Advanced Python Object Oriented and Functional Programming in Python for NYC Audience

Shivgan Joshi

Python Research for Extensive Business Solutions New York

S. Joshi

03/14/2019

Advanced Python Object Oriented and Functional Programming in Python for NYC Audience

Abstract: This article is about advanced topics in Python pertinent to functional and object oriented Python.
Author: Shivgan Joshi et. al.

My other online blogs::
http://learnpythondatasciencenyc.site/
http://bigdatascienceblockchainnyc.site/
http://ebscorp.us/

After reading this article you will be able to understand Python more deeply and launch your career from scripting to software development / web development / Quant in Python . If you know basic syntax of Python then this article will launch you to the next level.

Directions can you go if you learn OOP and Functional Python:

1. Web development: Django uses classes and functional style extensively.
2. Banks are converting their old code from proprietary softwares to the open source world so that they can harness the evolution in Big Data and Python.
3. Banks are building their own libraries and putting them in libraries to run their analytics which was earlier done by companies like MATLAB, SAS, etc. If you are a CFA, FRM or Quant developer this courses help you understand what you need to learn in Python.
4. Code made earlier can be made more optimal using functional aspects and then create classes and libraries of your earlier code. This ways you can organize your code and share it in a library.

This course introduces to OOPS and Functional aspects of Python.

1. Python allows us to use Object Oriented, Functional and scripting and hence it the language of data science, web, big data and software development.
2. Object oriented programming involves classes, inheritance, meta classes, encapsulation, overloading using classes.

Main Topics to be Discussed:

- Functional programming
  - Decorators & Higher Order Functions
  - Call backs
  - Late binding / Closures
  - Method Types in Python OOP: @classmethod, @staticmethod, and Instance Methods

Main Topics to be Discussed:

Introduction to Advanced Python

Object-Oriented Programming in Python
Exploring Python Features
Verifying Code and Unit Testing
Detecting Errors and Debugging Techniques
Implementing Python Design Patterns
Interfacing with REST Web Services and Clients
Measuring and Improving Application Performance
Installing and Distributing Modules
Concurrent Execution

- Functional programming

Idea about functional programming is to use Comprehensions, Generators, Recursion And Eliminate Loops and Eliminating Recursion and also use Lambda and higher level functions.

Name the functional approach that Python is taking.
Python provides the following:

map(aFunction, aSequence)
filter(aFunction, aSequence)
reduce(aFunction, aSequence)
lambda
list comprehension

- Decorators & Higher Order Functions

Decorators in Python are used to modify or inject code in functions or classes. Using decorators, you can wrap a class or function method call so that a piece of code can be executed before or after the execution of the original code. Decorators can be used to check for permissions, modify or track the arguments passed to a method, logging the calls to a specific method, etc.
Decorators provide a simple syntax for calling higher-order functions. By definition, a decorator is a function that takes another function and extends the behavior of the latter function without explicitly modifying it.

def our_decorator(func):

```
    def function_wrapper(x):
        print("Before calling " + func.__name__)
        func(x)
        print("After calling " + func.__name__)
    return function_wrapper

@our_decorator
def foo(x):
    print("Hi, foo has been called with " + str(x))

foo("Hi")
```

```
def hello_decorator(func):
    def inner1(*args, **kwargs):

        print("before Execution")

        # getting the returned value
        returned_value = func(*args, **kwargs)
        print("after Execution")

        # returning the value to the original frame
        return returned_value

    return inner1


# adding decorator to the function
@hello_decorator
def sum_two_numbers(a, b):
    print("Inside the function")
    return a + b

a, b = 1, 2

# getting the value through return of the function
print("Sum =", sum_two_numbers(a, b))
```

==========

The following decorator implemented as a class does the same "job":

```
class decorator2:
```

```
    def __init__(self, f):
        self.f = f

    def __call__(self):
        print("Decorating", self.f.__name__)
        self.f()

@decorator2
def foo():
    print("inside foo()")

foo()
```

https://realpython.com/primer-on-python-decorators/
https://www.hackerearth.com/practice/python/functional-programming/higher-order-functions-and-decorators/tutorial/

- Call backs

A callback can be informally described like this: function a calls function b, and wants to make b run a specific independent chunk of code at some point during b's execution.

python asynchronous callback

```
def callback1(a, b):
    print('Sum = {0}'.format(a+b))

def callback2(a):
    print('Square = {0}'.format(a**2))

def callback3():
    print('Hello, world!')

def main(callback=None, cargs=()):
    print('Calling callback.')
    if callback != None:
        callback(*cargs)
```

```
main(callback1, cargs=(1, 2))
main(callback2, cargs=(2,))
main(callback3)
```

```
import time
from concurrent.futures import ThreadPoolExecutor


def long_computation(duration):
    for x in range(0, duration):
        print(x)
        time.sleep(1)
    return duration * 2


print('Use polling')
with ThreadPoolExecutor(max_workers=1) as executor:
    future = executor.submit(long_computation, 5)
    while not future.done():
        print('waiting...')
        time.sleep(0.5)

    print(future.result())

print('Use callback')
executor = ThreadPoolExecutor(max_workers=1)
future = executor.submit(long_computation, 5)
future.add_done_callback(lambda f: print(f.result()))

print('waiting for callback')

executor.shutdown(False)  # non-blocking

print('shutdown invoked')
```

---

https://stackoverflow.com/questions/1239035/asynchronous-method-call-in-python
https://stackoverflow.com/questions/40843039/how-to-write-a-simple-callback-function

http://code.activestate.com/recipes/580787-implementing-function-based-callbacks-in-python/

Late binding / Closures

```
# Example of using callbacks with Python
#
# To run this code
#   1. Copy the content into a file called `callback.py`
#   2. Open Terminal and type: `python /path/to/callback.py`
#   3. Enter

def add(numbers, callback):
    results = []
    for i in numbers:
        results.append(callback(i))
    return results

def add2(number):
    return number + 2

def mul2(number):
    return number * 2

print (add([1,2,3,4], add2)) #=> [3, 4, 5, 6]
print (add([1,2,3,4], mul2)) #=> [2, 4, 6, 8]
```

In the first lines is that functions contains pointers to three different functions.
https://stackoverflow.com/questions/36463498/late-binding-python-closures

Method Types in Python OOP: @classmethod, @staticmethod, and Instance Methods

What's the difference between @classmethod, @staticmethod, and "plain/regular" instance methods in Python?
Regular (instance) methods need a class instance and can access the instance through `self`. They can read and modify an objects state freely. Class methods, marked with the @classmethod decorator, don't need a class instance. They can't access the instance (self) but they have access to the class itself via `cls`. Static methods, marked with the @staticmethod decorator, don't have access to `cls` or `self`. They work like regular functions but belong to the class's namespace.

https://www.youtube.com/watch?v=PNpt7cFjGsM
https://www.bogotobogo.com/python/python_differences_between_static_method_and_class_method_instance_method.php

Pass by reference / pass by value

Everything is pass by reference.

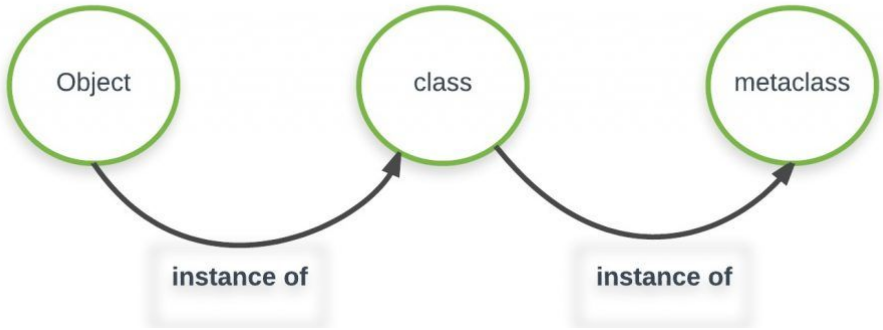
https://stackoverflow.com/questions/4689984/implementing-a-callback-in-python-passing-a-callable-reference-to-the-current

- # Object Oriented Programming

- ## Meta Programming & Meta Classes

Metaprogramming in Python relies on a special new type of class that is called the metaclass. This type of class, in short, holds the instructions about the behind-the-scenes code generation that you want to take place when another piece of code is being executed.

A metaclass is the class of a class. Like a class defines how an instance of the class behaves, a metaclass defines how a class behaves. A class is an instance of a metaclass.
A metaclass is most commonly used as a class-factory. Like you create an instance of the class by calling the class, Python creates a new class (when it executes the 'class' statement) by calling the metaclass



Ref:
Advanced Python 3 Programming Techniques By Mark Summerfield
https://stackoverflow.com/questions/100003/what-are-metaclasses-in-python?rq=1

https://www.geeksforgeeks.org/metaprogramming-metaclasses-python/
https://stackoverflow.com/questions/100003/what-are-metaclasses-in-python
https://medium.com/@guoxing/brief-intro-to-metaprogramming-with-python-a278fc104b3b

- Abstract Class / ABC Class

Abstract classes: Force a class to implement methods. Abstract classes can contain abstract methods: methods without an implementation. Objects cannot be created from an abstract class. A subclass can implement an abstract class.
Abstract classes are classes that contain one or more abstract methods. An abstract method is a method that is declared, but contains no implementation. Abstract classes may not be instantiated, and require subclasses to provide implementations for the abstract methods. Subclasses of an abstract class in Python are not required to implement abstract methods of the parent class.

https://www.youtube.com/watch?v=PDMe3wgAsWg
https://www.python-course.eu/python3_abstract_classes.php
http://blog.thedigitalcatonline.com/blog/2016/04/03/abstract-base-classes-in-python/

- __new__ vs __init__

Technically with __new__ the first argument is the class, while with __init__ the first argument is the instance. However, it is still true that they must both be able to accept the same arguments, since, except for that first argument, the arguments passed to __init__ are the same as those passed to __new__.) The `__new__` method is passed the class as first argument, while the `__init__` method is passed the result of the `__new__` method; the freshly created instance.

Types of Inheritance

Single Inheritance – where a derived class acquires the members of a single super class.
Multi-level inheritance – a derived class d1 in inherited from base class base1, and d2 are inherited from base2.
Hierarchical inheritance – from one base class you can inherit any number of child classes
Multiple inheritance – a derived class is inherited from more than one base class.

List out the inheritance styles in Django.
In Django, there is three possible inheritance styles:

Abstract Base Classes: This style is used when you only wants parent's class to hold information that you don't want to type out for each child model.

Multi-table Inheritance: This style is used If you are sub-classing an existing model and need each model to have its own database table.
Proxy models: You can use this model, If you only want to modify the Python level behavior of the model, without changing the model's fields.

Super function in Python

In Python, super() built-in has two major use cases:
Allows us to avoid using base class explicitly
Working with Multiple Inheritance

Singleton Method

This pattern restricts the instantiation of a class to one object. It is a type of creational pattern and involves only one class to create methods and specified objects.

```
class Singleton:
  __instance = None
  @staticmethod
  def getInstance():
    """ Static access method. """
    if Singleton.__instance == None:
      Singleton()
    return Singleton.__instance
  def __init__(self):
    """ Virtually private constructor. """
    if Singleton.__instance != None:
      raise Exception("This class is a singleton!")
    else:
      Singleton.__instance = self
s = Singleton()
print (s)

s = Singleton.getInstance()
print (s)

s = Singleton.getInstance()
print (s)
```

https://www.youtube.com/watch?v=qzCHtYoqh_I

Understanding Class and Instance Variables

https://www.digitalocean.com/community/tutorials/understanding-class-and-instance-variables-in-python-3

Static, class or abstract methods in Python

Static methods are a special case of methods. Sometimes, you'll write code that belongs to a class, but that doesn't use the object itself at all.

Class methods are methods that are not bound to an object, but to… a class!
An abstract method is a method defined in a base class, but that may not provide any implementation.

https://julien.danjou.info/guide-python-static-class-abstract-methods/

Design Pattern

https://www.tutorialspoint.com/python_design_patterns/python_design_patterns_factory.htm

Metaclass

A metaclass is the class of a class. Like a class defines how an instance of the class behaves, a metaclass defines how a class behaves. A class is an instance of a metaclass.
https://stackoverflow.com/questions/100003/what-are-metaclasses-in-python

- # Generators, Iterators, Decorators, and Context Managers

A decorator is a function that takes a function as an argument and returns a function as a return value.
an_iterator.__iter__()
itertools is a collection of utilities that make it easy to build an iterator that iterates over sequences in various common ways.

Generators give you the iterator immediately: no access to the underlying data ... if it even exists.
Context Managers: You can encapsulate the setup, error handling and teardown of resources in a few simple steps. The key is to use the with statement.

- Unit Testing Python

Advanced unit-testing. Mocks, patches, possibly a more advanced library like pytest
Python standard library is called unittest. The principles of unittest are easily portable to other frameworks, like:

1. unittest
2. nose or nose2
3. pytest

pytest supports execution of unittest test cases. The real advantage of pytest comes by writing pytest test cases. pytest test cases are a series of functions in a Python file starting with the name test_.

- Multithreading in Python? Why is it a bad idea?

Python doesn't allow multi-threading in the truest sense of the word. It has a multi-threading package but if you want to multi-thread to speed your code up, then it's usually not a good idea to use it. Python has a construct called the Global Interpreter Lock (GIL). The GIL makes sure that only one of your 'threads' can execute at any one time. A thread acquires the GIL, does a little work, then passes the GIL onto the next thread. This happens very quickly so to the human eye it may seem like your threads are executing in parallel, but they are really just taking turns using the same CPU core. All this GIL passing adds overhead to execution. This means that if you want to make your code run faster then using the threading package often isn't a good idea.

What is multi-threading? What is GIL(Global interpreter lock) issue ?

In CPython, the global interpreter lock, or GIL, is a mutex that protects access to Python objects, preventing multiple threads from executing Python bytecodes at once. This lock is necessary mainly because CPython's memory management is not thread-safe. (However, since the GIL exists, other features have grown to depend on the guarantees that it enforces.)

```
"""Test Callback Function"""
import multiprocessing as mp

def count(countvar):
```

```
    """This function will just count to 100"""
    print('Incoming Variable is equal to ' + str(countvar))
    countvar = 0
    while countvar < 1000000:
        countvar = countvar + 1

    response = "Count is set to " + str(countvar)
    return response

def callback(result):
    """This will print the result calleded via the callback."""

    if result is not None:
        print(str(result[0]) + ' Callback Succeeded')
    else:
        print("Callback Failure")

#
# Create pool
#
PROCESSES = 4
print('Creating pool with %d processes\n' % PROCESSES)
POOL = mp.Pool(PROCESSES)
print('POOL = %s' % POOL)
print()

TEST = None

MP_CALLBACK = None
RESULT = None
# RESULT = POOL.apply_async(count, (TEST, ), callback=MP_CALLBACK)
RESULT = POOL.map_async(count, (TEST, ), callback=callback)
POOL.close()
POOL.join()
```

Synchronization

https://docs.python.org/3/library/multiprocessing.html#module-multiprocessing
https://www.tutorialspoint.com/python_online_training/python_synchronizing_threads.asp
https://wiki.python.org/moin/GlobalInterpreterLock

- Why a list comprehension is faster than a for loop (which really is to say understand how bytecode is generated, at high level)

This is a topic of functional programming.

https://learning.oreilly.com/library/view/functional-programming-in/9781492048633/
https://stackoverflow.com/questions/22108488/are-list-comprehensions-and-functional-functions-faster-than-for-loops

- Overloading

Overloading, in the context of programming, refers to the ability of a function or an operator to behave in different ways depending on the parameters that are passed to the function, or the operands that the operator acts on.
Overloading Built-in Functions & Overloading User-Defined Functions
https://stackabuse.com/overloading-functions-and-operators-in-python/

Monkey Patching

In Python, the term monkey patch refers to dynamic (or run-time) modifications of a class or module. In Python, we can actually change the behavior of code at run-time.

```
# monk.py
class A:
    def func(self):
        print "func() is being called"


import monk
def monkey_f(self):
    print "monkey_f() is being called"

# replacing address of "func" with "monkey_f"
monk.A.func = monkey_f
obj = monk.A()

# calling function "func" whose address got replaced
```

```
# with function "monkey_f()"
obj.func()
```

https://riptutorial.com/python/example/9909/monkey-patching
https://www.geeksforgeeks.org/monkey-patching-in-python-dynamic-behavior/

Garbage Collection in Python

https://www.geeksforgeeks.org/garbage-collection-python/
https://rushter.com/blog/python-garbage-collector/

PEP 8 -- Style Guide for Python Code

https://stackoverflow.com/questions/356161/python-coding-standards-best-practices
https://www.python.org/dev/peps/pep-0008/

With Statement

The 'with' statement clarifies code that previously would use try...finally blocks to ensure that clean-up code is executed.

```
class controlled_execution:
        def __enter__(self):
            set things up
            return thing
        def __exit__(self, type, value, traceback):
            tear things down

    with controlled_execution() as thing:
         some code
```

The contextlib module
The new contextlib module provides some functions and a decorator that are useful for writing objects for use with the 'with' statement.

The decorator is called contextmanager, and lets you write a single generator function instead of defining a new class. The generator should yield exactly one value. The code up to the yield will be executed as the __enter__() method, and the value yielded will be the method's return value that will get bound to the variable in the 'with' statement's as clause, if any. The code after the yield will be executed in the __exit__() method. Any exception raised in the block will be raised by the yield statement.

https://docs.python.org/2.5/whatsnew/pep-343.html
https://stackoverflow.com/questions/1369526/what-is-the-python-keyword-with-used-for
http://effbot.org/zone/python-with-statement.htm

Other concepts

What is the difference between deep and shallow copy?

Ans: Shallow copy is used when a new instance type gets created and it keeps the values that are copied in the new instance. Shallow copy is used to copy the reference pointers just like it copies the values. These references point to the original objects and the changes made in any member of the class will also affect the original copy of it. Shallow copy allows faster execution of the program and it depends on the size of the data that is used.
Deep copy is used to store the values that are already copied. Deep copy doesn’t copy the reference pointers to the objects. It makes the reference to an object and the new object that is pointed by some other object gets stored. The changes made in the original copy won’t affect any other copy that uses the object. Deep copy makes execution of the program slower due to making certain copies for each object that is been called.

Terms for job interviews

- Different type of Inheritance in Python. Mixing static, class and abstract methods for inheritance.
- Python object model (metaclasses, slots, and descriptors, as well as how inheritance works), __prepare__ and __init_subclass__.
- Generators, Iterators, and Context Managers.
- Dunders
- Standard library: math, itertools, functools, random, collections, logging, sys, os, and threading/multiprocessing/asyncio
- Global optimization for spark for finding difference optimization methods

https://www.quora.com/How-can-I-prepare-for-a-Python-interview-for-experienced-developers

https://luminousmen.com/post/python-interview-questions-senior

Django

Django View Decorators

View decorators can be used to restrict access to certain views. Django come with some built-in decorators, like login_required, require_POST or has_permission. They are really useful, but sometimes you might need to restrict the access in a different level of granularity, for example only letting the user who created an entry of the model to edit or delete it.

https://simpleisbetterthancomplex.com/2015/12/07/working-with-django-view-decorators.html

Python Notebook

https://colab.research.google.com/drive/1a66aG95PM2UZ1WZFTpy80mf_29S6WE67#scrollTo=Di7_vk3t_aVD

Reference