OOP and Python 3+

Jefferson O. Silva 9408421
Juliana Clemente 7345727
Rogério Cardoso 5022470

Abundant syntax brings more burden than help

Guido Van Rossum (author of Python)

1

Things about Python

A (very) brief contextualization...

What is it? And What can it do?

- Python is a widely used **high-level**, **general-purpose**, and **interpreted** language
- Python supports **multiple programming paradigms**, such as OOP and functional programming
- Python features a **dynamic type system** and **automatic memory management** and has a **comprehensive standard library**

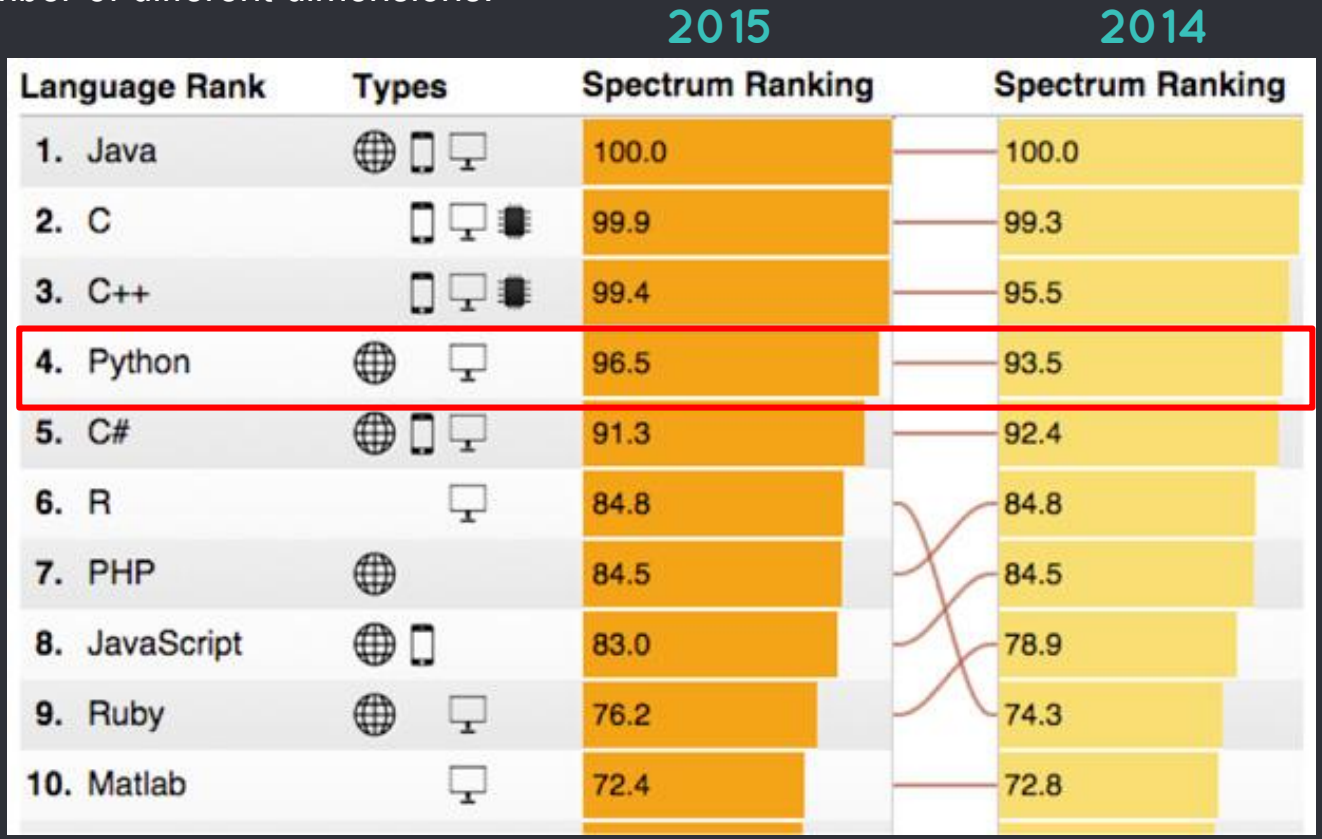
More on this at:

https://docs.python.org/2/faq/general.html#why-was-python-created-in-the-first-place
https://en.wikipedia.org/wiki/Python_(programming_language)

Most popular languages of 2015

Spectrum ranking

"Our ranking system [Spectrum's] is driven by weighting and combining 12 metrics from 10 data sources. We believe these sources—such as the IEEE Xplore digital library, GitHub, and CareerBuilder—are good proxies for the popularity of 48 languages along a number of different dimensions."



More on this at:

http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

2

Python and OOP

The most important things...

Class definition and object instantiation/use

- **class definition syntax:**

```
class MyPerson:
    pass
```

CamelCase

- **object instantiation syntax:**

```
my_person = MyPerson()
```

snake_case

- **attributes and method invocation:**

```
my_person.attribute_name
my_person.method_name()
```

More on this at:

https://www.python.org/dev/peps/pep-0008/#function-names'

Indent, otherwise Python won't play with you

Encapsulation

Encapsulation is generally used to refer to one of two related but distinct notions, and sometimes to the combination:

1. A language construct that facilitates the **bundling of data** with the methods (or other functions) operating on that data.

2. A language mechanism for **restricting direct access** to some of the object's components.

More on this at:
https://en.wikipedia.org/wiki/Encapsulation_(computer_programming)

Encapsulation

1. Python mechanisms for bundling data

```
class Person:
```

class member -> `i = 12345`

method -> `def __init__(self):`

private instance member -> `self.__age = 30`

public instance member -> `self.name = “Jeff”`

Encapsulation

2. Python mechanisms for hiding information

In Python, there are no attribute access modifiers such as **protected** or **private**. All attributes are **public**.

But there is a **convention** to define private: adding "__" (double underscore) in front of the variable or function name can hide them from outsiders

Encapsulation

2. Python mechanisms for hiding information

```
class Person:
    def __init__(self):
        self.name = "Jeff"
        self.__age = 30

    def print_name(self):
        print(self.name)
        print(self.__age)

>>> p = Person()
>>> p.name   SUCCESS
>>> p.__age  FAIL

>>> p.print_name() SUCCESS
```

Information hiding

"**Information hiding** is the **principle** of **segregation** of the **design decisions** in a computer program that are **most likely to change**, thus protecting other parts of the program from extensive modification if the design decision is changed.

The **protection** involves providing a **stable interface** which protects the remainder of the program from the implementation (the details that are most likely to change)" Wikipedia

More on this at:

https://en.wikipedia.org/wiki/Information_hiding

http://www.javaworld.com/article/2075271/core-java/encapsulation-is-not-information-hiding.html?page=3

Breaking Encapsulation

```
class SecretString:
    def __init__(self, plain_string, pass_phrase):
        self.__plain_string = plain_string
        self.__pass_phrase = pass_phrase

    def decrypt(self, pass_phrase):
        if (pass_phrase != self.__pass_phrase):
            raise Exception("Unauthorized access")
        return self.__plain_string

secret_string = SecretString("ACME: Top Secret", "pwd")
print( secret_string.decrypt("pwd") )
```

'ACME Top Secret' **-> OK. SHOULD BE ABLE TO ACCESS IT WITH THE PASSWORD**

```
print( secret_string._SecretString__plain_string )
```

'ACME Top Secret' **-> NOT OK. SHOULDN'T BE ABLE TO ACCESS IT WITHOUT THE PASSOWORD**

More on this at:
http://www.javaworld.com/article/2075271/core-java/encapsulation-is-not-information-hiding.html?page=3

... it's the Python way

If your application can't afford encapsulation to be broken in this way, perhaps you should consider using another language

Inheritance

- Example

```
class Person:
  def speak(self):
    return "I can speak"

class Man(Person):
  def wear(self):
    return "I wear shirt"

class Woman(Person):
  def wear(self):
    return "I wear skirt"

man = Man()
print( man.wear() )
print( man.speak() )
```

Definition ommited for brevity

Multiple Inheritance

Python supports a limited form of multiple inheritance

```
class DerivedClass(C1,C2,...,Cn):
    pass
```

The resolution rule is depth-first, left-to-right. Thus, if an attribute or method is not found in DerivedClass, it is searched in C1, then recursively in the super classes of C1, and only if it is not found there, it is searched in C2, and so on.

Multiple Inheritance
Example

What will be printed in the method calls?

`print( c.m1() )`
Since C does not implement m1, it will take the first superclass from the left(A). Since m1 is implemented in A, it will call A.m1() and print its result: "I am A"

`print( c.m2() )`
C does not implement m2. C will look m2 in A (first from the left). Finally, C will find m2 in B, which will print its result: "I am B"

```
class A:
    def m1(self):
        return "I am A"
class B:
    def m1(self):
        return "I am BA"
    def m2(self):
        return "I am B"
class C(A,B):
    def m3(self):
        return "I am C"

c = C()
print( c.m1() )
print( c.m2() )
```

Polymorphism

```
class AudioFile:
    def __init__(self, filename):
        if not filename.endswith(self.ext):
            raise Exception("Invalid file format")
        self.filename = filename

class MP3File(AudioFile):
    ext = "mp3"
    def play(self):
        print("playing {} as mp3".format(self.filename))

class WavFile(AudioFile):
    ext = "wav"
    def play(self):
        print("playing {} as wav".format(self.filename))

music = MP3File("myfile.mp3")
music.play()
```

More on this at:

https://en.wikipedia.org/wiki/Polymorphism_(computer_science)

Python's Duck Typing

```
class Duck:
    def quack(self):
        print("Quack, quack!");
    def fly(self):
        print("Flap, Flap!");

class Person:
    def quack(self):
        print("I'm Quackin'!");
    def fly(self):
        print("I'm Flyin'!");

def in_the_forest(mallard):
    mallard.quack()
    mallard.fly()

in_the_forest(Duck())
in_the_forest(Person())
```

Output:

```
Quack, quack!
Flap, Flap!
I'm Quackin'!
I'm Flyin'!
```

Functions are also objects

- Using functions as attributes

```
def dobra(x):
    return 2*x
def metade(x):
    return x/2
def operate(func, x):
    result = func(x)
    return result

>>> operate(dobra, 10)
20
>>>operate(metade, 10)
5
```

Constructors and Initializers

__new__(cls[,...])

__new__ is the first method to get called in an object's instantiation. It takes the class, then any other arguments that it will pass along to __init__. __new__ is used fairly rarely, but it does have its purposes, particularly when subclassing an immutable type like a tuple or a string.

__init__(self[,...])

The initializer for the class. It gets passed whatever the primary constructor was called with (so, for example, if we called x = SomeClass(10, 'foo'), __init__ would get passed 10 and 'foo' as arguments. __init__ is almost universally used in Python class definitions.

More on this at:

http://www.rafekettler.com/magicmethods.html#construction

https://www.python.org/download/releases/2.2/descrintro/#__new__

Constructors and Initializers

Implementation

```
class A(object):  # -> don't forget the object specified as base

    def __new__(cls):
        print "A.__new__ called"
        return super(A, cls).__new__(cls)

    def __init__(self):
        print "A.__init__ called"

A()
```

the output will be:

```
A.__new__ called
A.__init__ called
```

Finalizers

A **finalizer** is a method that **performs** some form of **cleanup** and is **executed during object destruction**, prior to the object being deallocated.

The term "**finalizer**" is primarily used in object-oriented languages that use **garbage collection** such as Java. This is contrasted with a "**destructor**" (__del__), which is a method called for finalization in languages with **deterministic object lifetimes** such as C++. These are generally exclusive – a language will have either finalizers (if garbage collected) or destructors (if deterministic).

Per spec, **Python is garbage collected**, but the reference **CPython** implementation uses **reference counting**. This reflects the fact that **reference counting results in semi-deterministic object lifetime**: for objects that are not part of a cycle, objects are destroyed deterministically when the reference count drops to zero, but objects that are part of a cycle are destroyed non-deterministically, as part of a separate form of garbage collection.

More on this at:

https://en.wikipedia.org/wiki/Finalizer

https://blogs.msdn.microsoft.com/ericlippert/2010/01/21/whats-the-difference-between-a-destructor-and-a-finalizer/

3 Python Object-Oriented Topics

Python Built-in functions

Reversed:

```
>>> a = [1, 2, 3]
>>> for i in reversed(a):
>>>     print(i)

3
2
1
```

Python Built-in functions

Enumerate:

```
>>> rgb = ['red', 'green', 'blue']
>>> list(enumerate(rgb))
[(0,'red'),(1,'green'),(2,'blue')]
>>> list(enumerate(rgb, start=1))
[(1,'green'),(2,'blue')]
```

Python Built-in functions

Zip:

```
>>> v1 = [1,2,3]
>>> v2 = [4,5,6]
>>> for a,b in zip(v1, v2):
>>>     print(a,'*',b,'=', a*b)
1*4= 4
2*5= 10
3*6= 18
```

Generators

Once the generator's function code reaches a "yield" statement, the generator yields its execution back to the for loop, returning a new value from the set.

```
def yrange(n):
    i = 0
    while i < n:
        yield i
        i += 1
```

Python Decorator

A **decorator** is a **callable that returns a callable**. Basically, a decorator takes ina function, adds some functionality and returns it.

The **goal** of Python Decorators is to **inject or modify code** in functions or classes. It is usually a simpler **alternative to metaclasses**.

More on this at:

http://www.programiz.com/python-programming/decorator

Python Decorator

```
def warm(func):
    def inner():
        print("I got decorated")
        func()
    return inner

def weather():
    Print("Nice weather")

>>> weather()
Nice weather
```

Python Decorator

```
>>> # let's decorate this weather function
>>> season = warm(weather)

>>> season()
I got decorated
Nice weather
```

In this case, warm is a decorator. Generally, we decorate a function and reassign it as:

```
weather = warm(weather)
```

Python Decorator

As a shortcut, we can use the @ symbol along with the name of the decorator function and place it above the definition of the function to be decorated.

```
@warm
def weather():
    print("Nice weather")
```

is equivalent to

```
def weather():
    print("Nice weather")
weather = warm(weather)
```

Properties

We can use Python properties to make the access of methods look like the access to a class attribute.

In many languages, properties are implemented as a pair of accessor (or getter) / mutator (or setter) methods, but accessed using the same syntax as for public fields. Omitting a method from the pair yields a read-only or an uncommon write-only property.

More on this at:
https://docs.python.org/3/howto/descriptor.html#descriptor-example

Properties

Manually implemented

```
class Celsius:
  def __init__(self, temp = 0):
    self.temp = temp

  def get_temp(self):
    print("Getting value")
    return self._temp

  def set_temp(self, value):
    if value < -273:
      raise ValueError("Too low")
    print("Setting value")
    self._temp = value

  temp = property(get_temp,set_temp)
```

How to Use properties?

```
>>> c = Celsius()
Setting value


>>> c.temp
Getting value
0


>>> c.temp = 37
Setting value
```

Properties

Implementation with decorators

```
class Celsius:
    def __init__(self, temp = 0):
    self._temp= temp

    @property
    def temp(self):
        print("Getting value")
        return self._temp

    @temp.setter
    def temp(self, value):
        if value < -273:
            raise ValueError("Too low")
        print("Setting value")
        self._temp = value
```

Modules and Packages

Modules are Python files with .py extension, which are imported using the **import** command.

```
#import module evolution
import evolution
```

Each package in Python is a directory which **MUST** contain a special file called __init__.py, and it can be imported the same way a module can be imported.

```
#import module evolution in package foo
import foo.evolution
```

Modules and Packages

It's possible to import just some functions of a module.

```
#import function urlopne from request
from urllib.request import urlopen
```

It's possible 're-import' an updated module.

```
import importlib
importlib.reload(name_module)
from name_module import name_function
```

4

Design Pattern Examples

Façade Pattern

```
class Roda:
    def run(self):
        print ("roda gira")
class Motor:
    def run(self):
        print ("acelera")
class Freio:
    def run(self):
        print ("stand by")
```

```
#Facade
class FacadeRunner:
    def __init__(self):
        self.roda = Roda()
        self.motor = Motor()
        self.freio = Freio()
    def runAll(self):
        self.roda.run()
        self.motor.run()
        self.freio.run()
#Client
testrunner = FacadeRunner()
testrunner.runAll()
```

```
>>> roda gira
>>> acelera
>>> stand by
```

Factory Pattern

```
class Book:
    def __init__(self):
        self.title = None
        self.option = None
    def getTitle(self):
        return self.title
    def getOption(self):
        return self.option
class New(Book):
    def __init__(self, title):
        print "Sell book:" + title
class Old(Book):
    def __init__(self, title):
        print "Buy book:" + title
```

Factory Pattern

```
class Factory:
    def getBook(self, title, option):
if option == 'S':
return New(title)
if option == 'B':
return Old(title)

>>>factory = Factory()
>>>book = factory.getBook("Design Patterns",
"S")
Sell book:Design Patterns
```

Proxy Pattern

The Proxy pattern is used when an object has to be protected from its clients, so proxy substitutes the real object by providing the same interface to the client.

```
class Proxy:
    def __init__( self, subject ):
        self.__subject = subject
    def __getattr__( self, name ):
        return getattr( self.__subject, name )
```

More on this at:
http://legacy.python.org/workshops/1997-10/proceedings/savikko.html

Proxy Pattern

```
class RGB:
    def __init__( self, red, green, blue ):
        self.__red = red
        self.__green = green
        self.__blue = blue
    def Red( self ):
        return self.__red
    def Green( self ):
        return self.__green
    def Blue( self ):
        return self.__blue
class NoBlueProxy( Proxy ):
    def Blue( self ):
        return 0
```

```
>>>rgb = RGB( 100, 192, 240 )
>>>print (rgb.Red())
100

>>>proxy = Proxy( rgb )
>>>print (proxy.Green())
192

>>>noblue = NoBlueProxy( rgb )
>>>print (noblue.Green())
192
```

5

Examples

Some codes in Python

Parsing a JSON from an API

- Like SCP(Smart-City platform) project
- Used to transfer data between different systems and patterns
- Different methods in Python2 and 3
- Essential modules: json and urllib

Simple model of use (from file)

```
>>> import json

>>> jsdata = open('file_name.json')
>>> js=json.load(jsdata)
>>> js
```

{'objects ': [{'name ': 'SheinaSpa Salon', 'locality': 'New York', ……., 'region': 'NY', 'categories': ['other ', 'beautysalon']}], 'meta': {'cache-expiry': 3600, 'limit ': 25}}

```
>>> type(js)
<class 'dict'>
```

Simple model of use

json.dumps - show data organized

```
>>> print(json.dumps(js, indent=4))
{
    "objects": [
        {
            "has_menu": true,
            "categories": [
                "other",
                "beauty salon"
            ],
            "postal_code": "10033",
            "website_url": "http://www.sheinaspa.com/",
            "name": "Sheina Spa Salon",
            "id": "b677b8da1505c9078763",
            "locality": "New York",
            "phone": "(212) 866-9100",
            "long": -73.936034,
            "lat": 40.849887
        },.....
```

Parsing JSON

```
def search(d, args):
   if isinstance(d, dict):
      for k, v in d.items():
         if k in args:
            print(k,' Value: ',v)
         if isinstance(v, dict) or isinstance(v, list):
            search(v, args)
   else:
      for o in d:
         if isinstance(o, dict) or isinstance(o, list):
            search(o, args)
```

```
>>> search(js, ['name '])
name  Value:  Sheina Spa Salon
name  Value:  www.aromatheraclean.com
```

Parsing JSON

```
>>> search(js, ['name','lat','long'])
lat  Value:  40.849887
long  Value:  -73.936034
name  Value:  Sheina Spa Salon
lat  Value:  40.80437
long  Value:  -73.935326
name  Value:  www.aromatheraclean.com
lat  Value:  40.75472
long  Value:  -73.981257
name  Value:  Harmony Skin Care
lat  Value:  40.713487
long  Value:  -74.00903
```

Simple model of use (from API)

```
>>> import json
>>> from urllib.request import urlopen

>>> url = 'https://xxxxxx'
>>> response = urlopen(url)
>>> str=response.read().decode('utf-8')
>>> json_obj = json.loads(str)

>>> type(json_obj)
<class 'dict'>
```

Thanks!

ANY QUESTIONS?

References

ECKEL, Bruce. Python 3 Patterns, Recipes and Idioms. 2016.

VERMA, Rahul; GIRIDHAR, Chetan. Design Patterns in Python. 2011.

https://docs.python.org/3/

http://www.programiz.com/python-programming

For the API exemples:
https://dev.locu.com/
https://locu.com/