James Powell on Core Concepts of Python

The protocol view of python

If you look at the object orientation of python there are three core patterns we have to understand. One of them is protocol view of python.

In [ ]:

```
class Polynomial:
    pass

p1 = Polynomial()
p2 = Polynomial()
p1.coeffs = 1, 2, 3 #x**2 + 2x + 3
p2.coeffs = 3, 4, 3 #x**2 + 4x + 3
```

__init__

Why write in 4 lines when we can write it in two lines? Two lines will do.

In [ ]:

```
class Polynomial:
    def __init__(self, *coeffs):
        self.coeffs = coeffs


p1 = Polynomial(1, 2, 3)
p2 = Polynomial(3, 4, 3)

print(p1)
print(p2)
```

__repr__

That looks ugly because we are missing the method that corresponds to what happens when we call top level repr function to figure out the representation of our python object.

In [ ]:

```
class Polynomial:
    def __init__(self, *coeffs):
        self.coeffs = coeffs

    def __repr__(self):
        return 'Polynomial(*{!r})'.format(self.coeffs)


p1 = Polynomial(1, 2, 3)
p2 = Polynomial(3, 4, 3)

print(p1)
print(p2)
```

__add__

What about adding polynomial objects together?

In [ ]:

```
class Polynomial:
    def __init__(self, *coeffs):
        self.coeffs = coeffs

    def __repr__(self):
        return 'Polynomial(*{!r})'.format(self.coeffs)

    def __add__(self, other):
        return Polynomial(*(x + y for x, y in zip(self.coeffs, other.coeffs)))


p1 = Polynomial(1, 2, 3)
p2 = Polynomial(3, 4, 3)

print(p1)
print(p2)

print(p1 + p2)
```

What pattern do we see?

If we have some behaviour we want to implement --> we write some __function__ to implement it. This functions are called dunder methods.

It has also an abhorrent name *data model methods*. Python documentation contains a whole list of this methods. Whenever we want to implement some behaviour in python we want to tell python "for this arbitary object do this behaviour, like give the printable representation for this object, perform an addition of this objects."

The pattern is: there is a top-level function or some top-level syntax and there is a corresponding dunder function. The exact arguments that the underscore function takes are determined by python documentation. The default names of the arguments can be picked up by the documentation also.

But there is something more fundamental than this. If we want to implement the summation of two objects x + y --> we implement __add__, if we want to define how we instantiate an object --> we implement __init__.

__len__

What a size of a polynomial might me? The highest degree of the polynomial. We will implement __len__ function to tell us the size of the polynomial

In [ ]:

```
class Polynomial:
    def __init__(self, *coeffs):
        self.coeffs = coeffs

    def __repr__(self):
        return 'Polynomial(*{!r})'.format(self.coeffs)

    def __add__(self, other):
        return Polynomial(*(x + y for x, y in zip(self.coeffs, other.coeffs)))

    def __len__(self):
        return len(self.coeffs)


p1 = Polynomial(1, 2, 3)
p2 = Polynomial(3, 4, 3)

print(len(p1))
```

The pattern as we see is as follows:

The python data model is a means by which we can implement protocols. Those protocols have some abstract meaning depending on the object itself. In the case of a polynomial summation means whatever it ment in the math class.

To get its printable representation we use __repr__. Printable representation is typically whatever string we have to type in the console to get a new instance of that object.

In each case that protocol exists, there is some underscore method that implements this protocol, there is some top-level function like len or top-level syntax like + sign that allows us to invoke that protocol and it all fits together.

When we implement smth like __len__ we do that by delegating back to the protocol itself. __len__ is implemented in terms of len() being called on a constituent object. __add__ is implemented by adding up some components.

If we have some top-level syntax like parentheses that come after the object x(), we call it the __call__ protocol.

In [ ]:

```
class Polynomial:
    def __init__(self, *coeffs):
        self.coeffs = coeffs

    def __repr__(self):
        return 'Polynomial(*{!r})'.format(self.coeffs)

    def __add__(self, other):
        return Polynomial(*(x + y for x, y in zip(self.coeffs, other.coeffs)))

    def __len__(self):
        return len(self.coeffs)

    def __call__(self):
        #no idea what should polynomial do
        pass


p1 = Polynomial(1, 2, 3)
p2 = Polynomial(3, 4, 3)
```

Metaclasses

It is very clear and very obvious for why and when you use it and it is someting you can kind of shelf away in your mind as "oh, this feature does this, this is why I want to use it, i don't have to use it all the time".

Let's imagine there are two teams on the project and one team writes libraty code and the other team writes user code. People on the user side cant touch library code.

In [ ]:

```
#library.py

class Base:
    def foo(self):
        return 'foo'
```

In [ ]:

```
#user.py
from library import Base

class Derived(Base):
    def bar(self):
        return self.foo()

#this code can break if there is no foo method.
```

What we can do to figure out our code brakes before the runtime? We can write a test for example. We will see that the code fails some time before it hits the runtime production environment.

But maybe there is one thing that we can add to this code so it fails before it hits the runtime production environment?

We can add assert and see if the Base has no foo method before we even came up to derived class.

In [ ]:

```
#user.py
from library import Base

assert hasattr(Base, 'foo'), "You broke it!"

class Derived(Base):
    def bar(self):
        return self.foo()
```

So now we have an early warning tha this has broken. And what we are trying to do here is enforce a constraint. The user level enforces a constraint on a library level, in other words, the Derived class is enforcing a constraint on a base class. The Derived class says "the Base class should have this characteristics in order for me to run and be happy. If it does not foo method implemented i will fail."

Let's say instead, we are working on a library side and we don't want people to screw code on a user side of the project. And we can't change anything on the user side of the code. And how do we make sure that the code will be implemented on the user side? There are three answers to this. One is metaclasses.

In [ ]:

```
#library2.py

class Base:
    def foo(self):
        return self.bar()

#we write this Base class under the assumption that developer in user department
#will implement bar() method, because if they don't, everything falls apart.

#we can also place try catch but that will only catch error during runtime.
```

In [ ]:

```
#user.py
from library2 import Base

class Derived(Base):
    def bar(self):
        return 'bar'
```

The reason we can call python a protocol oriented language is not just because the python data model is protocol oriented, but the entire python language itself has a notion of hooks and protocols and safety valves within it.

Python code runs from top to bottom linearly and almost every statement except the two of them are actually executable rutime code. In C++ or Java a class statement is not an executable code. In python it is.

In [ ]:

```
def _():
    class Base:
        pass

from dis import dis
dis(_)
```

LOAD_BUILD_CLASS is actual executable runtime instruction in the python enterpreteur for creating a class. There typycally in python tends to be a correspondence between some top-level syntax or function and some underscore method that implements that function. There happens to be some top-level mechanism here for building a class (it is not explicitly a function). It turns out in python there is a hook, there is an underscore function that allows us to do things with the process of building classes.

In [ ]:

```
def _():
    class Base:
        pass


#there is a function __build_class in python. It sits on a module builtins
old_bc = __build_class__
def my_bc(*a, **kw):
    print('my buildclass -->', a, kw)
    return old_bc(*a, **kw)
#we capture the original build class, write our own build clas

import builtins
#we make import from builtins ans swap out our classes
#so we are patching creating classes in python
builtins.__build_class__ = my_bc
```

In [ ]:

```
#user.py
from library2 import Base

class Derived(Base):
    def bar(self):
        return 'bar'
```

So we can actually catch the building of the class! We passed a function, the name of the class and the base class

So we can add our assert into the building class.

In [ ]:

```
class Base:
    def foo(self):
        return self.bar


#there is a function __build_class in python. It sits on a module builtins
old_bc = __build_class__
def my_bc(fun, name, base=None, **kw):
    if base is Base:
        print('check if bar method defined')
    if base is not None:
        old_bc(fun, name, base, **kw)
    return old_bc(fun, name, **kw)
#we capture the original build class, write our own build clas

import builtins
#we make import from builtins ans swap out our classes
#so we are patching creating classes in python
builtins.__build_class__ = my_bc
```

In [1]:

```
#user.py
from library2 import Base

class Derived(Base):
    def bar(self):
        return 'bar'
```

```
check if bar method defined
```

So protocol orientation is quite a fundamental piece of python. Almost everything python does in an execution context (like creating functions, importing modules, etc.) we can find a hook to get into that and, for example, add a check for existing the module.

In [ ]:

```

```