Pycategories Documentation

Release 1.0.0

Daniel Hones

Jun 25, 2019

Contents

1 Installation **3**

2 Support and Contributing **5**

3 License **7**

4 User Guide **9**
- 4.1 Introduction . . . 9
- 4.2 Quickstart . . . 10
- 4.3 API . . . 12

5 Indices and tables **17**

Python Module Index **19**

Index **21**

Pycategories is a Python 3 library that implements ideas from category theory, such as monoids, functors, and monads. It provides a Haskell-influenced interface for defining instances of those typeclasses and defines several right out of the box, for example the Maybe monad:

```
>>> from categories import apply
>>> from categories.maybe import Just, Nothing
>>> f = Just(lambda x: x ** 2)
>>> x = Just(17)
>>> apply(f, x)
Just(289)
>>> apply(f, Nothing())
Nothing
```

Or to define your own instance of a typeclass:

```
>>> from categories import mappend, mempty, monoid
>>> monoid.instance(dict, lambda: {}, lambda a, b: dict(**a, **b))
>>> mappend({'foo': 'bar'}, {'rhu': 'barb'})
{'foo': 'bar', 'rhu': 'barb'}
```

CHAPTER 1

Installation

```
pip install pycategories
```

To clone the repo and install dependencies for development:

```
git clone https://gitlab.com/danielhones/pycategories
cd pycategories
pip install -e .[dev]
```

CHAPTER 2

Support and Contributing

- Issue Tracker
- Source code
- Contributing Guide

CHAPTER 3

License

Pycategories is licensed under the MIT License

CHAPTER 4

User Guide

4.1 Introduction

This library takes its inspiration from category theory and Haskell. Its goal is to provide a useful set of features that enable programming in a functional style in Python, with category theory as the guiding principles. If you are new to category theory or don't know Haskell, Category Theory for Programmers by Bartosz Milewski is a great textbook that is immediately useful as a programmer and it covers the basics of Haskell and Category Theory. It also has two sets of video lectures to accompany the text.

The main features of this library are provided by typeclasses and some data types that defines instances of those typeclasses. A typeclass is like a declaration of an interface for an object. Their use in this library is to define interfaces for algebraic structures that come from category theory. There are certain laws that each of those algebraic structures should obey, and the module for each typeclass defines functions that allow you to check whether a data type conforms to those laws. For examples on how to use those functions, take a look at the tests for Maybe or Python builtins in the Pycategories source code.

These are the typeclasses defined in Pycategories:

- Semigroup
- Monoid
- Functor
- Applicative
- Monad

These are the data types provided by Pycategories, with the specified typeclass instances defined:

- **Maybe**
  - Semigroup
  - Monoid
  - Functor
  - Applicative

- **Either**
  - Functor
  - Applicative
  - Monad
- **Validation**
  - Semigroup
  - Functor
  - Applicative

And typeclass instances are defined for the following built-in Python types:

- **List**
  - Semigroup
  - Monoid
  - Functor
  - Applicative
  - Monad
- **String**
  - Semigroup
  - Monoid
  - Functor
- **Tuple**
  - Semigroup
  - Monoid
  - Functor
  - Applicative
  - Monad

The behavior of the instances defined for Functor, Applicative, and Monad match the behavior of those definitions in Haskell. If you need something else for your application, you can redefine the instances for tuple, or define a new product data type that's isomorphic to a tuple.

4.2 Quickstart

4.2.1 Basics

You can get a lot done by just using the data types provided by Pycategories and the instances defined for some of Python's built-in types. For example, the following example script uses `Maybe` to convert the contents of a file to uppercase and print the result, or exit with no output if the file doesn't exist. This could be altered to use `Either` to display an error message instead of exiting with no output for non-existent files:

```
from functools import partial
import os
import sys

from categories import fmap
from categories.maybe import Just, Nothing
from categories.utils import compose


def maybe_read_file(path):
    if os.path.exists(path):
        with open(path, 'r') as f:
            return Just(f.read())
    else:
        return Nothing()


def upper_case_file(path):
    return fmap(lambda x: x.upper(),
                maybe_read_file(path))


if __name__ == '__main__':
    maybe_upper_case = compose(partial(fmap, print),
                               upper_case_file)
    maybe_upper_case(sys.argv[1])
```

Each data type defines a new class to represent the type, and one or more data constructors that create and return objects of that type. For example, the `Maybe` data type has two data constructors, `Just` and `Nothing`. To create a Maybe object, you would use one of those constructors:

```
>>> Just(23)
Just(23)
>>> Nothing()
Nothing
```

Data constructors are used just like Python functions. If a data constructor takes no arguments, then you still need a set of parentheses after it, unlike in Haskell. Even though the data constructor names are capitalized, like Python class names conventionally are, they are not currently implemented as separate classes, and are actually functions that return objects of the data type class. This should be considered an implementation detail of the library and you should not rely on that behavior in your application code, since it may change in the future.

There is basic pattern matching available via the `match()` function on the data types provided by Pycategories. This allows you to avoid doing manual checking of attributes of a data type. Currently, the match function only matches on the data constructor, and not on value. It takes a data constructor as an argument and returns a boolean indicating whether the object it was called on matches that constructor. For example, reusing the `maybe_read_file` function in the previous example:

```
data = maybe_read_file('/tmp/testfile.txt')

if data.match(Just):
    print("The file existed and contained:\n\n{}".format(data.value))
else:
    print("That file did not exist")
```

However, it is more common in functional programming to take the opposite approach and lift your normal functions into the context of a datatype using `bind`, `apply`, or `fmap` as appropriate, like the first example script does with

fmap.

4.2.2 Defining Instances

A primary use of this library is defining typeclass instances for your own data types. The Python module that defines each typeclass will have a function called `instance`, which lets you define your own instances. As an example, let's define monoid and functor instances for Python's built-in dictionary:

```
>>> from categories import monoid, functor
>>> monoid.instance(dict, lambda: {}, lambda a, b: dict(**a, **b))
>>> functor.instance(dict, lambda f, xs: {k: f(v) for k, v in xs.items()})
```

Now that we've done that, we can call `mappend` and `fmap` on a dictionary:

```
>>> from categories import fmap, mappend
>>> test = mappend({'x': 'foobar'}, {'y': 'bazquux'})
{'x': 'foobar', 'y': 'bazquux'}
>>> fmap(lambda x: x.upper(), test)
{'x': 'FOOBAR', 'y': 'BAZQUUX'}
```

To be truly useful, typeclass instances should obey certain laws that make them behave consistently. To help test whether they conform to those laws, the typeclass modules have functions that return a boolean indicating whether the instance obeys the law. To use them, you need to call them with some example values of the type you're testing; the specific arguments needed can be different for each law function. The files in the tests/ directory have lots of examples of using those functions. Here's a brief example testing the monoid and functor laws for dictionary:

```
>>> monoid.identity_law({'a': 'test', 'b': 'other'})
True
>>> monoid.associativity_law({'a': 'test'}, {'b': 'other'}, {'c': 'something else'})
True
>>> functor.identity_law({'a': 'test', 'b': 'other'})
True
>>> f = lambda x: x.upper()
>>> g = lambda y: y[0:3]
>>> functor.composition_law(f, g, {'x': 'foobar', 'y': 'bazquux'})
True
```

For details on what each typeclass needs when defining instances, refer to the API documentation.

4.3 API

4.3.1 Typeclasses

`categories.semigroup.`**`instance`**(*type*, *sappend*)

`categories.monoid.`**`instance`**(*type*, *mempty*, *mappend*)

`categories.functor.`**`instance`**(*type*, *fmap*)

`categories.applicative.`**`instance`**(*type*, *pure*, *apply*)

`categories.monad.`**`instance`**(*type*, *mreturn*, *bind*)

4.3.2 Data Types

class categories.maybe.**Maybe**(*type: str*, *value: A = None*)

class categories.either.**Either**(*type: str, value: Union[E, A]*)

class categories.validation.**Validation**(*type: str, value: Union[E, A]*)

4.3.3 Utilities

categories.utils.**compose**(**fs*) → Callable
Return a function that is the composition of the functions in `fs`. All functions in `fs` must take a single argument.

Adapted from this StackOverflow answer: https://stackoverflow.com/a/34713317

Example

```
>>> compose(f, g)(x) == f(g(x))
```

categories.utils.**flip**(*f: Callable[[A, B], Any]*) → Callable[[B, A], Any]
Return a function that reverses the arguments it's called with.

Parameters **f** – A function that takes exactly two arguments

Example

```
>>> exp = lambda x, y: x ** y
>>> flip_exp = flip(exp)
>>> exp(2, 3)
8
>>> flip_exp(2, 3)
9
```

categories.utils.**unit**(*x: Any*) → Any
The identity function. Returns whatever argument it's called with.

4.3.4 Contents

categories package

Submodules

categories.applicative module

class categories.applicative.**Applicative**(*pure*, *apply*)
Bases: object

categories.applicative.**composition_law**(*u*, *v*, *w*, *type_form=None*)

pure (.) <*> u <*> v <*> w == u <*> (v <*> w) where (<*>) = apply

categories.applicative.**homomorphism_law**(*f*, *x*, *_type*, *type_form=None*)

pure f <*> pure x == pure (f x) where (<*>) = apply

categories.applicative.**identity_law**(*v*, *type_form=None*)

pure id <*> v == v where (<*>) = apply

categories.applicative.**instance**(*type*, *pure*, *apply*)

categories.applicative.**interchange_law**(*u*, *y*, *type_form=None*)

u <*> pure y == pure ($ y) <*> u where (<*>) = apply

categories.applicative.**pure**(*x*, *_type*, *type_form=None*)

The optional type_form arg is an attempt to provide some more flexibility with the type that pure is required to cast to. For example, pure for the tuple Applicative instance needs to know the types of its elements so it can return the correct object:

pure(7, tuple, type_form=(str, int)) == ("", 7)

categories.builtins module

categories.either module

class categories.either.**Either**(*type: str, value: Union[E, A]*)

Bases: typing.Generic

classmethod left(*value: E*) → Etr

match(*constructor*) → bool

classmethod right(*value: A*) → Etr

categories.either.**either**(*f: Callable*, *g: Callable*, *x: categories.either.Either*) → categories.either.Either

Given two functions and an Either object, call the first function on the value in the Either if it's a Left value, otherwise call the second function on the value in the Either. Return the result of the function that's called.

Parameters

- **f** – a function that accepts the type packed in x
- **g** – a function that accepts the type packed in x
- **x** – an Either object

Returns Whatever the functions f or g return

categories.functor module

class categories.functor.**Functor**(*fmap*)

Bases: object

categories.functor.**composition_law**(*f*, *g*, *x*)

fmap (g . f) == fmap g . fmap f

categories.functor.**identity_law**(*x*)

fmap id == id

categories.functor.**instance**(*type*, *fmap*)

categories.instances module

class categories.instances.**Typeclass**

Bases: object

categories.instances.**make_adder**(*instances: Dict[Type[CT_co], categories.instances.Typeclass]*) → Callable[[Type[CT_co], categories.instances.Typeclass], None]

categories.instances.**make_getter**(*instances: Dict[Type[CT_co], categories.instances.Typeclass], name: str*) → Callable[[Type[CT_co]], categories.instances.Typeclass]

categories.instances.**make_undefiner**(*instances: Dict[Type[CT_co], categories.instances.Typeclass]*) → Callable[[Type[CT_co]], None]

categories.maybe module

class categories.maybe.**Maybe**(*type: str*, *value: A = None*)
Bases: typing.Generic

classmethod just(*value: A*) → M

match(*constructor*) → bool

classmethod nothing() → M

categories.maybe.**maybe**(*default: B, f: Callable, x: categories.maybe.Maybe[~A][A]*) → B
Given a default value, a function, and a Maybe object, return the default if the Maybe object is Nothing, otherwise call the function with the value in the Maybe object, call the function on it, and return the result.

Parameters

- **default** – This is the value that gets returned if x is Nothing
- **f** – When x matches Just, this function is called on the value in x, and the result is returned
- **x** – a Maybe object

Returns Whatever type default or the return type of f is

categories.monad module

class categories.monad.**Monad**(*bind*, *mreturn*)
Bases: object

categories.monad.**associativity_law**(*m*, *f*, *g*, *type_form=None*)
(m >>= f) >>= g == m >>= (x -> f x >>= g)

categories.monad.**instance**(*type*, *mreturn*, *bind*)

categories.monad.**left_identity_law**(*a*, *f*, *_type*, *type_form=None*)
mreturn a >>= f == f a

categories.monad.**mreturn**(*x*, *_type*, *type_form=None*)

categories.monad.**right_identity_law**(*m*, *type_form=None*)
m >>= return == m

Ex: Just 7 >>= return == Just 7

categories.monoid module

class categories.monoid.**Monoid**(*mempty*, *mappend*)
Bases: object

categories.monoid.**associativity_law**(*x*, *y*, *z*)

(x <> y) <> z = x <> (y <> z) where (<>) = mappend

categories.monoid.**identity_law**(*x*)
Assert left and right identity laws:

mempty <> x = x x <> mempty = x

where (<>) = mappend

categories.monoid.**instance**(*type*, *mempty*, *mappend*)

categories.monoid.**mempty**(*type*)

categories.utils module

categories.utils.**flip**(*f: Callable[[A, B], Any]*) → Callable[[B, A], Any]
Return a function that reverses the arguments it's called with.

Parameters **f** – A function that takes exactly two arguments

Example

```
>>> exp = lambda x, y: x ** y
>>> flip_exp = flip(exp)
>>> exp(2, 3)
8
>>> flip_exp(2, 3)
9
```

categories.utils.**funcall**(*f: Callable*, **args*) → Any

categories.utils.**id_**(*x: Any*) → Any
The identity function. Returns whatever argument it's called with.

categories.utils.**unit**(*x: Any*) → Any
The identity function. Returns whatever argument it's called with.

Module contents

CHAPTER 5

Indices and tables

- genindex
- modindex
- search

Python Module Index

C

categories, 16
categories.applicative, 13
categories.either, 14
categories.functor, 14
categories.instances, 14
categories.maybe, 15
categories.monad, 15
categories.monoid, 16
categories.utils, 16

Index

A

Applicative (*class in categories.applicative*), 13
associativity_law() (*in module categories.monad*), 15
associativity_law() (*in module categories.monoid*), 16

C

categories (*module*), 16
categories.applicative (*module*), 13
categories.either (*module*), 14
categories.functor (*module*), 14
categories.instances (*module*), 14
categories.maybe (*module*), 15
categories.monad (*module*), 15
categories.monoid (*module*), 16
categories.utils (*module*), 16
compose() (*in module categories.utils*), 13
composition_law() (*in module categories.applicative*), 13
composition_law() (*in module categories.functor*), 14

E

Either (*class in categories.either*), 13, 14
either() (*in module categories.either*), 14

F

flip() (*in module categories.utils*), 16
funcall() (*in module categories.utils*), 16
Functor (*class in categories.functor*), 14

H

homomorphism_law() (*in module categories.applicative*), 13

I

id_() (*in module categories.utils*), 16
identity_law() (*in module categories.applicative*), 13
identity_law() (*in module categories.functor*), 14
identity_law() (*in module categories.monoid*), 16
instance() (*in module categories.applicative*), 12, 13
instance() (*in module categories.functor*), 12, 14
instance() (*in module categories.monad*), 12, 15
instance() (*in module categories.monoid*), 12, 16
instance() (*in module categories.semigroup*), 12
interchange_law() (*in module categories.applicative*), 14

J

just() (*categories.maybe.Maybe class method*), 15

L

left() (*categories.either.Either class method*), 14
left_identity_law() (*in module categories.monad*), 15

M

make_adder() (*in module categories.instances*), 14
make_getter() (*in module categories.instances*), 15
make_undefiner() (*in module categories.instances*), 15
match() (*categories.either.Either method*), 14
match() (*categories.maybe.Maybe method*), 15
Maybe (*class in categories.maybe*), 13, 15
maybe() (*in module categories.maybe*), 15
mempty() (*in module categories.monoid*), 16
Monad (*class in categories.monad*), 15
Monoid (*class in categories.monoid*), 16
mreturn() (*in module categories.monad*), 15

N

nothing() (*categories.maybe.Maybe class method*), 15

P

pure() (*in module categories.applicative*), 14

R

right() (*categories.either.Either class method*), 14
right_identity_law() (*in module categories.monad*), 15

T

Typeclass (*class in categories.instances*), 14

U

unit() (*in module categories.utils*), 13, 16

V

Validation (*class in categories.validation*), 13