1.Compare Java & Python

|  |  |  |
| --- | --- | --- |
| **Criteria** | **Java** | **Python** |
| Ease of use | Good | Very Good |
| Speed of coding | Average | Excellent |
| Data types | Static typed | Dynamically typed |
| Data Science & machine learning applications | Average | Very Good |

**Explain how Python does Compile-time and Run-time code checking?**

Python performs some amount of compile-time checking, but most of the checks such as type, name, etc are postponed until code execution. Consequently, if the Python code references a user -defined function that does not exist, the code will compile successfully. In fact, the code will fail with an exception only when the code execution path references the function which does not exists.

**Explain Python’s zip() function.?**

zip() function- it will take multiple lists say list1, list2, etc and transform them into a single list of tuples by taking the corresponding elements of the lists that are passed as parameters. Eg:

list1 = ['A',

'B','C'] and list2 = [10,20,30].

zip(list1, list2) # results in a list of tuples say [('A',10),('B',20),('C',30)]

**As Everything in Python is an Object, Explain the characteristics of Python’s Objects.**

As Python’s Objects are instances of classes, they are created at the time of instantiation. Eg: object-name = class-name(arguments)  
• one or more variables can reference the same object in Python  
• Every object holds unique id and it can be obtained by using id() method. Eg: id(obj-name) will return unique id of the given object.  
every object can be either mutable or immutable based on the type of data they hold.  
• Whenever an object is not being used in the code, it gets destroyed automatically garbage collected or destroyed  
• contents of objects can be converted into string representation using a method

**Explain how to overload constructors or methods in Python.**

Python’s constructor – \_init\_\_ () is a first method of a class. Whenever we try to instantiate a object \_\_init\_\_() is automatically invoked by python to initialize members of an object.

**What is a Python module?**

A module is a Python script that generally contains import statements, functions, classes and variable definitions, and Python runnable code and it “lives” file with a ‘.py’ extension. zip files and DLL files can also be modules.Inside the module, you can refer to the module name as a string that is stored in the global variable name .  
A module can be imported by other modules in one of the two ways. They are  
1. import  
2. from module-name import or

**What is JSON? How would convert JSON data into Python data?**

JSON – stands for JavaScript Object Notation. It is a popular data format for storing data in NoSQL  
databases. Generally JSON is built on 2 structures.  
1. A collection of <name, value> pairs.  
2. An ordered list of values.  
As Python supports JSON parsers, JSON-based data is actually represented as a dictionary in Python. You can convert json data into python using load() of json module.

**How instance variables are different from class variables?**

**Instance variables:** are the variables in an object that have values that are local to that object. Two objects of the same class maintain distinct values for their variables. These variables are accessed with “object-name.instancevariable-name”.  
**class variables:** these are the variables of class. All the objects of the same class will share value of “Class variables. They are accessed with their class name alone as “class- name.classvariable-name”. If you change the value of a class variable in one object, its new value is visible among all other objects of the same class. In the Java world, a variable that is declared as static is a class variable.

**Does Python supports interfaces like in Java? Discuss.**

Python does not provide interfaces like in Java. Abstract Base Class (ABC) and its feature are provided by the Python’s “abc” module. Abstract Base Class is a mechanism for specifying what methods must be implemented by its implementation subclasses. The use of ABC’c provides a sort of “understanding” about methods and their expected behaviour. This module was made available from Python 2.7 version onwards.

**How would you define a protected member in a Python class?**

All the members of a class in Python are public by default. You don’t need to define an access specifier for members of class. By adding ‘\_’ as a prefix to the member of a class, by convetion you are telling others please don’t this object, if you are not a subclass the respective class.  
Eg: class Person:  
empid = None  
\_salary = None #salary is a protected member & it can accessible by the subclasses of Person  
….

**Name few methods that are used to implement Functionally Oriented Programming in Python?**

Python supports methods (called iterators in Python3), such as filter(), map(), and reduce(), that are very useful when you need to iterate over the items in a list, create a dictionary, or extract a subset of a list.  
filter() – enables you to extract a subset of values based on conditional logic.  
map() – it is a built-in function that applies the function to each item in an iterable.  
reduce() – repeatedly performs a pair-wise reduction on a sequence until a single value is computed.

**Python and multi-threading. Is it a good idea? List some ways to get some Python code to run in a parallel way.**

Python doesn't allow multi-threading in the truest sense of the word. It has a [multi-threading package](https://docs.python.org/2/library/threading.html) 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.

There are reasons to use Python's threading package. If you want to run some things simultaneously, and efficiency is not a concern, then it's totally fine and convenient. Or if you are running code that needs to wait for something (like some IO) then it could make a lot of sense. But the threading library wont let you use extra CPU cores.

Multi-threading can be outsourced to the operating system (by doing multi-processing), some external application that calls your Python code (eg, Spark or Hadoop), or some code that your Python code calls (eg: you could have your Python code call a C function that does the expensive multi-threaded stuff).

Why this is important

Because the GIL is an A-hole. Lots of people spend a lot of time trying to find bottlenecks in their fancy Python multi-threaded code before they learn what the GIL is.

#Multiprocessing

**import** time  
**import** multiprocessing  
  
**def** calc\_square(numbers):  
 **for** n **in** numbers:  
 print(**'square '** + str(n\*n))  
  
**def** calc\_cube(numbers):  
 **for** n **in** numbers:  
 print(**'cube '** + str(n\*n\*n))  
  
**if** \_\_name\_\_ == **"\_\_main\_\_"**:  
 arr = [2,3,8]  
 p1 = multiprocessing.Process(target=calc\_square, args=(arr,))  
 p2 = multiprocessing.Process(target=calc\_cube, args=(arr,))  
  
 p1.start()  
 p2.start()  
  
 p1.join()  
 p2.join()  
  
 print(**"Done!"**)

OUTPUT:

cube 8

cube 27

cube 512

square 4

square 9

square 64

Done!

Treading:

import time

import threading

def calc\_square(numbers):

print("calculate square numbers")

for n in numbers:

time.sleep(1)

print('square:',n\*n)

def calc\_cube(numbers):

print("calculate cube of numbers")

for n in numbers:

time.sleep(1)

print('cube:',n\*n\*n)

arr = [2,3,8,9]

t = time.time()

t1= threading.Thread(target=calc\_square, args=(arr,))

t2= threading.Thread(target=calc\_cube, args=(arr,))

t1.start()

t2.start()

t1.join()

t2.join()

print("done in : ",time.time()-t)

print("Hah... I am done with all my work now!")

OUTPUT:

calculate square numbers

calculate cube of numbers

square: 4

cube: 8

square: 9

cube: 27

square: 64

cube: 512

square: 81

cube: 729

done in : 4.003229141235352

Hah... I am done with all my work now!

**What is monkey patching and is it ever a good idea?**

A MonkeyPatch is a piece of Python code which extends or modifies other code at runtime (typically at startup).

from SomeOtherProduct.SomeModule import SomeClass

def speak(self):

return "ook ook eee eee eee!"

SomeClass.speak = speak

**What does this stuff mean: \*args, \*\*kwargs? And why would we use it?**

Use \*args when we aren't sure how many arguments are going to be passed to a function, or if we want to pass a stored list or tuple of arguments to a function. \*\*kwargs is used when we dont know how many keyword arguments will be passed to a function, or it can be used to pass the values of a dictionary as keyword arguments. The identifiers args and kwargs are a convention, you could also use \*bob and \*\*billy but that would not be wise.

Here is a little illustration:

def f(\*args,\*\*kwargs): print(args, kwargs)

l = [1,2,3]

t = (4,5,6)

d = {'a':7,'b':8,'c':9}

f()

f(1,2,3) # (1, 2, 3) {}

f(1,2,3,"groovy") # (1, 2, 3, 'groovy') {}

f(a=1,b=2,c=3) # () {'a': 1, 'c': 3, 'b': 2}

f(a=1,b=2,c=3,zzz="hi") # () {'a': 1, 'c': 3, 'b': 2, 'zzz': 'hi'}

f(1,2,3,a=1,b=2,c=3) # (1, 2, 3) {'a': 1, 'c': 3, 'b': 2}

f(\*l,\*\*d) # (1, 2, 3) {'a': 7, 'c': 9, 'b': 8}

f(\*t,\*\*d) # (4, 5, 6) {'a': 7, 'c': 9, 'b': 8}

f(1,2,\*t) # (1, 2, 4, 5, 6) {}

f(q="winning",\*\*d) # () {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}

f(1,2,\*t,q="winning",\*\*d) # (1, 2, 4, 5, 6) {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}

def f2(arg1,arg2,\*args,\*\*kwargs): print(arg1,arg2, args, kwargs)

f2(1,2,3) # 1 2 (3,) {}

f2(1,2,3,"groovy") # 1 2 (3, 'groovy') {}

f2(arg1=1,arg2=2,c=3) # 1 2 () {'c': 3}

f2(arg1=1,arg2=2,c=3,zzz="hi") # 1 2 () {'c': 3, 'zzz': 'hi'}

f2(1,2,3,a=1,b=2,c=3) # 1 2 (3,) {'a': 1, 'c': 3, 'b': 2}

f2(\*l,\*\*d) # 1 2 (3,) {'a': 7, 'c': 9, 'b': 8}

f2(\*t,\*\*d) # 4 5 (6,) {'a': 7, 'c': 9, 'b': 8}

f2(1,2,\*t) # 1 2 (4, 5, 6) {}

f2(1,1,q="winning",\*\*d) # 1 1 () {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}

f2(1,2,\*t,q="winning",\*\*d) # 1 2 (4, 5, 6) {'a': 7, 'q': 'winning', 'c': 9, 'b': 8}

**What do these mean to you: @classmethod, @staticmethod, @property?**

These are decorators. A decorator is a special kind of function that either takes a function and returns a function, or takes a class and returns a class. The @ symbol is just syntactic sugar that allows you to decorate something in a way that's easy to read.

class MyClass(object):

def \_\_init\_\_(self):

self.\_some\_property = "properties are nice"

self.\_some\_other\_property = "VERY nice"

def normal\_method(\*args,\*\*kwargs):

print("calling normal\_method({0},{1})".format(args,kwargs))

@classmethod

def class\_method(\*args,\*\*kwargs):

print("calling class\_method({0},{1})".format(args,kwargs))

@staticmethod

def static\_method(\*args,\*\*kwargs):

print("calling static\_method({0},{1})".format(args,kwargs))

@property

def some\_property(self,\*args,\*\*kwargs):

print("calling some\_property getter({0},{1},{2})".format(self,args,kwargs))

return self.\_some\_property

@some\_property.setter

def some\_property(self,\*args,\*\*kwargs):

print("calling some\_property setter({0},{1},{2})".format(self,args,kwargs))

self.\_some\_property = args[0]

@property

def some\_other\_property(self,\*args,\*\*kwargs):

print("calling some\_other\_property getter({0},{1},{2})".format(self,args,kwargs))

return self.\_some\_other\_property

o = MyClass()

# undecorated methods work like normal, they get the current instance (self) as the first argument

o.normal\_method

# <bound method MyClass.normal\_method of <\_\_main\_\_.MyClass instance at 0x7fdd2537ea28>>

o.normal\_method()

# normal\_method((<\_\_main\_\_.MyClass instance at 0x7fdd2537ea28>,),{})

o.normal\_method(1,2,x=3,y=4)

# normal\_method((<\_\_main\_\_.MyClass instance at 0x7fdd2537ea28>, 1, 2),{'y': 4, 'x': 3})

# class methods always get the class as the first argument

o.class\_method

# <bound method classobj.class\_method of <class \_\_main\_\_.MyClass at 0x7fdd2536a390>>

o.class\_method()

# class\_method((<class \_\_main\_\_.MyClass at 0x7fdd2536a390>,),{})

o.class\_method(1,2,x=3,y=4)

# class\_method((<class \_\_main\_\_.MyClass at 0x7fdd2536a390>, 1, 2),{'y': 4, 'x': 3})

# static methods have no arguments except the ones you pass in when you call them

o.static\_method

# <function static\_method at 0x7fdd25375848>

o.static\_method()

# static\_method((),{})

o.static\_method(1,2,x=3,y=4)

# static\_method((1, 2),{'y': 4, 'x': 3})

# properties are a way of implementing getters and setters. It's an error to explicitly call them

# "read only" attributes can be specified by creating a getter without a setter (as in some\_other\_property)

o.some\_property

# calling some\_property getter(<\_\_main\_\_.MyClass instance at 0x7fb2b70877e8>,(),{})

# 'properties are nice'

o.some\_property()

# calling some\_property getter(<\_\_main\_\_.MyClass instance at 0x7fb2b70877e8>,(),{})

# Traceback (most recent call last):

# File "<stdin>", line 1, in <module>

# TypeError: 'str' object is not callable

o.some\_other\_property

# calling some\_other\_property getter(<\_\_main\_\_.MyClass instance at 0x7fb2b70877e8>,(),{})

# 'VERY nice'

# o.some\_other\_property()

# calling some\_other\_property getter(<\_\_main\_\_.MyClass instance at 0x7fb2b70877e8>,(),{})

# Traceback (most recent call last):

# File "<stdin>", line 1, in <module>

# TypeError: 'str' object is not callable

o.some\_property = "groovy"

# calling some\_property setter(<\_\_main\_\_.MyClass object at 0x7fb2b7077890>,('groovy',),{})

o.some\_property

# calling some\_property getter(<\_\_main\_\_.MyClass object at 0x7fb2b7077890>,(),{})

# 'groovy'

o.some\_other\_property = "very groovy"

# Traceback (most recent call last):

# File "<stdin>", line 1, in <module>

# AttributeError: can't set attribute

o.some\_other\_property

# calling some\_other\_property getter(<\_\_main\_\_.MyClass object at 0x7fb2b7077890>,(),{})

# 'VERY nice'

**@property**

#using method as attribute, e.g direclty attribute is called even it is a method

class Employee:

def \_\_init\_\_(self, first, last):

self.first = first

self.last = last

@property

def email(self):

return '{}.{}@email.com'.format(self.first, self.last)

@property

def fullname(self):

return '{} {}'.format(self.first, self.last)

emp\_1 = Employee('John', 'Smith')

print(emp\_1.first)

print(emp\_1.email)#look there is no (), direclty attribute is called even it is a method

print(emp\_1.fullname)

OUTPUT:

John

John.Smith@email.com

John Smith

**What is pickling and unpickling?**

Pickle module accepts any Python object and converts it into a string representation and dumps it into a file by using dump function, this process is called pickling.  While the process of retrieving original Python objects from the stored string representation is called unpickling.

Pickle

# Save a dictionary into a pickle file.

import pickle

favorite\_color = { "lion": "yellow", "kitty": "red" }

pickle.dump( favorite\_color, open( "save.p", "wb" ) )

Unpickle

# Load the dictionary back from the pickle file.

import pickle

favorite\_color = pickle.load( open( "save.p", "rb" ) )

# favorite\_color is now { "lion": "yellow", "kitty": "red" }

**What are the built-in type does python provides?**

Mutable built-in types

* List
* Sets
* Dictionaries

Immutable built-in types

* Strings
* Tuples
* Numbers

## **What is a namespace?**

To simply put it, namespace is a collection of names.

In Python, you can imagine a namespace as a mapping of every name, you have defined, to corresponding objects.

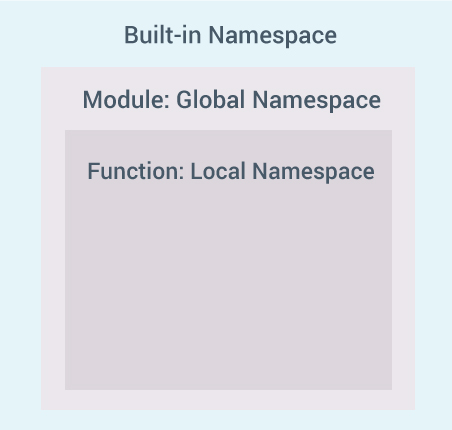
Different namespaces can co-exist at a given time but are completely isolated.

A namespace containing all the built-in names is created when we start the Python interpreter and exists as long we don't exit.

This is the reason that built-in functions like id(), print() etc. are always available to us from any part of the program. Each [module](https://www.programiz.com/python-programming/modules) creates its own global namespace.

These different namespaces are isolated. Hence, the same name that may exist in different modules do not collide.

Modules can have various functions and classes. A local namespace is created when a function is called, which has all the names defined in it. Similar, is the case with class. Following diagram may help to clarify this concept.



**shallow copy and deep copy in python**

A shallow copy constructs a new compound object and then (to the extent possible) inserts references into it to the objects found in the original.

In case of shallow copy, a reference of object is copied in other object. It means that **any changes** made to a copy of object **do reflect** in the original object.  
In python, this is implemented using “**copy()**” function.

A deep copy constructs a new compound object and then, recursively, inserts copies into it of the objects found in the original.

In case of deep copy, a copy of object is copied in other object. It means that **any changes** made to a copy of object **do not reflect** in the original object.  
In python, this is implemented using “**deepcopy()**” function.

Here's a little demonstration:

In case of shallow copy, a reference of object is copied in other object. It means that **any changes** made to a copy of object **do reflect** in the original object.

import copy

a = [1, 2, 3]

b = [4, 5, 6]

c = [a, b]

Using normal assignment operatings to copy:

d = c

print id(c) == id(d) # True - d is the same object as c

print id(c[0]) == id(d[0]) # True - d[0] is the same object as c[0]

Using a shallow copy:

d = copy.copy(c)

print id(c) == id(d) # False - d is now a new object

print id(c[0]) == id(d[0]) # True - d[0] is the same object as c[0]

Using a deep copy:

d = copy.deepcopy(c)

print id(c) == id(d) # False - d is now a new object

print id(c[0]) == id(d[0]) # False - d[0] is now a new object

**DEEP COPY**

In case of deep copy, a copy of object is copied in other object. It means that **any changes** made to a copy of object **do not reflect** in the original object.

Another example:

# importing "copy" for copy operations

import copy

# initializing list 1

li1 = [1, 2, [3,5], 4]

# using deepcopy to deep copy

li2 = copy.deepcopy(li1)

# original elements of list

print ("The original elements before deep copying")

for i in range(0,len(li1)):

    print (li1[i],end=" ")

print("\r")

# adding and element to new list

li2[2][0] = 7

# Change is reflected in l2

print ("The new list of elements after deep copying ")

for i in range(0,len( li1)):

    print (li2[i],end=" ")

print("\r")

# Change is NOT reflected in original list

# as it is a deep copy

print ("The original elements after deep copying")

for i in range(0,len( li1)):

    print (li1[i],end=" ")

Output:

The original elements before deep copying

1 2 [3, 5] 4

The new list of elements after deep copying

1 2 [7, 5] 4

The original elements after deep copying

1 2 [3, 5] 4

**COPY**

# importing "copy" for copy operations

# importing "copy" for copy operations

import copy

# initializing list 1

li1 = [1, 2, [3,5], 4]

# using copy to shallow copy

li2 = copy.copy(li1)

# original elements of list

print ("The original elements before shallow copying")

for i in range(0,len(li1)):

print (li1[i],end=" ")

print("\r")

# adding and element to new list

li2[2][0] = 7

# checking if change is reflected

print ("The original elements after shallow copying")

for i in range(0,len( li1)):

print (li1[i],end=" ")

Output:

The original elements before shallow copying

1 2 [3, 5] 4

The original elements after shallow copying

1 2 [7, 5] 4

### How memory is managed in Python?

Memory is managed in Python in following way:

* Memory is managed in Python by private heap space. All Python objects and data structures are located in a private heap. The programmer does not have an access to this private heap and interpreter takes care of this Python private heap.
* Python memory manager is responsible for allocating Python heap space for Python objects.
* Python also have an inbuilt garbage collector, which recycle all the unused memory and frees the memory and makes it available to the heap space.

**OOPS:**

**Polymorphism**

there are two types of polymorphism: compile time polymorphism (overloading) and runtime polymorphism (overriding).

Mehtod overriding: Overriding occurs when a class method has the same name and signature as a method in parent class. When you override methods, JVM determines the proper methods to call at the program’s run time, not at the compile time.

Overloading: Overloading is determined at the compile time. It occurs when several methods have same names with:

Different method signature and different number or type of parameters.

Same method signature but different number of parameters.

Same method signature and same number of parameters but of different type

class BookDetails {

String title;

setBook(String title){}

}

class ScienceBook extends BookDetails {

setBook(String title){} //overriding

setBook(String title, String publisher,float price){} //overloading

}

**Inheritance**

Inheritance allows a Child class to inherit properties from its parent class. In Java this is achieved by using extends keyword. Only properties with access modifier public and protected can be accessed in child class.

**class** Instrument(object):  
 **def** \_\_init\_\_(self, name):  
 self.name = name  
 **def** has\_strings(self):  
 **return True  
  
class** PercussionInstrument(Instrument):  
 **def** has\_strings(self):  
 **return False**guitar = Instrument(**'guitar'**)  
drums = PercussionInstrument(**'drums'**)  
  
print(**'Guitar has strings: {0}'**.format(guitar.has\_strings()))  
print(**'Guitar name: {0}'**.format(guitar.name))  
print(**'Drums have strings: {0}'**.format(drums.has\_strings()))  
print(**'Drums name: {0}'**.format(drums.name)

Multi level, multi inheritance

class A(object):

def go(self):

print("go A go!")

def stop(self):

print("stop A stop!")

def pause(self):

raise Exception("Not Implemented")

class B(A):

def go(self):

super(B, self).go()

print("go B go!")

class C(A):

def go(self):

super(C, self).go()

print("go C go!")

def stop(self):

super(C, self).stop()

print("stop C stop!")

class D(B,C):

def go(self):

super(D, self).go()

print("go D go!")

def stop(self):

super(D, self).stop()

print("stop D stop!")

def pause(self):

print("wait D wait!")

class E(B,C): pass

a = A()

b = B()

c = C()

d = D()

e = E()

# specify output from here onwards

a.go()

b.go()

c.go()

d.go()

e.go()

a.stop()

b.stop()

c.stop()

d.stop()

e.stop()

a.pause()

b.pause()

c.pause()

d.pause()

e.pause()

---- OUTPUT

a.go()

# go A go!

b.go()

# go A go!

# go B go!

c.go()

# go A go!

# go C go!

d.go()

# go A go!

# go C go!

# go B go!

# go D go!

e.go()

# go A go!

# go C go!

# go B go!

a.stop()

# stop A stop!

b.stop()

# stop A stop!

c.stop()

# stop A stop!

# stop C stop!

d.stop()

# stop A stop!

# stop C stop!

# stop D stop!

e.stop()

# stop A stop!

a.pause()

# ... Exception: Not Implemented

b.pause()

# ... Exception: Not Implemented

c.pause()

# ... Exception: Not Implemented

d.pause()

# wait D wait!

e.pause()

# ...Exception: Not Implemented

**abstraction**

Abstraction is a way of converting real world objects in terms of class. Its a concept of defining an idea in terms of classes or interface. For example creating a class Vehicle and injecting properties into it. E.g

public class Vehicle {

public String colour;

public String model;

}

**Encapsulation**

The encapsulation is achieved by combining the methods and attribute into a class. The class acts like a container encapsulating the properties. The users are exposed mainly public methods.The idea behind is to hide how thinigs work and just exposing the requests a user can do.

A single underscore indicates to the user of a class that an attribute should be considered private to the class, and should not be accessed directly.

A double underscore indicates the same, however, Python will mangle the attribute name somewhat to attempt to hide it.

class C(object):

def \_\_init\_\_(self):

self.a = 123 # OK to access directly

self.\_a = 123 # should be considered private

self.\_\_a = 123 # considered private, name mangled

>>> c = C()

>>> c.a

123

>>> c.\_a

123

>>> c.\_\_a

Traceback (most recent call last):

File "<stdin>", line 1, in <module>

AttributeError: 'C' object has no attribute '\_\_a'

>>> c.\_C\_\_a

123

Wrapping data and methods within classes in combination with implementation hiding (through access control) is often called encapsulation. The result is a data type with characteristics and behaviors. Encapsulation essentially has both i.e. information hiding and implementation hiding.

**Emulation**

Python provides many special methods on classes that can be used to emulate other types, such as functions, iterators, containers and more.

Functions. In order to emulate a function object, a class must define the method \_\_call\_\_(). If the call operator () is used on an instance of the class, this method will be called behind the scenes.

class Adder(object):

def \_\_init\_\_(self, extra):

self.extra = extra

def \_\_call\_\_(self, base):

return self.extra + base

add2 = Adder(2)

add2(3)

5

add5 = Adder(5)

add5(3)

8

add2(1)

3

**Iterators**

When an object is used in a for ... in statement, the object’s \_\_iter\_\_() method is called and the returned value should be an iterator. At that point, the interpreter iterates over the result, assigning each object returned from the iterator to the loop variable in the for ... in statement.

class Lister(object):

def \_\_init\_\_(self, \*args):

self.items = tuple(args)

def \_\_iter\_\_(self):

return (i for i in self.items)

l = Lister('a', 'b', 'c')

for letter in l:

print(letter)

Here is the same example using a generator function instead of a generator expression.

class Lister(object):

def \_\_init\_\_(self, \*args):

self.items = tuple(args)

def \_\_iter\_\_(self):

for i in self.items:

yield i

l = Lister('a', 'b', 'c')

for letter in l:

print(letter,)

***Function and method arguments*:**

Always use **self** for the first argument to instance methods.

Always use **cls** for the first argument to class methods.

**classmethod and staticmethod**

A class method in Python is defined by creating a method on a class in the standard way, but applying the classmethod decorator to the method.

instead of self, the class method’s first argument is named cls

classmethod must have a reference to a class object as the first parameter, whereas staticmethod can have no parameters at all.

**classmethod**

class Song(object):

def \_\_init\_\_(self, title, artist):

self.title = title

self.artist = artist

def \_\_str\_\_(self):

return ('"%(title)s" by %(artist)s' %

self.\_\_dict\_\_)

@classmethod

def create\_songs(cls, songlist):

for artist, title in songlist:

yield cls(title, artist)

songs = (('Glen Hansard', 'Leave'),

('Stevie Ray Vaughan', 'Lenny'))

for song in Song.create\_songs(songs):

print(song)

The important difference is that static methods receive neither an instance object nor a class object as the first argument. They only receive the passed arguments.a

class Song(object):

def \_\_init\_\_(self, title, artist):

self.title = title

self.artist = artist

def \_\_str\_\_(self):

return ('"%(title)s" by %(artist)s' %

self.\_\_dict\_\_)

@staticmethod

def create\_songs(songlist):

for artist, title in songlist:

yield Song(title, artist)

songs = (('Glen Hansard', 'Leave'),

('Stevie Ray Vaughan', 'Lenny'))

for song in Song.create\_songs(songs):

print(song)

class A(object):

def foo(self,x):

print("executing foo(%s,%s)"%(self,x))

@classmethod

def class\_foo(cls,x):

print("executing class\_foo(%s,%s)"%(cls,x))

@staticmethod

def static\_foo(x):

print("executing static\_foo(%s)"%x )

a=A()

print(a.class\_foo)

print(a.static\_foo)

Output:

<bound method A.class\_foo of <class '\_\_main\_\_.A'>> #has class reference

<function A.static\_foo at 0x00A7C540> #doesnt have a class reference

# Python Iterators

They are elegantly implemented within for loops, comprehensions, generators etc. but hidden in plain sight.

Iterator in Python is simply an [object](https://www.programiz.com/python-programming/class) that can be iterated upon. An object which will return data, one element at a time.

Technically speaking, Python **iterator object** must implement two special methods, \_\_iter\_\_() and \_\_next\_\_(), collectively called the **iterator protocol**.

# define a list

my\_list = [4, 7, 0, 3]

# get an iterator using iter()

my\_iter = iter(my\_list)

## iterate through it using next()

#prints 4

print(next(my\_iter))

#prints 7

print(next(my\_iter))

## next(obj) is same as obj.\_\_next\_\_()

#prints 0

print(my\_iter.\_\_next\_\_())

#prints 3

print(my\_iter.\_\_next\_\_())

## This will raise error, no items left

next(my\_iter)

**OUTPUT**

4

7

0

3

## Why generators are used in Python?

### 1. Easy to Implement

### 2. Memory Efficient

A normal function to return a sequence will create the entire sequence in memory before returning the result. This is an overkill if the number of items in the sequence is very large.

Generator implementation of such sequence is memory friendly and is preferred since it only produces one item at a time.

### 3. Represent Infinite Stream

### 4. Pipelining Generators

**Return**

The return statement is where all the local variables are destroyed and the resulting value is given back (returned) to the caller. Should the same function be called some time later, the function will get a fresh new set of variables.

**Yield**

But what if the local variables aren't thrown away when we exit a function? This implies that we can resume the function where we left off. This is where the concept of generators are introduced and the yield statement resumes where the function left off.

**def** rev\_str(my\_str):  
 length = len(my\_str)  
 **for** i **in** range(length):  
 **yield** my\_str[i]  
  
  
**for** char **in** rev\_str(**"hello"**):  
 print(char)  
   
*# For loop to reverse the string  
# Output:  
# o  
# l  
# l  
# e  
# h*

Xrange and range

The variable storing the **range** created by range() **takes more memory** as compared to variable storing the range using xrange(). The basic reason for this is the return type of range() is list and xrange() is xrange() object.

# Deque in Python

**Operations on deque :**

**1. append()** :- This function is used to **insert** the value in its argument to the **right end** of deque.

**2. appendleft()** :- This function is used to **insert** the value in its argument to the **left end** of deque.

**3. pop()** :- This function is used to **delete** an argument from the **right end** of deque.

**4. popleft()** :- This function is used to **delete** an argument from the **left end** of deque.

**5. index(ele, beg, end)** :- This function **returns the first index of the value** mentioned in arguments, **starting searching from beg till end** index.

**6. insert(i, a)** :- This function **inserts the value** mentioned in arguments(a) **at index(i)** specified in arguments.

**7. remove()** :- This function **removes the first occurrence** of value mentioned in arguments.

**8. count()** :- This function **counts the number of occurrences** of value mentioned in arguments.

**9. extend(iterable)** :- This function is used to **add multiple values at the right end** of deque. The argument passed is an iterable.

**10. extendleft(iterable)** :- This function is used to **add multiple values at the left end** of deque. The argument passed is an iterable. **Order is reversed** as a result of left appends.

**11. reverse()** :- This function is used to **reverse order** of deque elements.

**12. rotate()** :- This function **rotates the deque** by the number specified in arguments. **If the number specified is negative, rotation occurs to left. Else rotation is to right.**

**map, filter, lambda**  
   
map – The map() function applies a function to every member of iterable and returns the result. If there are multiple arguments, map() returns a list consisting of tuples containing the corresponding items from all iterables.  
   
filter – It takes a function returning True or False and applies it to a sequence, returning a list of only those members of the sequence for which the function returned True.  
   
lambda- Python provides the ability to create a simple (no statements allowed internally) anonymous inline function called lambda function. Using lambda and map you can have two for loops in one line.

# Python program to test map, filter and lambda

# Function to test map

def cube(x):

    return x\*\*2

# Driver to test above function

# Program for working of map

print "MAP EXAMPLES"

cubes = map(cube, range(10))

print cubes

print "LAMBDA EXAMPLES"

# first parentheses contains a lambda form, that is

# a squaring function and second parentheses represents

# calling lambda

print (lambda x: x\*\*2)(5)

# Make function of two arguments that return their product

print (lambda x, y: x\*y)(3, 4)

print "FILTER EXAMPLE"

special\_cubes = filter(lambda x: x > 9 and x < 60, cubes)

print special\_cubes

Output:

MAP EXAMPLES

[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]

LAMBDA EXAMPLES

25

12

FILTER EXAMPLE

[16, 25, 36, 49]

## Types of Design Patterns

|  |  |
| --- | --- |
| 1 | **Creational Patterns** These design patterns provide a way to create objects while hiding the creation logic, rather than instantiating objects directly using new operator. This gives program more flexibility in deciding which objects need to be created for a given use case. Singleton, Abstract factory, factory, builder pattern are the examples. |
| 2 | **Structural Patterns** These design patterns concern class and object composition. Concept of inheritance is used to compose interfaces and define ways to compose objects to obtain new functionalities. Adapter pattern, aggregate, Proxy pattern are the examples. |
| 3 | **Behavioral Patterns** These design patterns are specifically concerned with communication between objects. Iterator pattern, strategy pattern, and state pattern are the examples. |
| 4 | **J2EE Patterns** These design patterns are specifically concerned with the presentation tier. These patterns are identified by Sun Java Center. |

**Singleton pattern:**

Use **\_\_new\_\_** when you need to control the creation of a new instance. Use **\_\_init\_\_** when you need to control initialization of a new instance.

# Singleton/SingletonPattern.py

class OnlyOne:

class \_\_OnlyOne:

def \_\_init\_\_(self, arg):

self.val = arg

def \_\_str\_\_(self):

return repr(self) + self.val

instance = None

def \_\_init\_\_(self, arg):

if not OnlyOne.instance:

OnlyOne.instance = OnlyOne.\_\_OnlyOne(arg)

else:

OnlyOne.instance.val = arg

def \_\_getattr\_\_(self, name):

return getattr(self.instance, name)

class OnlyMe:

pass

x = OnlyMe()

y = OnlyOne('eggs')

z = OnlyOne('spam')

print('---z---')

print(z)

print('---x---')

print(x)

print('---y---')

print(y)

print('---x---')

print(x)

print('---y---')

print(y)

print('---z---')

print(z)

print("------------------------------------------------------------------")

class Singleton2(object):

\_instance = None

def \_\_new\_\_(cls, \*args, \*\*kwargs):

if not cls.\_instance:

cls.\_instance = super(Singleton2, cls).\_\_new\_\_(

cls, \*args, \*\*kwargs)

return cls.\_instance

if \_\_name\_\_ == '\_\_main\_\_':

s1 = Singleton2()

s2 = Singleton2()

print(s1)

print(s2)

if (id(s1) == id(s2)):

print("Same")

else:

print("Different")

print("------------------------------------------------------------------")

class Singleton3(object):

def \_\_new\_\_(cls):

if not hasattr(cls, 'instance'):

cls.instance = super(Singleton3, cls).\_\_new\_\_(cls)

return cls.instance

singleton = Singleton3()

another\_singleton = Singleton3()

print(singleton)

print(another\_singleton)

print(singleton is another\_singleton)

# Factory method

We may not always know what kind of objects we want to create in advance.  
Some objects can be created only at execution time after a user requests so.

Examples when you may use a **factory method**:

* A user may click on a certain button that creates an object.
* A user may create several new documents of different types.
* If a user starts a webbrowser, the browser does not know in advance how many tabs (where every tab is an object) will be opened.

class Car(object):

def factory(type):

if type == "Racecar":

return Racecar()

if type == "Van":

return Van()

assert 0, "Bad car creation: " + type

factory = staticmethod(factory)

class Racecar(Car):

def drive(self): print("Racecar driving.")

class Van(Car):

def drive(self): print("Van driving.")

# Create object using factory.

obj = Car.factory("Racecar")

obj.drive()

obj2 = Car.factory("Racecar")

print(obj)

print(obj2)

GIL

[**https://docs.python.org/2/faq/programming.html**](https://docs.python.org/2/faq/programming.html)

[**http://www.geeksforgeeks.org/python/**](http://www.geeksforgeeks.org/python/)

[**http://embeddedgeeks.com/content/python-interview-questions**](http://embeddedgeeks.com/content/python-interview-questions)

**https://www.youtube.com/watch?v=DEwgZNC-KyE**