1)
if post
  return post.title
else
  return nil
end
```

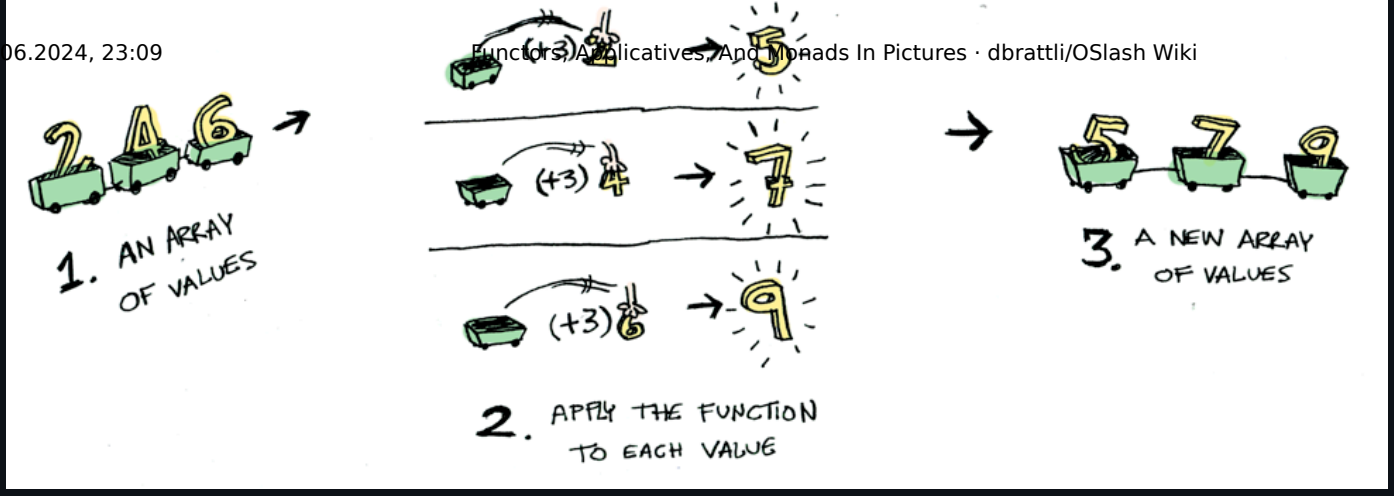
But in Python:

```
find_post(1).map(get_post_title)
```

If `find_post()` returns a post, we will get the title with `get_post_title`. If it returns `Nothing`, we will return `Nothing`! Pretty neat, huh? `%` (`<$>` in Haskell) is the infix version of `map`, so you will often see this instead:

```
get_post_title % find_post(1)
```

Here's another example: what happens when you apply a function to a list?

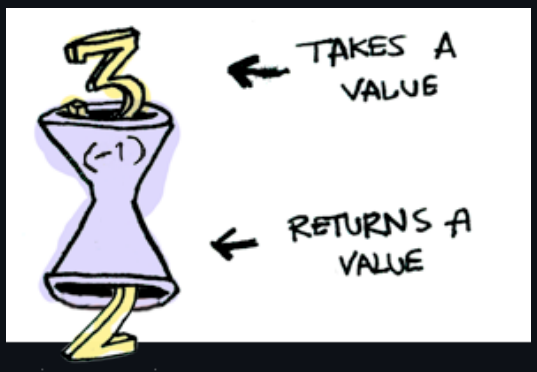


Lists are functors too! Here's the definition:

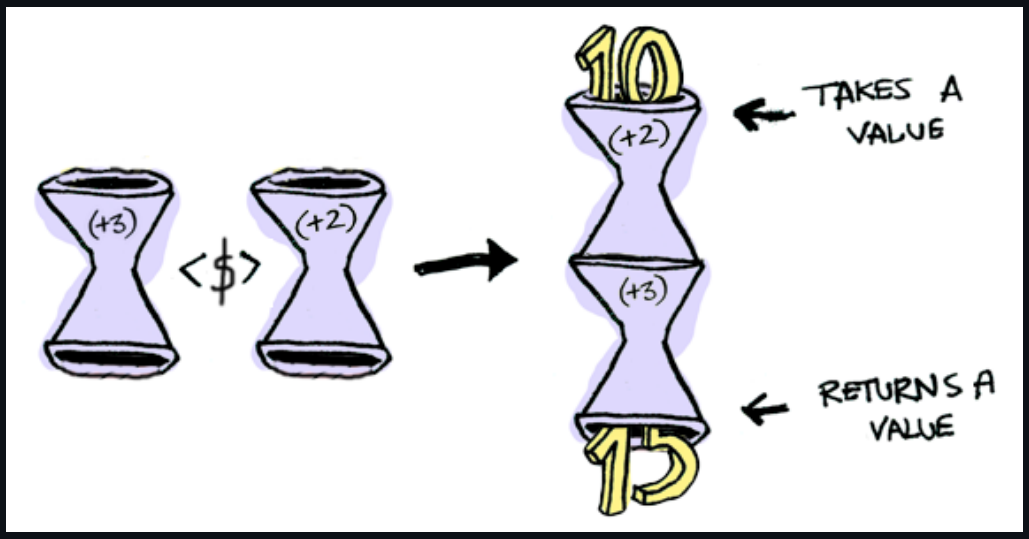
```
class List(Monad, Monoid, Applicative, Functor, list)
```

Okay, okay, one last example: what happens when you apply a function to another function?

```
map(lambda x: x+2, lambda y: y+3)
```



Here's a function applied to another function:



The result is just another function!

```
>>> def fmap(f, g):
...     return lambda x: g(f(x))
...
>>> foo = fmap(lambda x: x+3, lambda y: y+2)
>>> foo(10)
15
```

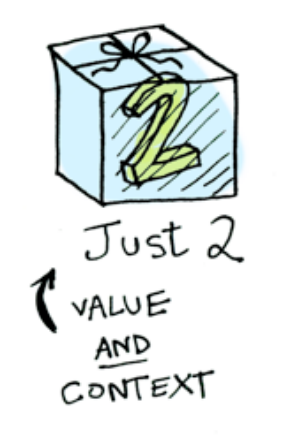
So functions are Functors too!

> It was actually quite easy to define an `fmap` function in Python that made it possible for us to compose functions.

When you use fmap on a function, you're just doing function composition!

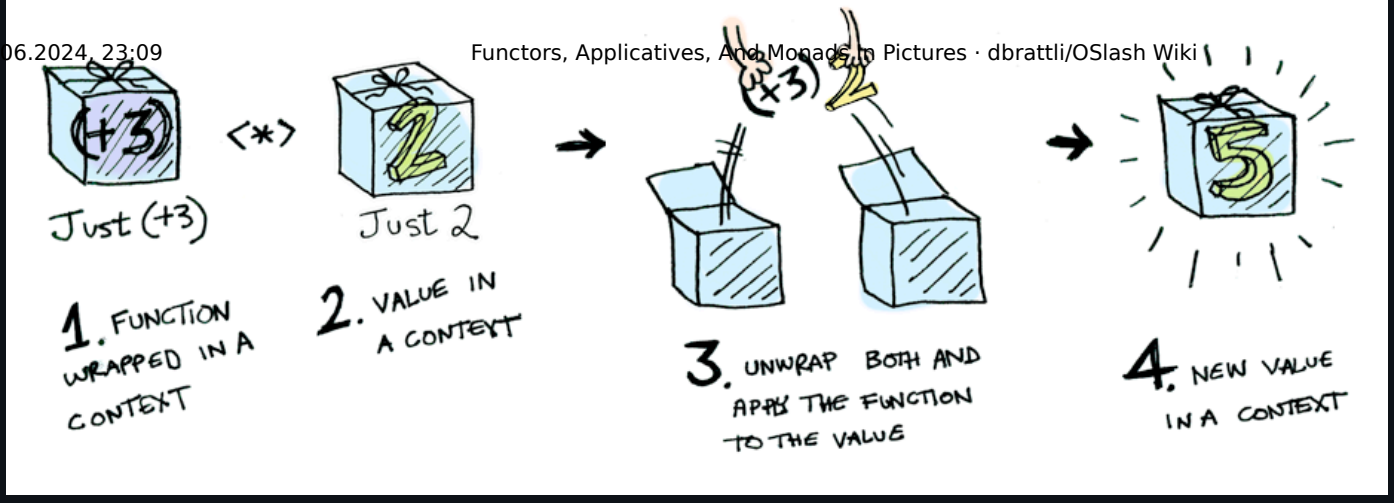
Applicatives

Applicatives take it to the next level. With an applicative, our values are wrapped in a context, just like Functors:



But our functions are wrapped in a context too!

Yeah. Let that sink in. Applicatives don't kid around. Applicative defines `*` (`<*>` in Haskell), which knows how to apply a function wrapped in a context to a value wrapped in a context:

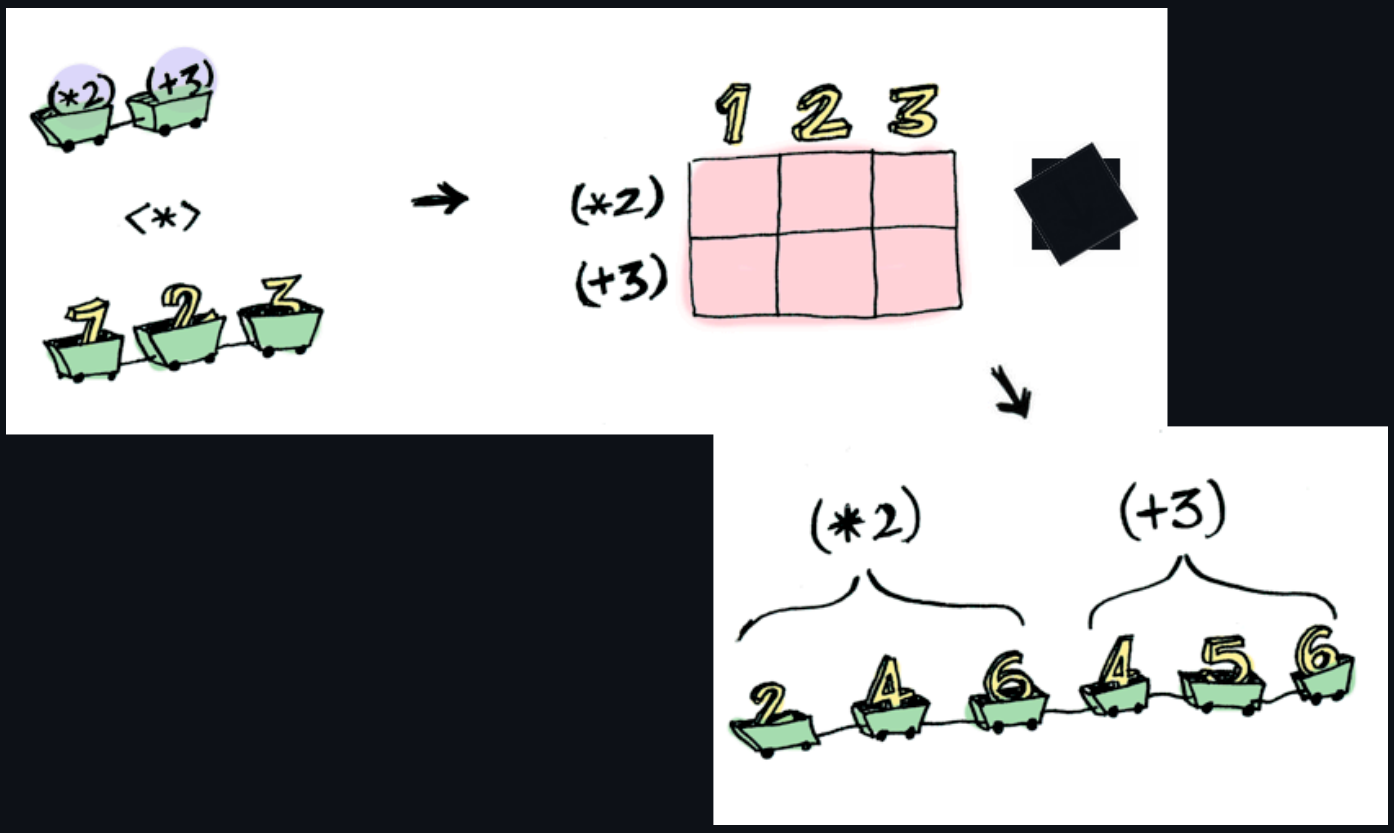


i.e:

```
>>> Just(lambda x: x+3) * Just(2) == Just(5)
True
```

Using `*` can lead to some interesting situations. For example:

```
>> List([lambda x: x*2, lambda y: y+3]) * List([1, 2, 3])
[2, 4, 6, 4, 5, 6]
```



Here's something you can do with Applicatives that you can't do with Functors. How do you apply a function that takes two arguments to two wrapped values?

```
>>> (lambda x,y: x+y) % Just(5)
Just functools.partial(<function <lambda> at 0x1003c1bf8>, 5)
```

```
>>> Just(lambda x: x+5) % Just(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for <<: 'Just' and 'Just'
```

Applicatives:

```
>>> (lambda x,y: x+y) % Just(5)
Just functools.partial(<function <lambda> at 0x1003c1bf8>, 5)

>>> Just(lambda x: x+5) * Just(5)
Just 10
```

`Applicative` pushes `Functor` aside. “Big boys can use functions with any number of arguments,” it says. “Armed with `%` and `*`, I can take any function that expects any number of unwrapped values. Then I pass it all wrapped values, and I get a wrapped value out! AHAHAHAHAH!”

```
>>> (lambda x,y: x*y) % Just(5) * Just(3)
Just 15
```

And hey! There’s a method called `lift_a2` that does the same thing:

```
>>> Just(5).lift_a2(lambda x,y: x*y, Just(3))
Just 15
```

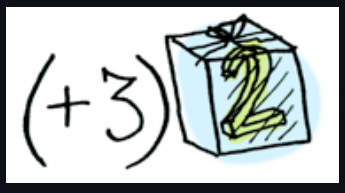
Monads

How to learn about Monads:

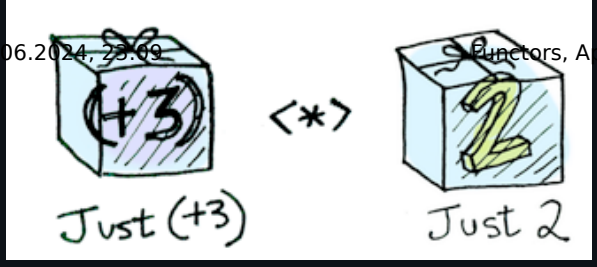
1. Get a PhD in computer science.
2. Throw it away because you don’t need it for this section!

Monads add a new twist.

Functors apply a function to a wrapped value:

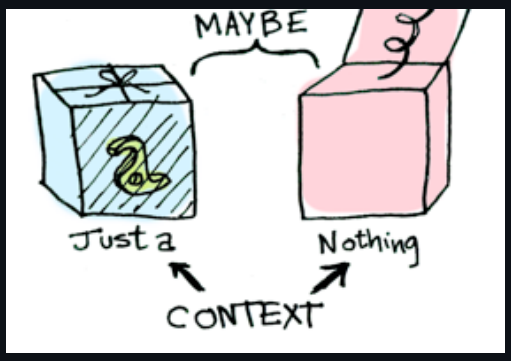


Applicatives apply a wrapped function to a wrapped value:



Monads apply a function that returns a wrapped value to a wrapped value. Monads have a function `|` (>>= in Haskell) (pronounced “bind”) to do this.

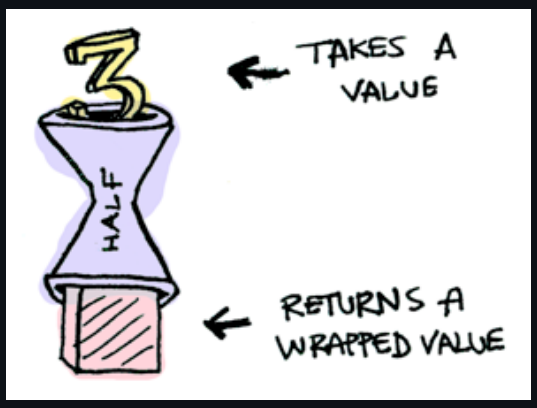
Let’s see an example. Good ol’ Maybe is a monad:



Just a monad hanging out

Suppose `half` is a function that only works on even numbers:

```
half = lambda x: Just(x // 2) if (x % 2== 0) else Nothing()
```



What if we feed it a wrapped value?

We need to use `|` (`>>=` in Haskell) to shove our wrapped value into the function. Here’s a photo of `|`:

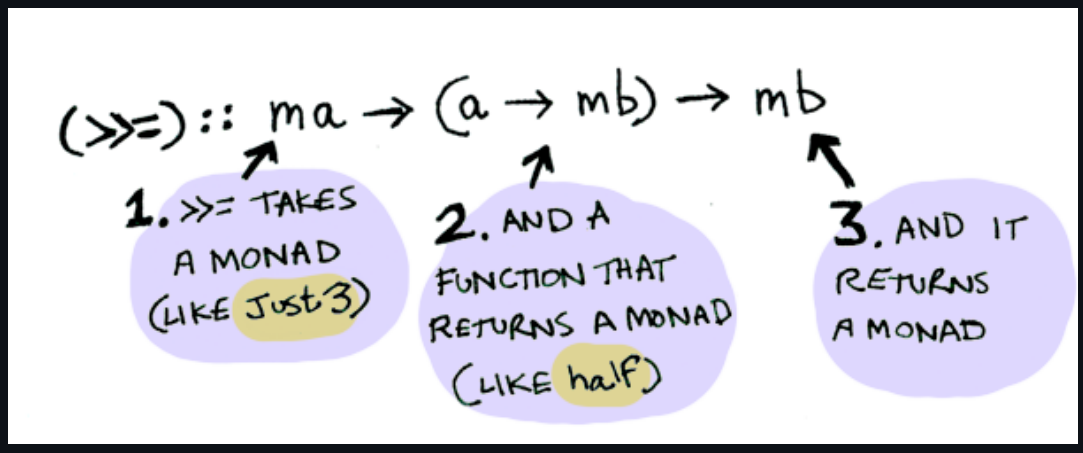
Here's how it works:

```
>>> Just(3) | half
Nothing
>>> Just(4) | half
Just 2
>>> Nothing() | half
Nothing
```

What's happening inside? Monad is another Abstract Base Class (typeclass in Haskell). Here's a partial definition:

```
class Monad(metaclass=ABCMeta):
    @abstractmethod
    def bind(self, func) -> "Monad":
        """
        :param Monad[A] self:
        :param Callable[[A], Monad[B]] func:
        :rtype: Monad[B]
        :returns: New Monad wrapping B
        """
```

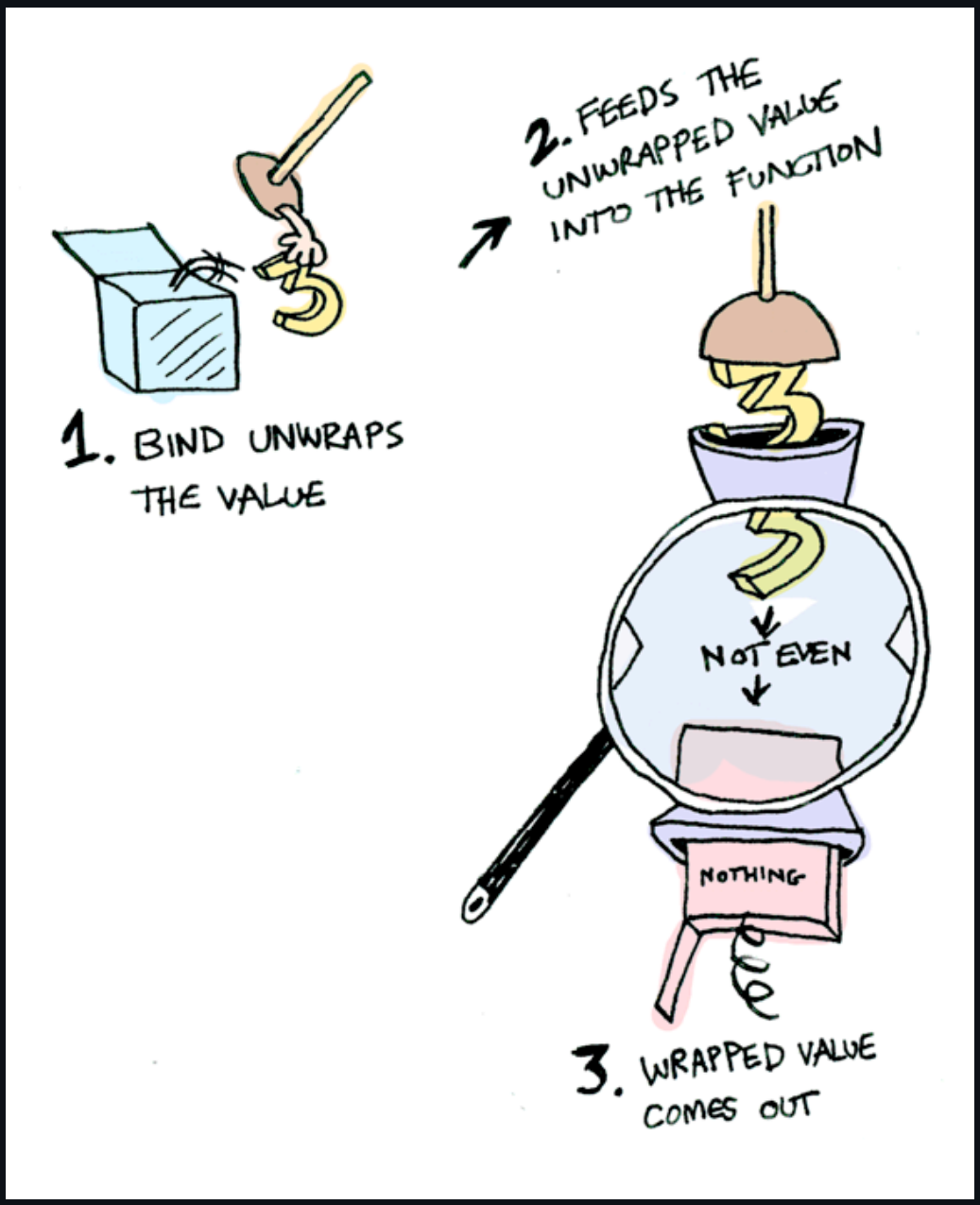
Where bind (`|` in Python, `>>=` in Haskell) is:



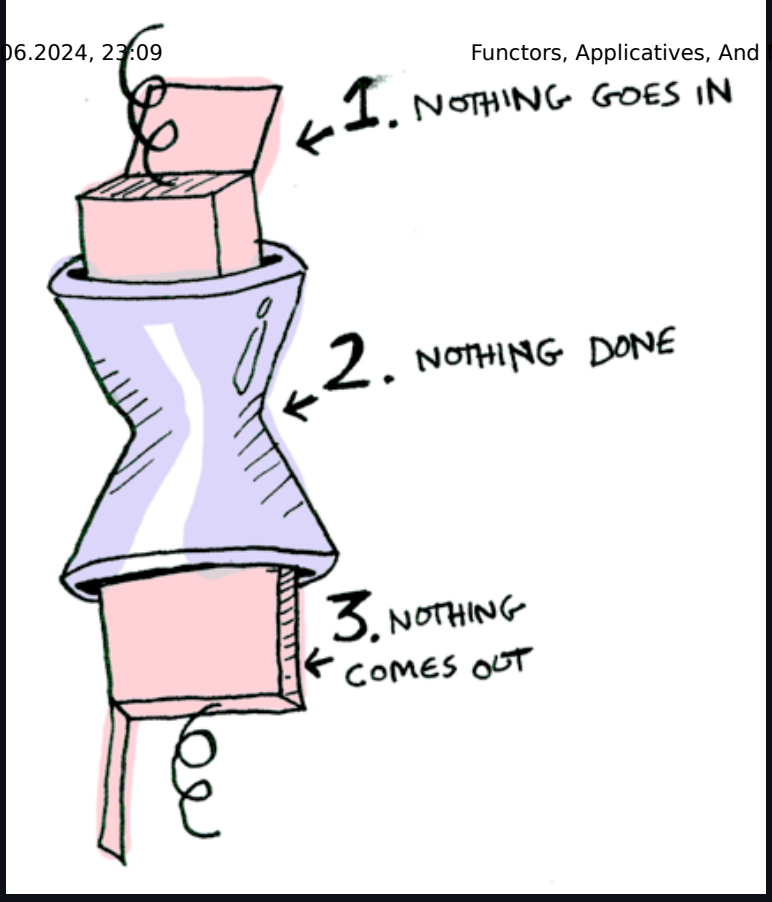
So `Maybe` is a Monad:

```python
class Maybe(Monad, Monoid, Applicative, Functor, metaclass=ABCMeta):
    ...

class Just(Maybe):
    def __init__(self, val):
        self.val = val

    def bind(self, func):
        return func(self.val)

    ...

class Nothing(Maybe):
    def bind(self, func):
        return Nothing()

    ...
```

Here it is in action with a Just 3!

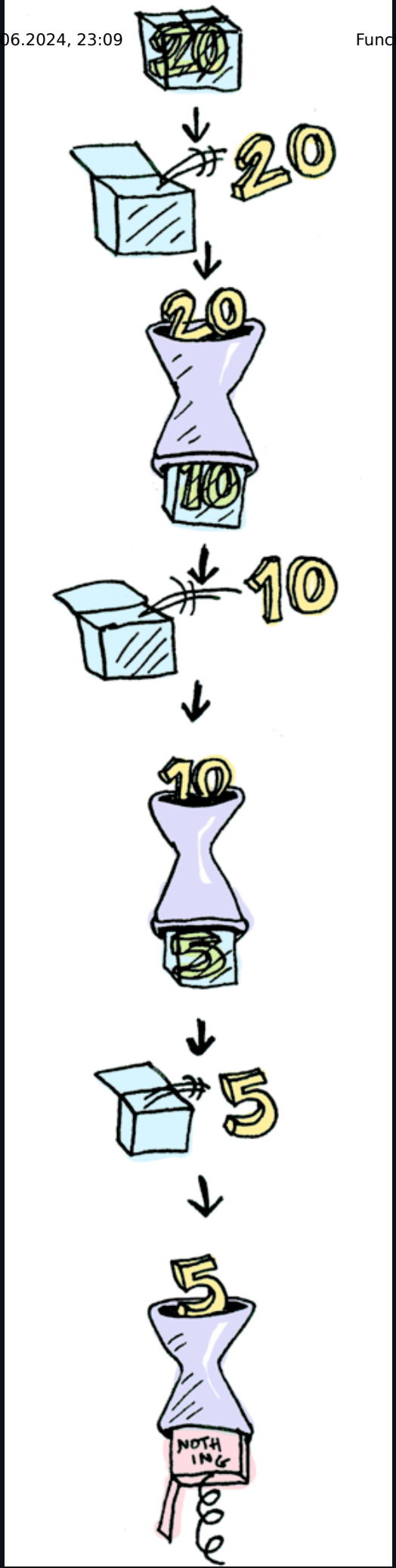


And if you pass in a `Nothing` it's even simpler:



You can also chain these calls:

```
>>> Just(20) | half | half | half
Nothing
```


20
20
20
10
10
10
5
5
NOTH
ING

Cool stuff! So now we know that Maybe is a Functor, an Applicative, and a Monad.

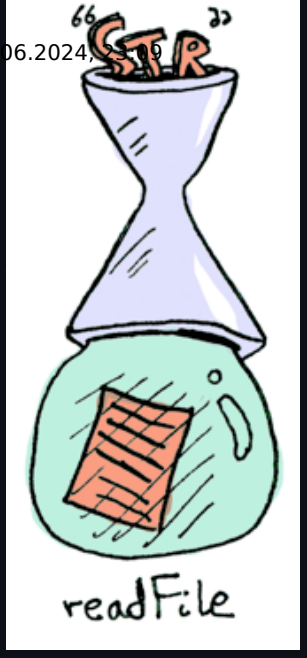
IO Monad

Now let's mosey on over to another example: the `IO` monad:

Specifically three functions. `get_line` (`getLine` in Haskell) takes no arguments and gets user input:

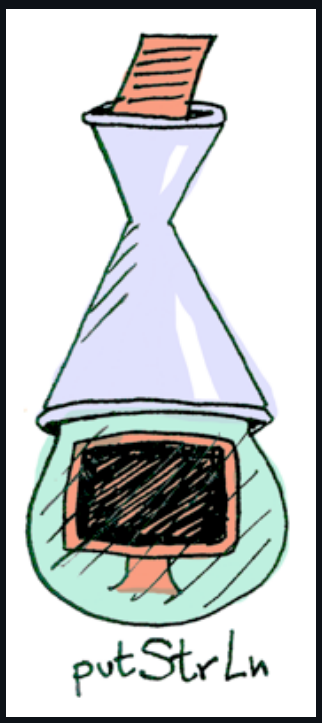

```
def get_line() -> IO:
    return Get(lambda s: IO(s))
```

`read_file` (`readFile` in Haskell) takes a string (a filename) and returns that file's contents:



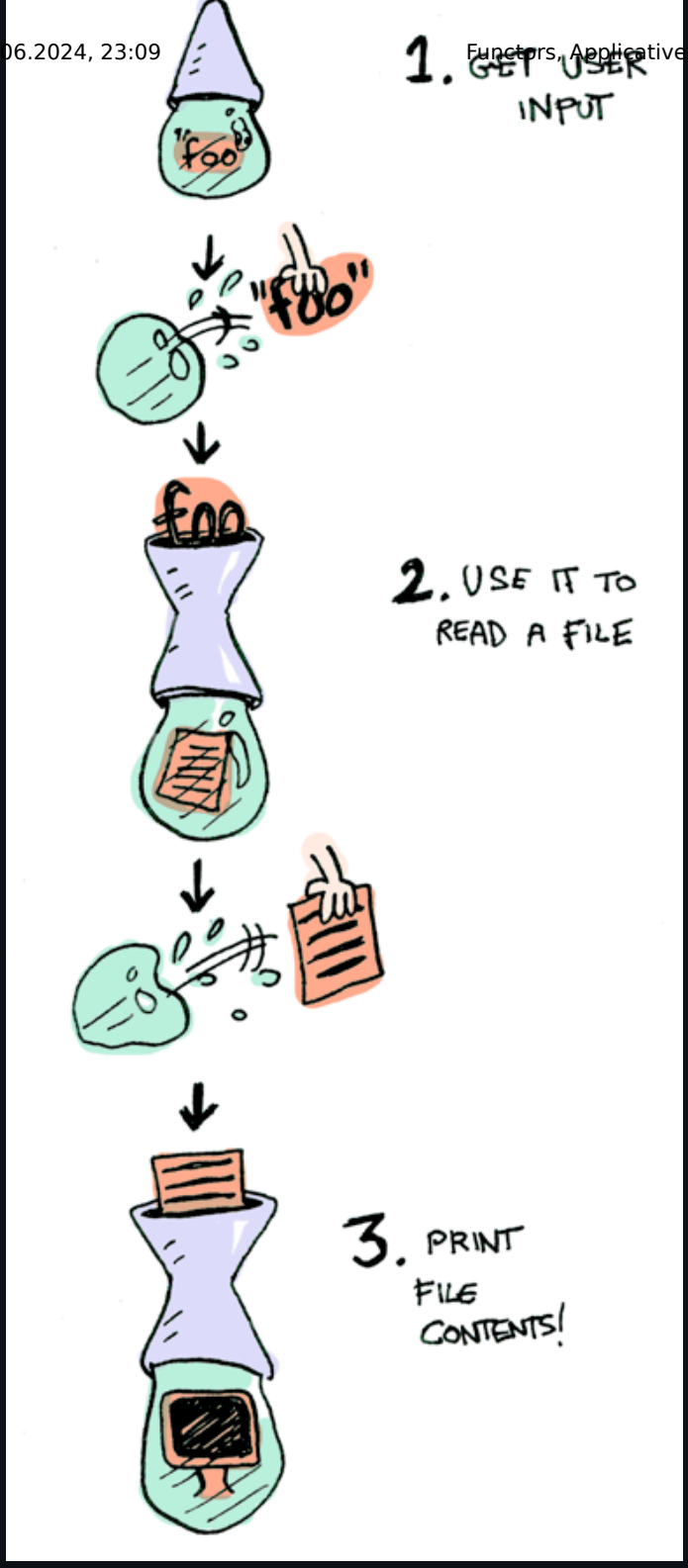

```
def read_file(filename) -> IO:
    return ReadFile(filename, lambda s: IO(s))
```

`put_line` (`putStrLn` in Haskell) takes a string and prints it:




```
def put_line(string) -> IO:
    return Put(string, IO(()))
```

All three functions take a regular value (or no value) and return a wrapped value. We can chain all of these using `|`!




```
get_line() | read_file | put_line
```

Aw yeah! Front row seats to the monad show!

Haskell also provides us with some syntactical sugar for monads, called do notation:

```
foo = do
    filename <- getLine
    contents <- readFile filename
    putStrLn contents
```

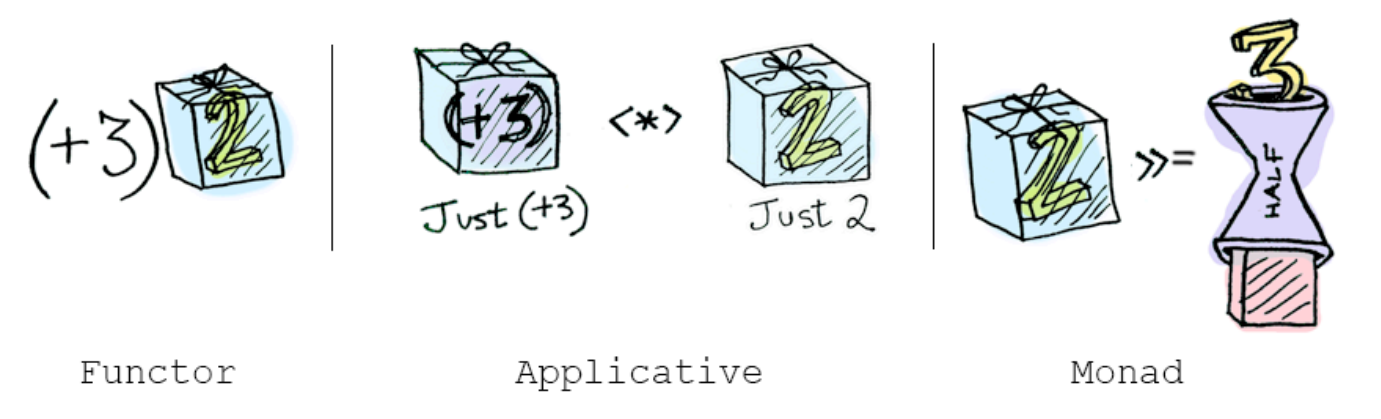
> Python does not have a do notation, so we have to write things a bit differently.

```
foo = get_line() | (lambda filename:
      read_file(filename) | (lambda contents:
      put_line(contents)))
```

Conclusion

1. A functor is a data type that implements the `Functor` abstract base class.
2. An applicative is a data type that implements the `Applicative` abstract base class.
3. A monad is a data type that implements the `Monad` abstract base class.
4. A `Maybe` implements all three, so it is a functor, an applicative, and a monad.

What is the difference between the three?



- **functors**: you apply a function to a wrapped value using `map` or `%`
- **applicatives**: you apply a wrapped function to a wrapped value using `*` or `lift`
- **monads**: you apply a function that returns a wrapped value, to a wrapped value using ´|´ or `bind`

So, dear friend (I think we are friends by this point), I think we both agree that monads are easy and a SMART IDEA(tm). Now that you've wet your whistle on this guide, why not pull a Mel Gibson and grab the whole bottle. Check out LYAH's section on Monads. There's a lot of things I've glossed over because Miran does a great job going in-depth with this stuff.

+ Add a custom sidebar

Clone this wiki locally

```
https://github.com/dbrattli/OSlash.wiki.git
```