ABSTRACT BASE CLASSES

A SMART USE OF METACLASSES IN PYTHON

LEONARDO GIORDANI @LGIORDANI

http://thedigitalcatonline.com

About Me

Born in 1977 with Star Wars, bash, Apple][, BSD, finger, Zork, Galaxy Express 999, Little Pollon, Dire Straits, The Police, Rumours, The Silmarillion, Squad Leader.

Interested in operating systems and computer languages, photography, fantasy and science fiction, video- and boardgames, guitar playing, climbing, horseback riding, Aikido, fencing, rollerskating, drawing, painting, bookbinding.

I programmed in Z80 and x86 Assembly, GW-Basic, Logo, Borland Turbo Pascal, Prolog, C, C++, PHP, Lisp, Ada, Objective-C, bash, Python, Erlang, Clojure, Scala, JavaScript.

I love mathematics and cryptography.

tl; dr

me = nerd + coder

Overview

Level 1

Polymorphism - Behaviours and delegation - Collections

Level 2

Registering - Abstract Base Classes - Categories

Level 3

Build your ABCs - Metaclasses - ABCs as interfaces

LEVEL 1

OOP: BEHAVIOUR OVER STRUCTURE

Python EAFP
Test the behaviour, not the structure

```
try:
    someobj[1]
except TypeError:
    # object is not subscriptable
    ...
```

Example of behaviour

```
class ListView(View):
    def status(self):
        ...
```

```
class ItemView(View):
    def status(self):
        ...
```

How to check how an object behaves?

```
try:
    someobj.status
except AttributeError:
    ...
```

How to check if something behaves like a list?

```
try:
    someobj.append
    someobj.extend
    someobj.index
except AttributeError:
    ...
```

This is checking the structure: wrong

Temptation

```
if isinstance(someobj, list):
    ...
```

This is checking the type: wrong

Temptation - Part 2

```
if isinstance(someobj, list) or \
        isinstance(someobj, tuple) or \
        isinstance(someobj, str):
    ...
```

And quickly becomes unmaintainable

Perfect solution

```
if behaveslike(someobj, ListBehaviour):
    ...
```

PYTHON IS BASED ON DELEGATION

Original version of isinstance()*

```
def isinstance(obj, _class):
    return _class in obj.__class__.__bases__
```

*simplified version

New version of isinstance()*

```
def isinstance(obj, _class):
    return _class.__instancecheck__(obj)
```

*simplified version

This is now possible

```
if isinstance(someobj, ListBehaviour):
    ...
```

We just need to implement “behaviour” classes

Collections
classes that represent interesting behaviours

```
>>> import collections
>>>
>>> isinstance(“a string”, collections.Sequence)
True
>>>
>>> isinstance(“a string”, collections.Mapping)
False
>>>
```

https://docs.python.org/3/library/collections.abc.html

LEVEL 2

BEING A SUBCLASS IN PYTHON

Either a class is a REAL subclass

```
class ChildClass(ParentClass):
    pass
```

ChildClass knows ParentClass
ParentClass doesn't know ChildClass

Or a VIRTUAL subclass

```
ParentClass.register(ChildClass)
```

ParentClass knows ChildClass
ChildClass doesn't know ParentClass

Classes that can register other classes are called Abstract Base Classes or ABCs.

```
...
Sequence.register(tuple)
Sequence.register(str)
Sequence.register(range)
...
MutableSequence.register(list)
...
```

https://github.com/python/cpython/blob/master/Lib/_collections_abc.py

REGISTERING IS A PROMISE: NO CHECK!

Example

```
>>> import collections
>>> class MyClass:
...     pass
...
>>>
```

Example

```
>>> import collections
>>> class MyClass:
...     pass
...
>>> issubclass(MyClass, collections.Sequence)
False
>>>
```

Example

```
>>> import collections
>>> class MyClass:
...     pass
...
>>> issubclass(MyClass, collections.Sequence)
False
>>> collections.Sequence.register(MyClass)
<class '__main__.MyClass'>
>>>
```

Example

```
>>> import collections
>>> class MyClass:
...     pass
...
>>> issubclass(MyClass, collections.Sequence)
False
>>> collections.Sequence.register(MyClass)
<class '__main__.MyClass'>
>>> issubclass(MyClass, collections.Sequence)
True
>>>
```

ABSTRACT BASE CLASSES ARE CATEGORIES

LEVEL 3

Build your own Abstract Base Class (straight from the docs)

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

WHY METACLASSES?

DON'T PANIC

(The Hitchhiker's Guide to the Galaxy)

When you build an instance you use a class. The class can put things into the instance.

```
# Class definition
class Book():
    def __init__(self):
        self.answer = 42

# Link instance and class
b = Book()

# Use the instance
assert b.answer == 42
```

When you build a class you use a metaclass. The metaclass can put things into the class.

```
# Metaclass definition
class AnswerType(type):
    def __init__(self, name, bases, namespace):
        self.answer = 42

# Link class and metaclass
class Book(metaclass=AnswerType): pass

# Use the class
assert Book.answer == 42
```

Let’s compare the two side by side

```
# Class definition
class Book():
    def __init__(self):
        self.answer = 42

# Link instance and class
b = Book()

# Use the instance
assert b.answer == 42
```

```
# Metaclass definition
class AnswerType(type):
    def __init__(self, ...):
        self.answer = 42

# Link class and metaclass
class Book(metaclass=AnswerType): pass

# Use the class
assert Book.answer == 42
```

IT. IS. SIMPLE.

Here the metaclass puts the register() method in the new class

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass

MyABC.register(tuple)

assert issubclass(tuple, MyABC)
assert isinstance((), MyABC)
```

Metaclasses are inherited

```
from abc import ABCMeta

class MyABC(metaclass=ABCMeta):
    pass

class OtherABC(MyABC):
    pass

OtherABC.register(tuple)

assert issubclass(tuple, OtherABC)
assert issubclass(tuple, MyABC)
```

You can use ABCs to build interfaces

```
from abc import ABCMeta, abstractmethod

class StatusInterface(metaclass=ABCMeta):

    @abstractmethod
    def status(self):
        pass

si = StatusInterface()
```

TypeError: Can't instantiate abstract class StatusInterface with abstract methods status

You can use ABCs to build interfaces

```
from abc import ABCMeta, abstractmethod

class StatusInterface(metaclass=ABCMeta):

    @abstractmethod
    def status(self):
        pass

class MyStatusView(StatusInterface):
    pass

msv = MyStatusView()
```

TypeError: Can't instantiate abstract class MyStatusView with abstract methods status

QUESTIONS & ANSWERS

AMQP - architectures - C - Clojure - compilers
concurrent programming - C++ decorators - Django - Erlang
functional programming - generators - Git - metaclasses
metaprogramming - Notebook - OOP - operating systems
Python - Python2 - Python3 - RabbitMQ - Scala - TDD
testing - versioning

HTTP://THEDIGITALCATONLINE.COM

@thedigicat

THANKS