Introduction to OOPs in Python

Python is a multi-paradigm programming language. Meaning, it supports different programming approach.

One of the popular approach to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).

An object has two characteristics:

- attributes
- behavior

Let's take an example:

Parrot is an object,

- name, age, color are attributes
- singing, dancing are behavior

The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).

In Python, the concept of OOP follows some basic principles:

Inheritance	A process of using details from a new class without modifying existing class.
Encapsulation	Hiding the private details of a class from other objects.
Polymorphism	A concept of using common operation in different ways for different data input.

Class

A class is a blueprint for the object.

We can think of class as an sketch of a parrot with labels. It contains all the details about the name, colors, size etc. Based on these descriptions, we can study about the parrot. Here, parrot is an object.

The example for class of parrot can be :

- class Parrot:

  pass

Here, we use class keyword to define an empty class Parrot. From class, we construct instances. An instance is a specific object created from a particular class.

Object

An object (instance) is an instantiation of a class. When class is defined, only the description for the object is defined. Therefore, no memory or storage is allocated.

The example for object of parrot class can be:

obj = Parrot()

Here, obj is object of class Parrot.

Suppose we have details of parrot. Now, we are going to show how to build the class and objects of parrot.

Example 1: Creating Class and Object in Python

```
class Parrot:


# class attribute

species = "bird"


# instance attribute

def __init__(self, name, age):

self.name = name

self.age = age
```

```
# instantiate the Parrot class

blu = Parrot("Blu", 10)

woo = Parrot("Woo", 15)


# access the class attributes

print("Blu is a {}".format(blu.__class__.species))

print("Woo is also a {}".format(woo.__class__.species))


# access the instance attributes

print("{} is {} years old".format( blu.name, blu.age))

print("{} is {} years old".format( woo.name, woo.age))
```

When we run the program, the output will be:

- Blu is a bird
- Woo is also a bird
- Blu is 10 years old

Woo is 15 years old

In the above program, we create a class with name Parrot. Then, we define attributes. The attributes are a characteristic of an object.

Then, we create instances of the Parrot class. Here, blu and woo are references (value) to our new objects.

Then, we access the class attribute using __class __.species. Class attributes are same for all instances of a class. Similarly, we access the instance attributes using blu.name and blu.age. However, instance attributes are different for every instance of a class.

Methods

Methods are functions defined inside the body of a class. They are used to define the behaviors of an object.

Example 2 : Creating Methods in Python

```
    class Parrot:


    # instance attributes

    def __init__(self, name, age):

    self.name = name

    self.age = age


    # instance method

    def sing(self, song):

    return "{} sings {}".format(self.name, song)


    def dance(self):

    return "{} is now dancing".format(self.name)


# instantiate the object

blu = Parrot("Blu", 10)


# call our instance methods

print(blu.sing("'Happy'"))
```

print(blu.dance())

When we run program, the output will be:

- Blu sings 'Happy'

Blu is now dancing

In the above program, we define two methods i.e sing() and dance(). These are called instance method because they are called on an instance object i.e blu.

Inheritance

Inheritance is a way of creating new class for using details of existing class without modifying it. The newly formed class is a derived class (or child class). Similarly, the existing class is a base class (or parent class).

Example 3: Use of Inheritance in Python

```
# parent class
class Bird:


    def __init__(self):
    print("Bird is ready")


    def whoisThis(self):
    print("Bird")


    def swim(self):
    print("Swim faster")
```

```
# child class

class Penguin(Bird):


    def __init__(self):

    # call super() function

    super().__init__()

    print("Penguin is ready")


    def whoisThis(self):

    print("Penguin")


    def run(self):

    print("Run faster")


peggy = Penguin()

peggy.whoisThis()

peggy.swim()

peggy.run()
```

When we run this program, the output will be:

- Bird is ready
- Penguin is ready
- Penguin
- Swim faster

Run faster

In the above program, we created two classes i.e. Bird (parent class) and Penguin (child class). The child class inherits the functions of parent class. We can see this from swim() method. Again, the child class modified the behavior of parent class. We can see this from whoisThis() method. Furthermore, we extend the functions of parent class, by creating a new run() method.

Additionally, we use super() function before __init__() method. This is because we want to pull the content of __init__() method from the parent class into the child class.

Encapsulation

Using OOP in Python, we can restrict access to methods and variables. This prevent data from direct modification which is called encapsulation. In Python, we denote private attribute using underscore as prefix i.e single “ _ “ or double “ __“.

Example 4: Data Encapsulation in Python

```
class Computer:


    def __init__(self):

    self.__maxprice = 900


    def sell(self):

    print("Selling Price: {}".format(self.__maxprice))


    def setMaxPrice(self, price):

    self.__maxprice = price


c = Computer()

c.sell()
```

```
# change the price

c.__maxprice = 1000

c.sell()


# using setter function

c.setMaxPrice(1000)

c.sell()
```

When we run this program, the output will be:

- Selling Price: 900
- Selling Price: 900

Selling Price: 1000

In the above program, we defined a class Computer. We use __init__() method to store the maximum selling price of computer. We tried to modify the price. However, we can’t change it because Python treats the __maxprice as private attributes. To change the value, we used a setter function i.e setMaxPrice() which takes price as parameter.

Polymorphism

Polymorphism is an ability (in OOP) to use common interface for multiple form (data types).

Suppose, we need to color a shape, there are multiple shape option (rectangle, square, circle). However we could use same method to color any shape. This concept is called Polymorphism.

Example 5: Using Polymorphism in Python

```
class Parrot:
    def fly(self):

    print("Parrot can fly")


    def swim(self):

    print("Parrot can't swim")


class Penguin:


    def fly(self):

    print("Penguin can't fly")


    def swim(self):

    print("Penguin can swim")


# common interface

def flying_test(bird):

    bird.fly()


#instantiate objects

blu = Parrot()

peggy = Penguin()


# passing the object
```

flying_test(blu)

flying_test(peggy)

When we run above program, the output will be:

- Parrot can fly

Penguin can't fly

In the above program, we defined two classes Parrot and Penguin. Each of them have common method fly() method. However, their functions are different. To allow polymorphism, we created common interface i.e flying_test() function that can take any object. Then, we passed the objects blu and peggy in the flying_test() function, it ran effectively.

Key Points to Remember:

- The programming gets easy and efficient.
- The class is sharable, so codes can be reused.
- The productivity of programmars increases
- Data is safe and secure with data abstraction.

What are classes and objects in Python?

Python is an object oriented programming language. Unlike procedure oriented programming, where the main emphasis is on functions, object oriented programming stress on objects. Object is simply a collection of data (variables) and methods (functions) that act on those data. And, class is a blueprint for the object.
We can think of class as a sketch (prototype) of a house. It contains all the details about the floors, doors, windows etc. Based on these descriptions we build the house. House is the object.
As, many houses can be made from a description, we can create many objects from a class. An object is also called an instance of a class and the process of creating this object is called **instantiation**.

Defining a Class in Python

Like function definitions begin with the keyword def, in Python, we define a class using the keyword class.
The first string is called docstring and has a brief description about the class. Although not mandatory, this is recommended.
Here is a simple class definition.

```
class MyNewClass:
  '''This is a docstring. I have created a new class'''
  pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions.
There are also special attributes in it that begins with double underscores (__). For example, __doc__ gives us the docstring of that class.
As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class MyClass:
   "This is my second class"
   a = 10
   def func(self):
         print('Hello')

# Output: 10
print(MyClass.a)

# Output: <function MyClass.func at 0x0000000003079BF8>
print(MyClass.func)

# Output: 'This is my second class'
print(MyClass.__doc__)
```

When you run the program, the output will be:

```
10
<function 0x7feaa932eae8="" at="" myclass.func="">
This is my second class
```

Creating an Object in Python

We saw that the class object could be used to access different attributes.
It can also be used to create new object instances (instantiation) of that class. The procedure to create an object is similar to a function call.

1. >>> ob = MyClass()

This will create a new instance object named ob. We can access attributes of objects using the object name prefix.
Attributes may be data or method. Method of an object are corresponding functions of that class. Any function object that is a class attribute defines a method for objects of that class.
This means to say, since MyClass.func is a function object (attribute of class), ob.func will be a method object.

```
class MyClass:
   "This is my second class"
   a = 10
   def func(self):
          print('Hello')

# create a new MyClass
ob = MyClass()

# Output: <function MyClass.func at 0x000000000335B0D0>
print(MyClass.func)

# Output: <bound method MyClass.func of <__main__.MyClass object at
0x000000000332DEF0>>
print(ob.func)

# Calling function func()
# Output: Hello
```

You may have noticed the self parameter in function definition inside the class but, we called the method simply as ob.func() without any arguments. It still worked.

This is because, whenever an object calls its method, the object itself is passed as the first argument. So, ob.func() translates into MyClass.func(ob).

In general, calling a method with a list of n arguments is equivalent to calling the corresponding function with an argument list that is created by inserting the method's object before the first argument.
For these reasons, the first argument of the function in class must be the object itself. This is conventionally called self. It can be named otherwise but we highly recommend to follow the convention.

Now you must be familiar with class object, instance object, function object, method object and their differences. If not! You know ring me up

---

Constructors in Python

Class functions that begins with double underscore (__) are called special functions as they have special meaning.
Of one particular interest is the __init__() function. This special function gets called whenever a new object of that class is instantiated.
This type of function is also called constructors in Object Oriented Programming (OOP). We normally use it to initialize all the variables.

```
class ComplexNumber:
        def __init__(self,r = 0,i = 0):
        self.real = r
        self.imag = i

        def getData(self):
        print("{0}+{1}j".format(self.real,self.imag))

# Create a new ComplexNumber object
c1 = ComplexNumber(2,3)

# Call getData() function
# Output: 2+3j
c1.getData()

# Create another ComplexNumber object
# and create a new attribute 'attr'
c2 = ComplexNumber(5)
c2.attr = 10

# Output: (5, 0, 10)
print((c2.real, c2.imag, c2.attr))
```

```
# but c1 object doesn't have attribute 'attr'
# AttributeError: 'ComplexNumber' object has no attribute 'attr'
C1.attr
```

In the above example, we define a new class to represent complex numbers. It has two functions, __init__() to initialize the variables (defaults to zero) and getData() to display the number properly.
An interesting thing to note in the above step is that attributes of an object can be created on the fly. We created a new attribute attr for object c2 and we read it as well. But this did not create that attribute for object c1.

Deleting Attributes and Objects

Any attribute of an object can be deleted anytime, using the del statement. Try the following on the Python shell to see the output.

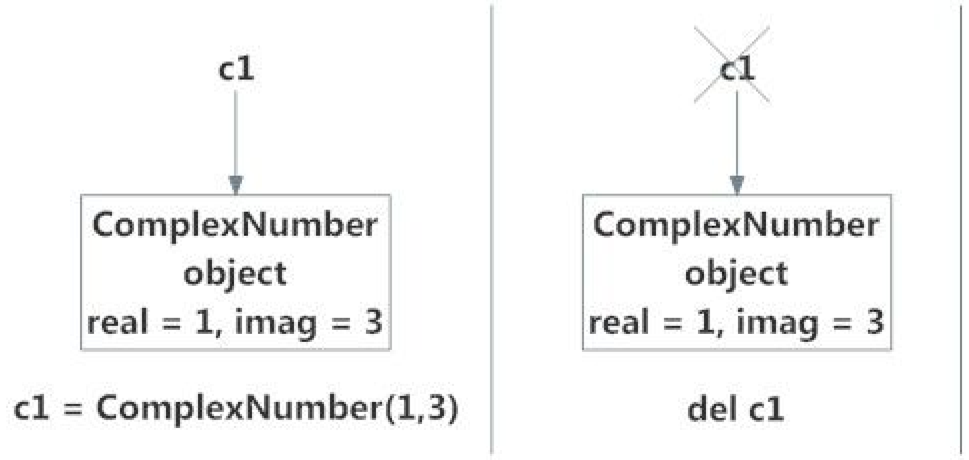
```
>>> c1 = ComplexNumber(2,3)
>>> del c1.imag
>>> c1.getData()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'imag'

>>> del ComplexNumber.getData
>>> c1.getData()
Traceback (most recent call last):
...
AttributeError: 'ComplexNumber' object has no attribute 'getData'
```

We can even delete the object itself, using the del statement.

```
>>> c1 = ComplexNumber(1,3)
>>> del c1
>>> c1
Traceback (most recent call last):
...
NameError: name 'c1' is not defined
```

Actually, it is more complicated than that. When we do c1 = ComplexNumber(1,3), a new instance object is created in memory and the name c1 binds with it.

On the command del c1, this binding is removed and the name c1 is deleted from the corresponding namespace. The object however continues to exist in memory and if no other name is bound to it, it is later automatically destroyed.
This automatic destruction of unreferenced objects in Python is also called garbage collection.


c1
ComplexNumber object real = 1, imag = 3
c1 = ComplexNumber(1,3)
c1
ComplexNumber object real = 1, imag = 3
del c1

What is Inheritance?

Inheritance is a powerful feature in object oriented programming.
It refers to defining a new class with little or no modification to an existing class. The new class is called **derived (or child) class** and the one from which it inherits is called the **base (or parent) class**.

Python Inheritance Syntax

```
class BaseClass:
  Body of base class
class DerivedClass(BaseClass):
  Body of derived class
```

Derived class inherits features from the base class, adding new features to it. This results into re-usability of code.

Example of Inheritance in Python

To demonstrate the use of inheritance, let us take an example.
A polygon is a closed figure with 3 or more sides. Say, we have a class called Polygon defined as follows.

1. class Polygon:

```
    def __init__(self, no_of_sides):
        self.n = no_of_sides
        self.sides = [0 for i in range(no_of_sides)]

    def inputSides(self):
        self.sides = [float(input("Enter side "+str(i+1)+" : ")) for i in range(self.n)]

    def dispSides(self):
        for i in range(self.n):
            print("Side",i+1,"is",self.sides[i])
```

This class has data attributes to store the number of sides, n and magnitude of each side as a list, sides.
Method inputSides() takes in magnitude of each side and similarly, dispSides() will display these properly.
A triangle is a polygon with 3 sides. So, we can created a class called Triangle which inherits from Polygon. This makes all the attributes available in class Polygon readily available in Triangle. We don't need to define them again (code re-usability). Triangle is defined as follows.

```
class Triangle(Polygon):
    def __init__(self):
        Polygon.__init__(self,3)

    def findArea(self):
        a, b, c = self.sides
        # calculate the semi-perimeter
        s = (a + b + c) / 2
        area = (s*(s-a)*(s-b)*(s-c)) ** 0.5
        print('The area of the triangle is %0.2f' %area)
```

However, class Triangle has a new method findArea() to find and print the area of the triangle.
Here is a sample run.

```
>>> t = Triangle()

>>> t.inputSides()
Enter side 1 : 3
Enter side 2 : 5
Enter side 3 : 4

>>> t.dispSides()
Side 1 is 3.0
Side 2 is 5.0
Side 3 is 4.0
```

```

>>> t.findArea()
The area of the triangle is 6.00
```

We can see that, even though we did not define methods like inputSides() or dispSides() for class Triangle, we were able to use them.
If an attribute is not found in the class, search continues to the base class. This repeats recursively, if the base class is itself derived from other classes.

Method Overriding in Python

In the above example, notice that __init__() method was defined in both classes, Triangle as well Polygon. When this happens, the method in the derived class overrides that in the base class. This is to say, __init__() in Triangle gets preference over the same in Polygon.

Generally when overriding a base method, we tend to extend the definition rather than simply replace it. The same is being done by calling the method in base class from the one in derived class (calling Polygon.__init__() from __init__() in Triangle).

A better option would be to use the built-in function super(). So, super().__init__(3) is equivalent to Polygon.__init__(self,3) and is preferred. You can learn more about the super() function in Python.
Two built-in functions isinstance() and issubclass() are used to check inheritances. Function isinstance() returns True if the object is an instance of the class or other classes derived from it. Each and every class in Python inherits from the base class object.

```
>>> isinstance(t,Triangle)
True

>>> isinstance(t,Polygon)
True

>>> isinstance(t,int)
False

>>> isinstance(t,object)
True
```

Similarly, issubclass() is used to check for class inheritance.

```
>>> issubclass(Polygon,Triangle)
False

```

```
>>> issubclass(Triangle,Polygon)
True

>>> issubclass(bool,int)
True
```

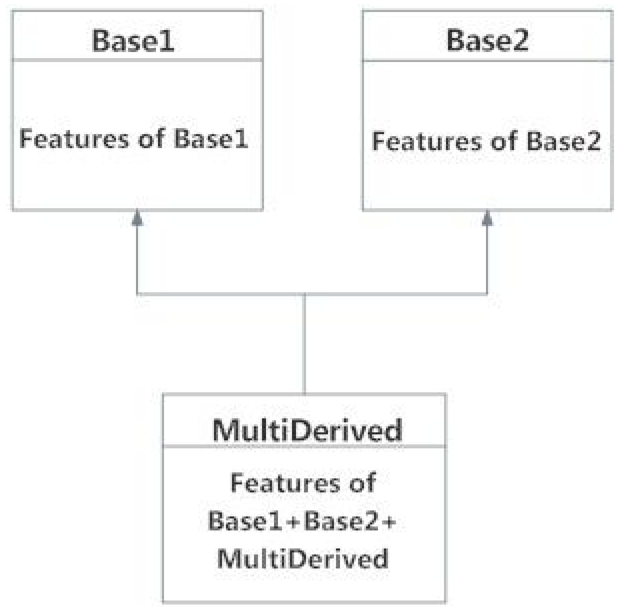
Multiple Inheritance in Python

Like C++, a class can be derived from more than one base classes in Python. This is called multiple inheritance.
In multiple inheritance, the features of all the base classes are inherited into the derived class.
The syntax for multiple inheritance is similar to single inheritance.

Example

```
class Base1:
  pass

class Base2:
  pass

class MultiDerived(Base1, Base2):
  pass
```

Here, MultiDerived is derived from classes Base1 and Base2.



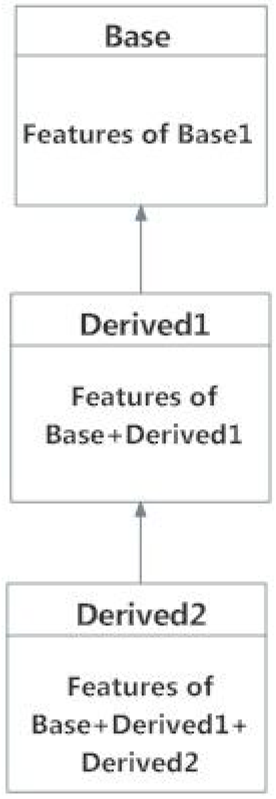
The class MultiDerived inherits from both Base1 and Base2.

Multilevel Inheritance in Python

On the other hand, we can also inherit form a derived class. This is called multilevel inheritance. It can be of any depth in Python.
In multilevel inheritance, features of the base class and the derived class is inherited into the new derived class.
An example with corresponding visualization is given below.

```
class Base:
  pass

class Derived1(Base):
  pass

class Derived2(Derived1):
  pass
```

Here, Derived1 is derived from Base, and Derived2 is derived from Derived1.


Base
Features of Base1
Derived1
Features of Base+Derived1
Derived2
Features of Base+Derived1+ Derived2

Method Resolution Order in Python

Every class in Python is derived from the class object. It is the most base type in Python.
So technically, all other class, either built-in or user-defines, are derived classes and all objects are instances of object class.

```
# Output: True
print(issubclass(list,object))

# Output: True
print(isinstance(5.5,object))

# Output: True
print(isinstance("Hello",object))
```

In the multiple inheritance scenario, any specified attribute is searched first in the current class. If not found, the search continues into parent classes in depth-first, left-right fashion without searching same class twice.

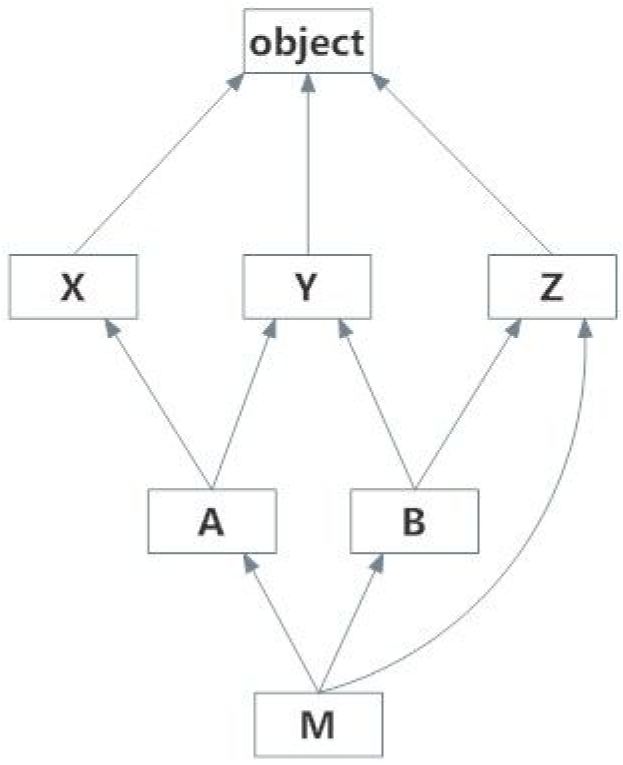
So, in the above example of MultiDerived class the search order is [MultiDerived, Base1, Base2, object]. This order is also called linearization of MultiDerived class and the set of rules used to find this order is called **Method Resolution Order (MRO)**.

MRO must prevent local precedence ordering and also provide monotonicity. It ensures that a class always appears before its parents and in case of multiple parents, the order is same as tuple of base classes.

MRO of a class can be viewed as the __mro__ attribute or mro() method. The former returns a tuple while latter returns a list.

```
>>> MultiDerived.__mro__
(<class '__main__.MultiDerived'>,
<class '__main__.Base1'>,
<class '__main__.Base2'>,
<class 'object'>)

>>> MultiDerived.mro()
[<class '__main__.MultiDerived'>,
<class '__main__.Base1'>,
<class '__main__.Base2'>,
<class 'object'>]
```

Here is a little more complex multiple inheritance example and its visualization along with the MRO.



```
class X: pass
class Y: pass
class Z: pass

class A(X,Y): pass
class B(Y,Z): pass

class M(B,A,Z): pass

# Output:
# [<class '__main__.M'>, <class '__main__.B'>,
# <class '__main__.A'>, <class '__main__.X'>,
# <class '__main__.Y'>, <class '__main__.Z'>,
# <class 'object'>]

print(M.mro())
```

What is operator overloading in Python?

Python operators work for built-in classes. But same operator behaves differently with different types. For example, the + operator will, perform arithmetic addition on two numbers, merge two lists and concatenate two strings.
This feature in Python, that allows same operator to have different meaning according to the context is called operator overloading.
So what happens when we use them with objects of a user-defined class? Let us consider the following class, which tries to simulate a point in 2-D coordinate system.

```
class Point:
   def __init__(self, x = 0, y = 0):
      self.x = x
      self.y = y
```

Now, run the code and try to add two points in Python shell.

```

>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> p1 + p2
Traceback (most recent call last):
...
TypeError: unsupported operand type(s) for +: 'Point' and 'Point'
```

Whoa! That's a lot of complains. TypeError was raised since Python didn't know how to add two Point objects together.
However, the good news is that we can teach this to Python through operator overloading. But first, let's get a notion about special functions.

Special Functions in Python

Class functions that begins with double underscore __ are called special functions in Python. This is because, well, they are not ordinary. The __init__() function we defined above, is one of them. It gets called every time we create a new object of that class. There are a ton of special functions in Python.
Using special functions, we can make our class compatible with built-in functions.

```
>>> p1 = Point(2,3)
>>> print(p1)
<__main__.Point object at 0x00000000031F8CC0>
```

That did not print well. But if we define __str__() method in our class, we can control how it gets printed. So, let's add this to our class.

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)
```

Now let's try the print() function again.

```

>>> p1 = Point(2,3)
>>> print(p1)
(2,3)
```

That's better. Turns out, that this same method is invoked when we use the built-in function str() or format().

```

>>> str(p1)
'(2,3)'

>>> format(p1)
'(2,3)'
```

So, when you do str(p1) or format(p1), Python is internally doing p1.__str__(). Hence the name, special functions.
Ok, now back to operator overloading.

Overloading the + Operator in Python

To overload the + sign, we will need to implement __add__() function in the class. With great power comes great responsibility. We can do whatever we like, inside this function. But it is sensible to return a Point object of the coordinate sum.

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)
```

```
def __add__(self,other):
    x = self.x + other.x
    y = self.y + other.y
    return Point(x,y)
```

Now let's try that addition again.

```
>>> p1 = Point(2,3)
>>> p2 = Point(-1,2)
>>> print(p1 + p2)
(1,5)
```

What actually happens is that, when you do p1 + p2, Python will call p1.__add__(p2) which in turn is Point.__add__(p1,p2). Similarly, we can overload other operators as well. The special function that we need to implement is tabulated below.

Operator	Expression	Internally
Addition	p1 + p2	p1.__add__(p2)
Subtraction	p1 - p2	p1.__sub__(p2)
Multiplication	p1 * p2	p1.__mul__(p2)
Power	p1 ** p2	p1.__pow__(p2)
Division	p1 / p2	p1.__truediv__(p2)
Floor Division	p1 // p2	p1.__floordiv__(p2)
Remainder (modulo)	p1 % p2	p1.__mod__(p2)
Bitwise Left Shift	p1 << p2	p1.__lshift__(p2)
Bitwise Right Shift	p1 >> p2	p1.__rshift__(p2)
Bitwise AND	p1 & p2	p1.__and__(p2)
Bitwise OR	p1 | p2	p1.__or__(p2)
Bitwise XOR	p1 ^ p2	p1.__xor__(p2)
Bitwise NOT	~p1	p1.__invert__()

Overloading Comparison Operators in Python

Python does not limit operator overloading to arithmetic operators only. We can overload comparison operators as well.
Suppose, we wanted to implement the less than symbol < symbol in our Point class.
Let us compare the magnitude of these points from the origin and return the result for this purpose. It can be implemented as follows.

```
class Point:
    def __init__(self, x = 0, y = 0):
        self.x = x
        self.y = y

    def __str__(self):
        return "({0},{1})".format(self.x,self.y)

    def __lt__(self,other):
        self_mag = (self.x ** 2) + (self.y ** 2)
        other_mag = (other.x ** 2) + (other.y ** 2)
        return self_mag < other_mag
```

Try these sample runs in Python shell.

```

>>> Point(1,1) < Point(-2,-3)
True

>>> Point(1,1) < Point(0.5,-0.2)
False

>>> Point(1,1) < Point(1,1)
False
```

Similarly, the special functions that we need to implement, to overload other comparison operators are tabulated below.

Operator	Expression	Internally
Less than	p1 < p2	p1.__lt__(p2)
Less than or equal to	p1 <= p2	p1.__le__(p2)

Equal to	p1 == p2	p1.__eq__(p2)
Not equal to	p1 != p2	p1.__ne__(p2)
Greater than	p1 > p2	p1.__gt__(p2)
Greater than or equal to	p1 >= p2	p1.__ge__(p2)

What are iterators in Python?

Iterators are everywhere in Python. They are elegantly implemented within for loops, comprehensions, generators etc. but hidden in plain sight.
Iterator in Python is simply an object 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**.
An object is called **iterable** if we can get an iterator from it. Most of built-in containers in Python like: list, tuple, string etc. are iterables.
The iter() function (which in turn calls the __iter__() method) returns an iterator from them.

Iterating Through an Iterator in Python

We use the next() function to manually iterate through all the items of an iterator. When we reach the end and there is no more data to be returned, it will raise StopIteration. Following is an example.

```
# 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)
```

A more elegant way of automatically iterating is by using the for loop. Using this, we can iterate over any object that can return an iterator, for example list, string, file etc.

```
>>> for element in my_list:
...     print(element)
...
4
7
0
3
```

How for loop actually works?

As we see in the above example, the for loop was able to iterate automatically through the list. In fact the for loop can iterate over any iterable. Let's take a closer look at how the for loop is actually implemented in Python.

```
for element in iterable:
    # do something with element
```

Is actually implemented as.

```
# create an iterator object from that iterable
iter_obj = iter(iterable)
```

```

# infinite loop
while True:
    try:
        # get the next item
        element = next(iter_obj)
        # do something with element
    except StopIteration:
        # if StopIteration is raised, break from loop
        break
```

So internally, the for loop creates an iterator object, iter_obj by calling iter() on the iterable.
Ironically, this for loop is actually an infinite while loop.
Inside the loop, it calls next() to get the next element and executes the body of the for loop with this value. After all the items exhaust, StopIteration is raised which is internally caught and the loop ends. Note that any other kind of exception will pass through.

Building Your Own Iterator in Python

Building an iterator from scratch is easy in Python. We just have to implement the methods __iter__() and __next__().
The __iter__() method returns the iterator object itself. If required, some initialization can be performed.
The __next__() method must return the next item in the sequence. On reaching the end, and in subsequent calls, it must raise StopIteration.
Here, we show an example that will give us next power of 2 in each iteration. Power exponent starts from zero up to a user set number.

```
class PowTwo:
    """Class to implement an iterator
    of powers of two"""

    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n <= self.max:
```

```
    result = 2 ** self.n
    self.n += 1
    return result
  else:
    raise StopIteration
```

Now we can create an iterator and iterate through it as follows.

```
>>> a = PowTwo(4)
>>> i = iter(a)
>>> next(i)
1
>>> next(i)
2
>>> next(i)
4
>>> next(i)
8
>>> next(i)
16
>>> next(i)
Traceback (most recent call last):
...
StopIteration
```

We can also use a for loop to iterate over our iterator class.

```
>>> for i in PowTwo(5):
...     print(i)
...
1
2
4
8
16
32
```

Python Infinite Iterators

It is not necessary that the item in an iterator object has to exhaust. There can be infinite iterators (which never ends). We must be careful when handling such iterator.
Here is a simple example to demonstrate infinite iterators.

The built-in function iter() can be called with two arguments where the first argument must be a callable object (function) and second is the sentinel. The iterator calls this function until the returned value is equal to the sentinel.

```
>>> int()
0

>>> inf = iter(int,1)
>>> next(inf)
0
>>> next(inf)
0
```

We can see that the int() function always returns 0. So passing it as iter(int,1) will return an iterator that calls int() until the returned value equals 1. This never happens and we get an infinite iterator.
We can also built our own infinite iterators. The following iterator will, theoretically, return all the odd numbers.

```
class InfIter:
    """Infinite iterator to return all
        odd numbers"""

    def __iter__(self):
        self.num = 1
        return self

    def __next__(self):
        num = self.num
        self.num += 2
        return num
```

A sample run would be as follows.

```
>>> a = iter(InfIter())
>>> next(a)
1
>>> next(a)
3
>>> next(a)
5
>>> next(a)
7
```

And so on...
Be careful to include a terminating condition, when iterating over these type of infinite iterators.
The advantage of using iterators is that they save resources. Like shown above, we could get all the odd numbers without storing the entire number system in memory. We can have infinite items (theoretically) in finite memory.
Iterator also makes our code look cool.

What are generators in Python?

There is a lot of overhead in building an iterator in Python; we have to implement a class with __iter__() and __next__() method, keep track of internal states, raise StopIteration when there was no values to be returned etc.
This is both lengthy and counter intuitive. Generator comes into rescue in such situations.
Python generators are a simple way of creating iterators. All the overhead we mentioned above are automatically handled by generators in Python.
Simply speaking, a generator is a function that returns an object (iterator) which we can iterate over (one value at a time).

How to create a generator in Python?

It is fairly simple to create a generator in Python. It is as easy as defining a normal function with yield statement instead of a return statement.
If a function contains at least one yield statement (it may contain other yield or return statements), it becomes a generator function. Both yield and return will return some value from a function.
The difference is that, while a return statement terminates a function entirely, yield statement pauses the function saving all its states and later continues from there on successive calls.

Differences between Generator function and a Normal function

Here is how a generator function differs from a normal function.

- Generator function contains one or more yield statement.
- When called, it returns an object (iterator) but does not start execution immediately.
- Methods like __iter__() and __next__() are implemented automatically. So we can iterate through the items using next().
- Once the function yields, the function is paused and the control is transferred to the caller.

- Local variables and their states are remembered between successive calls.
- Finally, when the function terminates, StopIteration is raised automatically on further calls.

Here is an example to illustrate all of the points stated above. We have a generator function named my_gen() with several yield statements.

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n
```

An interactive run in the interpreter is given below. Run these in the Python shell to see the output.

```
>>> # It returns an object but does not start execution immediately.
>>> a = my_gen()

>>> # We can iterate through the items using next().
>>> next(a)
This is printed first
1
>>> # Once the function yields, the function is paused and the control is transferred to the caller.

>>> # Local variables and theirs states are remembered between successive calls.
>>> next(a)
This is printed second
2

>>> next(a)
This is printed at last
3
```

```

>>> # Finally, when the function terminates, StopIteration is raised automatically on further calls.
>>> next(a)
Traceback (most recent call last):
...
StopIteration
>>> next(a)
Traceback (most recent call last):
...
StopIteration
```

One interesting thing to note in the above example is that, the value of variable n is remembered between each call.
Unlike normal functions, the local variables are not destroyed when the function yields.
Furthermore, the generator object can be iterated only once.
To restart the process we need to create another generator object using something like a = my_gen().
Note: One final thing to note is that we can use generators with for loops directly.
This is because, a for loop takes an iterator and iterates over it using next() function. It automatically ends when StopIteration is raised. Check here to know how a for loop is actually implemented in Python.

```
# A simple generator function
def my_gen():
    n = 1
    print('This is printed first')
    # Generator function contains yield statements
    yield n

    n += 1
    print('This is printed second')
    yield n

    n += 1
    print('This is printed at last')
    yield n

# Using for loop
for item in my_gen():
    print(item)
```

When you run the program, the output will be:

```
This is printed first
1
This is printed second
2
This is printed at last
3
```

Python Generators with a Loop

The above example is of less use and we studied it just to get an idea of what was happening in the background.
Normally, generator functions are implemented with a loop having a suitable terminating condition.
Let's take an example of a generator that reverses a string.

```
def rev_str(my_str):
    length = len(my_str)
    for i in range(length - 1,-1,-1):
        yield my_str[i]

# For loop to reverse the string
# Output:
# o
# l
# l
# e
# h
for char in rev_str("hello"):
     print(char)
```

In this example, we use range() function to get the index in reverse order using the for loop.
It turns out that this generator function not only works with string, but also with other kind of iterables like list, tuple etc.

Python Generator Expression

Simple generators can be easily created on the fly using generator expressions. It makes building generators easy.

Same as lambda function creates an anonymous function, generator expression creates an anonymous generator function.
The syntax for generator expression is similar to that of a list comprehension in Python. But the square brackets are replaced with round parentheses.
The major difference between a list comprehension and a generator expression is that while list comprehension produces the entire list, generator expression produces one item at a time.
They are kind of lazy, producing items only when asked for. For this reason, a generator expression is much more memory efficient than an equivalent list comprehension.

```
# Initialize the list
my_list = [1, 3, 6, 10]

# square each term using list comprehension
# Output: [1, 9, 36, 100]
[x**2 for x in my_list]

# same thing can be done using generator expression
# Output: <generator object <genexpr> at 0x0000000002EBDAF8>
(x**2 for x in my_list)
```

We can see above that the generator expression did not produce the required result immediately. Instead, it returned a generator object with produces items on demand.

```
# Intialize the list
my_list = [1, 3, 6, 10]

a = (x**2 for x in my_list)
# Output: 1
print(next(a))

# Output: 9
print(next(a))

# Output: 36
print(next(a))

# Output: 100
print(next(a))

# Output: StopIteration
next(a)
```

Generator expression can be used inside functions. When used in such a way, the round parentheses can be dropped.

```
>>> sum(x**2 for x in my_list)
146

>>> max(x**2 for x in my_list)
100
```

Why generators are used in Python?

There are several reasons which make generators an attractive implementation to go for.

1. Easy to Implement

Generators can be implemented in a clear and concise way as compared to their iterator class counterpart. Following is an example to implement a sequence of power of 2's using iterator class.

```
class PowTwo:
    def __init__(self, max = 0):
        self.max = max

    def __iter__(self):
        self.n = 0
        return self

    def __next__(self):
        if self.n > self.max:
            raise StopIteration

        result = 2 ** self.n
        self.n += 1
        return result
```

This was lengthy. Now lets do the same using a generator function.

```
def PowTwoGen(max = 0):
    n = 0
    while n < max:
        yield 2 ** n
        n += 1
```

Since, generators keep track of details automatically, it was concise and much cleaner in implementation.

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

Generators are excellent medium to represent an infinite stream of data. Infinite streams cannot be stored in memory and since generators produce only one item at a time, it can represent infinite stream of data.
The following example can generate all the even numbers (at least in theory).

```
def all_even():
  n = 0
  while True:
     yield n
     n += 2
```

4. Pipelining Generators

Generators can be used to pipeline a series of operations. This is best illustrated using an example.
Suppose we have a log file from a famous fast food chain. The log file has a column (4th column) that keeps track of the number of pizza sold every hour and we want to sum it to find the total pizzas sold in 5 years.
Assume everything is in string and numbers that are not available are marked as 'N/A'. A generator implementation of this could be as follows.

```
with open('sells.log') as file:
  pizza_col = (line[3] for line in file)
  per_hour = (int(x) for x in pizza_col if x != 'N/A')
  print("Total pizzas sold = ",sum(per_hour))
```

This pipelining is efficient and easy to read (and yes, a lot cooler!).

Nonlocal variable in a nested function

Before getting into what a closure is, we have to first understand what a nested function and nonlocal variable is.

A function defined inside another function is called a nested function. Nested functions can access variables of the enclosing scope.
In Python, these non-local variables are read only by default and we must declare them explicitly as non-local (using nonlocal keyword) in order to modify them.
Following is an example of a nested function accessing a non-local variable.

```
def print_msg(msg):
# This is the outer enclosing function

    def printer():
# This is the nested function
        print(msg)

    printer()

# We execute the function
# Output: Hello
print_msg("Hello")
```

We can see that the nested function printer() was able to access the non-local variable msg of the enclosing function.

Defining a Closure Function

In the example above, what would happen if the last line of the function print_msg() returned the printer() function instead of calling it? This means the function was defined as follows.

```
def print_msg(msg):
# This is the outer enclosing function

    def printer():
# This is the nested function
        print(msg)

    return printer  # this got changed

# Now let's try calling this function.
# Output: Hello
another = print_msg("Hello")
another()
```

That's unusual.
The print_msg() function was called with the string "Hello" and the returned function was bound to the name another. On calling another(), the message was still remembered although we had already finished executing the print_msg() function.
This technique by which some data ("Hello") gets attached to the code is called **closure in Python**.
This value in the enclosing scope is remembered even when the variable goes out of scope or the function itself is removed from the current namespace.
Try running the following in the Python shell to see the output.

```
>>> del print_msg
>>> another()
Hello
>>> print_msg("Hello")
Traceback (most recent call last):
...
NameError: name 'print_msg' is not defined
```

When do we have a closure?

As seen from the above example, we have a closure in Python when a nested function references a value in its enclosing scope.
The criteria that must be met to create closure in Python are summarized in the following points.

- We must have a nested function (function inside a function).
- The nested function must refer to a value defined in the enclosing function.
- The enclosing function must return the nested function.

When to use closures?

So what are closures good for?
Closures can avoid the use of global values and provides some form of data hiding. It can also provide an object oriented solution to the problem.
When there are few methods (one method in most cases) to be implemented in a class, closures can provide an alternate and more elegant solutions. But when the number of attributes and methods get larger, better implement a class.
Here is a simple example where a closure might be more preferable than defining a class and making objects. But the preference is all yours.

```
def make_multiplier_of(n):
    def multiplier(x):
        return x * n
    return multiplier

# Multiplier of 3
times3 = make_multiplier_of(3)

# Multiplier of 5
times5 = make_multiplier_of(5)

# Output: 27
print(times3(9))

# Output: 15
print(times5(3))

# Output: 30
print(times5(times3(2)))
```

Decorators in Python make an extensive use of closures as well.
On a concluding note, it is good to point out that the values that get enclosed in the closure function can be found out.
All function objects have a __closure__ attribute that returns a tuple of cell objects if it is a closure function. Referring to the example above, we know times3 and times5 are closure functions.

```
>>> make_multiplier_of.__closure__
>>> times3.__closure__
(<cell at 0x0000000002D155B8: int object at 0x000000001E39B6E0>,)
```

The cell object has the attribute cell_contents which stores the closed value.

```
>>> times3.__closure__[0].cell_contents
3
>>> times5.__closure__[0].cell_contents
5
```

What are decorators in Python?

Python has an interesting feature called **decorators** to add functionality to an existing code.
This is also called **metaprogramming** as a part of the program tries to modify another part of the program at compile time.

Prerequisites for learning decorators

In order to understand about decorators, we must first know a few basic things in Python.
We must be comfortable with the fact that, everything in Python (Yes! Even classes), are objects. Names that we define are simply identifiers bound to these objects. Functions are no exceptions, they are objects too (with attributes). Various different names can be bound to the same function object.
Here is an example.

```
def first(msg):
    print(msg)

first("Hello")

second = first
second("Hello")
```

When you run the code, both functions first and second gives same output. Here, the names first and second refer to the same function object.
Now things start getting weirder.
Functions can be passed as arguments to another function.
If you have used functions like map, filter and reduce in Python, then you already know about this.
Such function that take other functions as arguments are also called **higher order functions**.
Here is an example of such a function.

```
def inc(x):
    return x + 1

def dec(x):
    return x - 1

def operate(func, x):
    result = func(x)
    return result
```

We invoke the function as follows.

```
>>> operate(inc,3)
4
>>> operate(dec,3)
```

4. 2

Furthermore, a function can return another function.

```
def is_called():
    def is_returned():
        print("Hello")
    return is_returned

new = is_called()

#Outputs "Hello"
new()
```

Here, is_returned() is a nested function which is defined and returned, each time we call is_called().
Finally, we must know about closures in Python.

Getting back to Decorators

Functions and methods are called **callable** as they can be called.
In fact, any object which implements the special method __call__() is termed callable. So, in the most basic sense, a decorator is a callable that returns a callable.
Basically, a decorator takes in a function, adds some functionality and returns it.

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

def ordinary():
    print("I am ordinary")
```

When you run the following codes in shell,

```
>>> ordinary()
I am ordinary

>>> # let's decorate this ordinary function
>>> pretty = make_pretty(ordinary)
>>> pretty()
I got decorated
```

```
I am ordinary
```

In the example shown above, make_pretty() is a decorator. In the assignment step.

```
pretty = make_pretty(ordinary)
```

The function ordinary() got decorated and the returned function was given the name pretty.
We can see that the decorator function added some new functionality to the original function. This is similar to packing a gift. The decorator acts as a wrapper. The nature of the object that got decorated (actual gift inside) does not alter. But now, it looks pretty (since it got decorated).
Generally, we decorate a function and reassign it as,

```
ordinary = make_pretty(ordinary).
```

This is a common construct and for this reason, Python has a syntax to simplify this.
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. For example,

```
@make_pretty
def ordinary():
  print("I am ordinary")
```

is equivalent to

```
def ordinary():
  print("I am ordinary")
ordinary = make_pretty(ordinary)
```

This is just a syntactic sugar to implement decorators.

Decorating Functions with Parameters

The above decorator was simple and it only worked with functions that did not have any parameters. What if we had functions that took in parameters like below?

```
def divide(a, b):
  return a/b
```

This function has two parameters, a and b. We know, it will give error if we pass in b as 0.

```
>>> divide(2,5)
0.4
>>> divide(2,0)
Traceback (most recent call last):
...
```

```
ZeroDivisionError: division by zero
```

Now let's make a decorator to check for this case that will cause the error.

```
def smart_divide(func):
  def inner(a,b):
    print("I am going to divide",a,"and",b)
    if b == 0:
      print("Whoops! cannot divide")
      return

    return func(a,b)
  return inner

@smart_divide
def divide(a,b):
   return a/b
```

This new implementation will return None if the error condition arises.

```
>>> divide(2,5)
I am going to divide 2 and 5
0.4

>>> divide(2,0)
I am going to divide 2 and 0
Whoops! cannot divide
```

In this manner we can decorate functions that take parameters.
A keen observer will notice that parameters of the nested inner() function inside the decorator is same as the parameters of functions it decorates. Taking this into account, now we can make general decorators that work with any number of parameter.
In Python, this magic is done as function(*args, **kwargs). In this way, args will be the tuple of positional arguments and kwargs will be the dictionary of keyword arguments. An example of such decorator will be.

```
def works_for_all(func):
  def inner(*args, **kwargs):
      print("I can decorate any function")
      return func(*args, **kwargs)
  return inner
```

Chaining Decorators in Python

Multiple decorators can be chained in Python.
This is to say, a function can be decorated multiple times with different (or same) decorators.
We simply place the decorators above the desired function.

```
def star(func):
    def inner(*args, **kwargs):
        print("*" * 30)
        func(*args, **kwargs)
        print("*" * 30)
    return inner

def percent(func):
    def inner(*args, **kwargs):
        print("%" * 30)
        func(*args, **kwargs)
        print("%" * 30)
    return inner

@star
@percent
def printer(msg):
    print(msg)
printer("Hello")
```

This will give the output.

```
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
Hello
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
```

The above syntax of,

```
@star
@percent
def printer(msg):
    print(msg)
```

is equivalent to

```
def printer(msg):
    print(msg)
printer = star(percent(printer))
```

The order in which we chain decorators matter. If we had reversed the order as,

```
@percent
@star
def printer(msg):
    print(msg)
```

The execution would take place as,

```
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
******************************
Hello
******************************
%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
```

Python has a great concept called property which makes the life of an object oriented programmer much simpler.
Before defining and going into details of what @property is, let us first build an intuition on why it would be needed in the first place.

An Example To Begin With

Let us assume that you decide to make a class that could store the temperature in degree Celsius. It would also implement a method to convert the temperature into degree Fahrenheit. One way of doing this is as follows.

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

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32
```

We could make objects out of this class and manipulate the attribute temperature as we wished. Try these on Python shell.

```
>>> # create new object
>>> man = Celsius()

>>> # set temperature
>>> man.temperature = 37

>>> # get temperature
>>> man.temperature
37

>>> # get degrees Fahrenheit
>>> man.to_fahrenheit()
98.60000000000001
```

The extra decimal places when converting into Fahrenheit is due to the floating point arithmetic error (try 1.1 + 2.2 in the Python interpreter).
Whenever we assign or retrieve any object attribute like temperature, as show above, Python searches it in the object's __dict__ dictionary.

```
>>> man.__dict__
{'temperature': 37}
```

Therefore, man.temperature internally becomes man.__dict__['temperature'].
Now, let's further assume that our class got popular among clients and they started using it in their programs. They did all kinds of assignments to the object.
One fateful day, a trusted client came to us and suggested that temperatures cannot go below -273 degree Celsius (students of thermodynamics might argue that it's actually -273.15), also called the absolute zero. He further asked us to implement this value constraint. Being a company that strive for customer satisfaction, we happily heeded the suggestion and released version 1.01 (an upgrade of our existing class).

Using Getters and Setters

An obvious solution to the above constraint will be to hide the attribute temperature (make it private) and define new getter and setter interfaces to manipulate it. This can be done as follows.

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

    def to_fahrenheit(self):
        return (self.get_temperature() * 1.8) + 32

    # new update
    def get_temperature(self):
        return self._temperature
    def set_temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        self._temperature = value
```

We can see above that new methods get_temperature() and set_temperature() were defined and furthermore, temperature was replaced with _temperature. An underscore (_) at the beginning is used to denote private variables in Python.

```
>>> c = Celsius(-277)
Traceback (most recent call last):
...
ValueError: Temperature below -273 is not possible

>>> c = Celsius(37)
>>> c.get_temperature()
37
>>> c.set_temperature(10)

>>> c.set_temperature(-300)
Traceback (most recent call last):
...
ValueError: Temperature below -273 is not possible
```

This update successfully implemented the new restriction. We are no longer allowed to set temperature below -273.
Please note that private variables don't exist in Python. There are simply norms to be followed. The language itself don't apply any restrictions.

```
>>> c._temperature = -300
>>> c.get_temperature()
-300
```

But this is not of great concern. The big problem with the above update is that, all the clients who implemented our previous class in their program have to modify their code from obj.temperature to obj.get_temperature() and all assignments like obj.temperature = val to obj.set_temperature(val).
This refactoring can cause headaches to the clients with hundreds of thousands of lines of codes.
All in all, our new update was not backward compatible. This is where property comes to rescue.

The Power of @property

The pythonic way to deal with the above problem is to use property. Here is how we could have achieved it.

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

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    def get_temperature(self):
        print("Getting value")
        return self._temperature

    def set_temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        print("Setting value")
        self._temperature = value

    temperature = property(get_temperature,set_temperature)
```

And, issue the following code in shell once you run it.

```
>>> c = Celsius()
```

We added a print() function inside get_temperature() and set_temperature() to clearly observe that they are being executed.
The last line of the code, makes a property object temperature. Simply put, property attaches some code (get_temperature and set_temperature) to the member attribute accesses (temperature).
Any code that retrieves the value of temperature will automatically call get_temperature() instead of a dictionary (__dict__) look-up. Similarly, any code that assigns a value to temperature will automatically call set_temperature(). This is one cool feature in Python.
We can see above that set_temperature() was called even when we created an object.

Can you guess why?
The reason is that when an object is created, __init__() method gets called. This method has the line self.temperature = temperature. This assignment automatically called set_temperature().

```
>>> c.temperature
Getting value
0
```

Similarly, any access like c.temperature automatically calls get_temperature(). This is what property does. Here are a few more examples.

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

>>> c.to_fahrenheit()
Getting value
98.60000000000001
```

By using property, we can see that, we modified our class and implemented the value constraint without any change required to the client code. Thus our implementation was backward compatible and everybody is happy.
Finally note that, the actual temperature value is stored in the private variable _temperature. The attribute temperature is a property object which provides interface to this private variable.

Digging Deeper into Property

In Python, property() is a built-in function that creates and returns a property object. The signature of this function is

```
property(fget=None, fset=None, fdel=None, doc=None)
```

where, fget is function to get value of the attribute, fset is function to set value of the attribute, fdel is function to delete the attribute and doc is a string (like a comment). As seen from the implementation, these function arguments are optional. So, a property object can simply be created as follows.

```
>>> property()
<property object at 0x0000000003239B38>
```

A property object has three methods, getter(), setter(), and deleter() to specify fget, fset and fdel at a later point. This means, the line

```
temperature = property(get_temperature,set_temperature)
```

could have been broken down as

```
# make empty property
temperature = property()
# assign fget
temperature = temperature.getter(get_temperature)
# assign fset
temperature = temperature.setter(set_temperature)
```

These two pieces of codes are equivalent.
Programmers familiar with decorators in Python can recognize that the above construct can be implemented as decorators.
We can further go on and not define names get_temperature and set_temperature as they are unnecessary and pollute the class namespace. For this, we reuse the name temperature while defining our getter and setter functions. This is how it can be done.

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

    def to_fahrenheit(self):
        return (self.temperature * 1.8) + 32

    @property
    def temperature(self):
        print("Getting value")
        return self._temperature

    @temperature.setter
    def temperature(self, value):
        if value < -273:
            raise ValueError("Temperature below -273 is not possible")
        print("Setting value")
        self._temperature = value
```

The above implementation is both, simple and recommended way to make properties. You will most likely encounter these types of constructs when looking for property in Python.

Copy an Object in Python

In Python, we use = operator to create a copy of an object. You may think that this creates a new object; it doesn't. It only creates a new variable that shares the reference of the original object.
Let's take an example where we create a list named old_list and pass an object reference to new_list using = operator.

Example 1: Copy using = operator

```
old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 'a']]
```

```
new_list = old_list

new_list[2][2] = 9

print('Old List:', old_list)
print('ID of Old List:', id(old_list))

print('New List:', new_list)
print('ID of New List:', id(new_list))
```

When we run above program, the output will be:

```
Old List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of Old List: 140673303268168

New List: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
ID of New List: 140673303268168
```

As you can see from the output both variables old_list and new_list shares the same id i.e 140673303268168.
So, if you want to modify any values in new_list or old_list, the change is visible in both.

Essentially, sometimes you may want to have the original values unchanged and only modify the new values or vice versa. In Python, there are two ways to create copies:

1. Shallow Copy
2. Deep Copy

To make these copy work, we use the copy module.

Copy Module

We use the copy module of Python for shallow and deep copy operations. Suppose, you need to copy the compound list say x. For example:

```
import copy
copy.copy(x)
copy.deepcopy(x)
```

Here, the copy() return a shallow copy of x. Similarly, deepcopy() return a deep copy of x.

Shallow Copy

A shallow copy creates a new object which stores the reference of the original elements.

So, a shallow copy doesn't create a copy of nested objects, instead it just copies the reference of nested objects. This means, a copy process does not recurse or create copies of nested objects itself.

Example 2: Create a copy using shallow copy

```
import copy

old_list = [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
new_list = copy.copy(old_list)

print("Old list:", old_list)
print("New list:", new_list)
```

When we run the program , the output will be:

```
Old list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
New list: [[1, 2, 3], [4, 5, 6], [7, 8, 9]]
```

In above program, we created a nested list and then shallow copy it using copy() method.
This means it will create new and independent object with same content. To verify this, we print the both old_list and new_list.
To confirm that new_list is different from old_list, we try to add new nested object to original and check it.

Example 3: Adding [4, 4, 4] to old_list, using shallow copy

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)

old_list.append([4, 4, 4])

print("Old list:", old_list)
print("New list:", new_list)
```

When we run the program, it will output:

```
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3], [4, 4, 4]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, we created a shallow copy of old_list. The new_list contains references to original nested objects stored in old_list. Then we add the new list i.e [4, 4, 4] into old_list. This new sublist was not copied in new_list.
However, when you change any nested objects in old_list, the changes appear in new_list.

Example 4: Adding new nested object using Shallow copy

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.copy(old_list)

old_list[1][1] = 'AA'

print("Old list:", old_list)
print("New list:", new_list)
```

When we run the program, it will output:

```
Old list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 'AA', 2], [3, 3, 3]]
```

In the above program, we made changes to old_list i.e old_list[1][1] = 'AA'. Both sublists of old_list and new_list at index [1][1] were modified. This is because, both lists share the reference of same nested objects.

Deep Copy

A deep copy creates a new object and recursively adds the copies of nested objects present in the original elements.
Let’s continue with example 2. However, we are going to create deep copy using deepcopy() function present in copy module. The deep copy creates independent copy of original object and all its nested objects.

Example 5: Copying a list using deepcopy()

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)
```

```
print("Old list:", old_list)
print("New list:", new_list)
```

When we run the program, it will output:

```
Old list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

In the above program, we use deepcopy() function to create copy which looks similar. However, if you make changes to any nested objects in original object old_list, you'll see no changes to the copy new_list.

Example 6: Adding a new nested object in the list using Deep copy

```
import copy

old_list = [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
new_list = copy.deepcopy(old_list)

old_list[1][0] = 'BB'

print("Old list:", old_list)
print("New list:", new_list)
```

When we run the program, it will output:

```
Old list: [[1, 1, 1], ['BB', 2, 2], [3, 3, 3]]
New list: [[1, 1, 1], [2, 2, 2], [3, 3, 3]]
```

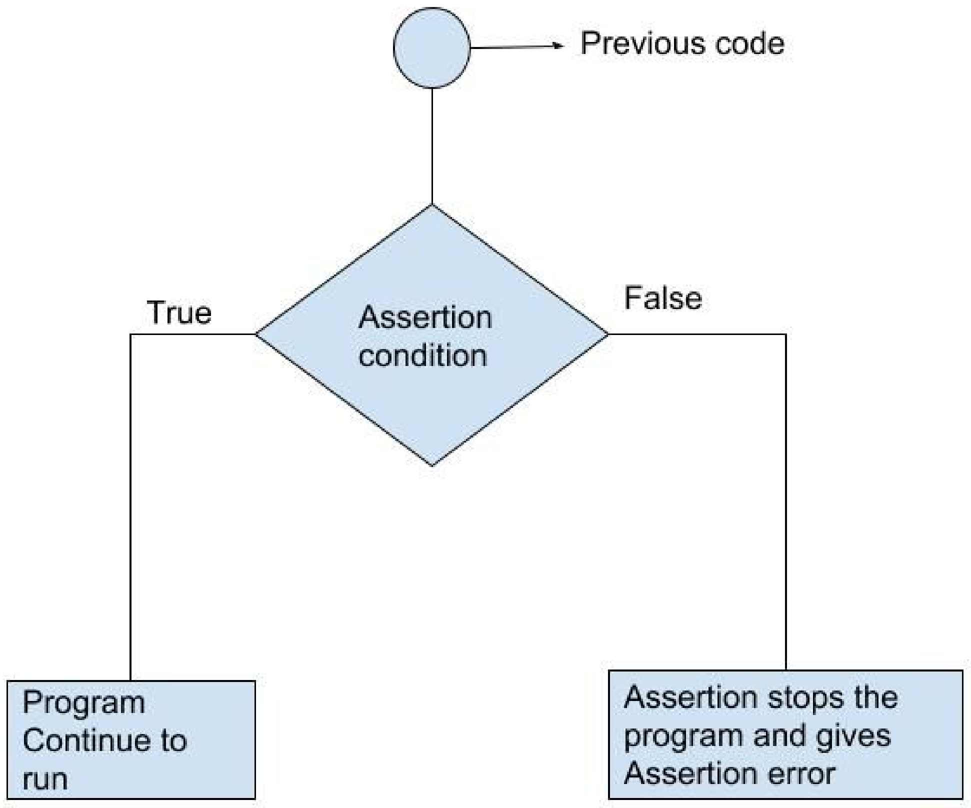
In the above program, when we assign a new value to old_list, we can see only the old_list is modified. This means, both the old_list and the new_list are independent. This is because the old_list was recursively copied, which is true for all its nested objects.

What is Assertion?

Assertions are statements that assert or state a fact confidently in your program. For example, while writing a division function, you're confident the divisor shouldn't be zero, you assert divisor is not equal to zero.

Assertions are simply boolean expressions that checks if the conditions return true or not. If it is true, the program does nothing and move to the next line of code. However, if it's false, the program stops and throws an error.

It is also a debugging tool as it brings the program on halt as soon as any error is occurred and shows on which point of the program error has occurred.
You can learn more about assertions in the article: The benefits of programming with Assertions
We can be clear by looking at the flowchart below:



Python assert Statement

Python has built-in assert statement to use assertion condition in the program. assert statement has a condition or expression which is supposed to be always true. If the condition is false assert halts the program and gives an AssertionError.

Syntax for using Assert in Pyhton:
assert <condition>
assert <condition>,<error message>
In Python we can use assert statement in two ways as mentioned above.

1. assert statement has a condition and if the condition is not satisfied the program will stop and give AssertionError.

2. assert statement can also have a condition and a optional error message. If the condition is not satisfied assert stops the program and gives AssertionError along with the error message.

Let's take an example, where we have a function which will calculate the average of the values passed by the user and the value should not be an empty list. We will use assert statement to check the parameter and if the length is of the passed list is zero, program halts.

Example 1: Using assert without Error Message

```
def avg(marks):
   assert len(marks) != 0
   return sum(marks)/len(marks)

mark1 = []
print("Average of mark1:",avg(mark1))
```

When we run the above program, the output will be:

```
AssertionError
```

We got an error as we passed an empty list mark1 to assert statement, the condition became false and assert stops the program and give AssertionError.
Now let's pass another list which will satisfy the assert condition and see what will be our output.

Example 2: Using assert with error message

```
def avg(marks):
   assert len(marks) != 0,"List is empty."
   return sum(marks)/len(marks)

mark2 = [55,88,78,90,79]
print("Average of mark2:",avg(mark2))

mark1 = []
print("Average of mark1:",avg(mark1))
```

When we run the above program, the output will be:

```
Average of mark2: 78.0
AssertionError: List is empty.
```

We passed a non-empty list mark2 and also an empty list mark1 to the avg() function and we got output for mark2 list but after that we got an error AssertionError: List is empty. The assert condition was satisfied by the mark2 list and program to continue to run. However, mark1 doesn't satisfy the condition and gives an AssertionError.

Key Points to Remember

- Assertions are the condition or boolean expression which are always supposed to be true in the code.
- assert statement takes an expression and optional message.
- assert statement is used to check types, values of argument and the output of the function.
- assert statement is used as debugging tool as it halts the program at the point where an error occurs.

A **Re**gular **Ex**pression (RegEx) is a sequence of characters that defines a search pattern. For example,

1. ^a...s$

The above code defines a RegEx pattern. The pattern is: **any five letter string starting with a and ending with s**.
A pattern defined using RegEx can be used to match against a string.

Expression	String	Matched?
^a...s$	abs	No match
	alias	Match
	abyss	Match
	Alias	No match
	An abacus	No match

Python has a module named re to work with RegEx. Here's an example:

```
1.
2. import re
3.
4. pattern = '^a...s$'
```

```
test_string = 'abyss'
result = re.match(pattern, test_string)

if result:
 print("Search successful.")
else:
 print("Search unsuccessful.")
```

Here, we used re.match() function to search pattern within the test_string. The method returns a match object if the search is successful. If not, it returns None.

There are other several functions defined in the re module to work with RegEx. Before we explore that, let's learn about regular expressions themselves.

Specify Pattern Using RegEx

To specify regular expressions, metacharacters are used. In the above example, ^ and $ are metacharacters.

MetaCharacters

Metacharacters are characters that are interpreted in a special way by a RegEx engine. Here's a list of metacharacters:
[] . ^ $ * + ? {} () \ |

[] - Square brackets
Square brackets specifies a set of characters you wish to match.

Expression	String	Matched?
[abc]	a	1 match
	ac	2 matches
	Hey Jude	No match
	abc de ca	5 matches

Here, [abc] will match if the string you are trying to match contains any of the a, b or c.
You can also specify a range of characters using - inside square brackets.

- [a-e] is the same as [abcde].

- [1-4] is the same as [1234].
- [0-39] is the same as [01239].

You can complement (invert) the character set by using caret ^ symbol at the start of a square-bracket.

- [^abc] means any character except a or b or c.
- [^0-9] means any non-digit character.

. - **Period**
A period matches any single character (except newline '\n').

Expression	String	Matched?
..	a	No match
	ac	1 match
	acd	1 match
	acde	2 matches (contains 4 characters)

---

^ - **Caret**
The caret symbol ^ is used to check if a string **starts with** a certain character.

Expression	String	Matched?
^a	a	1 match
	abc	1 match
	bac	No match
^ab	abc	1 match
	acb	No match (starts with a but not followed by b)

$ - **Dollar**
The dollar symbol $ is used to check if a string **ends with** a certain character.

Expression	String	Matched?
a$	a	1 match

	formula	1 match
	cab	No match

*** - Star**
The star symbol * matches **zero or more occurrences** of the pattern left to it.

Expression	String	Matched?
ma*n	mn	1 match
	man	1 match
	maaan	1 match
	main	No match (a is not followed by n)
	woman	1 match

+ - Plus
The plus symbol + matches **one or more occurrences** of the pattern left to it.

Expression	String	Matched?
ma+n	mn	No match (no a character)
	man	1 match
	maaan	1 match
	main	No match (a is not followed by n)
	woman	1 match

? - **Question Mark**
The question mark symbol ? matches **zero or one occurrence** of the pattern left to it.

Expression	String	Matched?
ma?n	mn	1 match
	man	1 match
	maaan	No match (more than one a character)

	main	No match (a is not followed by n)
	woman	1 match

{} - **Braces**
Consider this code: {n,m}. This means at least n, and at most m repetitions of the pattern left to it.

Expression	String	Matched?
a{2,3}	abc dat	No match
	abc daat	1 match (at daat)
	aabc daaat	2 matches (at aabc and daaat)
	aabc daaaat	2 matches (at aabc and daaaat)

Let's try one more example. This RegEx [0-9]{2, 4} matches at least 2 digits but not more than 4 digits

Expression	String	Matched?
[0-9]{2,4}	ab123csde	1 match (match at ab123csde)
	12 and 345673	2 matches (at 12 and 345673)
	1 and 2	No match

| - **Alternation**
Vertical bar | is used for alternation (or operator).

Expression	String	Matched?
a|b	cde	No match
	ade	1 match (match at ade)
	acdbea	3 matches (at acdbea)

Here, a|b match any string that contains either a or b

() - **Group**

Parentheses () is used to group sub-patterns. For example, (a|b|c)xz match any string that matches either a or b or c followed by xz

Expression	String	Matched?
(a|b|c)xz	ab xz	No match
	abxz	1 match (match at abxz)
	axz cabxz	2 matches (at axzbc cabxz)

\ - **Backslash**
Backlash \ is used to escape various characters including all metacharacters. For example, \$a match if a string contains $ followed by a. Here, $ is not interpreted by a RegEx engine in a special way.
If you are unsure if a character has special meaning or not, you can put \ in front of it. This makes sure the character is not treated in a special way.

Special Sequences
Special sequences make commonly used patterns easier to write. Here's a list of special sequences:
\A - Matches if the specified characters are at the start of a string.

Expression	String	Matched?
\Athe	the sun	Match
	In the sun	No match

\b - Matches if the specified characters are at the beginning or end of a word.

Expression	String	Matched?
\bfoo	football	Match
	a football	Match
	afootball	No match
foo\b	the foo	Match
	the afoo test	Match

	the afootest	No match

\B - Opposite of \b. Matches if the specified characters are **not** at the beginning or end of a word.

Expression	String	Matched?
\Bfoo	football	No match
	a football	No match
	afootball	Match
foo\B	the foo	No match
	the afoo test	No match
	the afootest	Match

\d - Matches any decimal digit. Equivalent to [0-9]

Expression	String	Matched?
\d	12abc3	3 matches (at 12abc3)
	Python	No match

\D - Matches any non-decimal digit. Equivalent to [^0-9]

Expression	String	Matched?
\D	1ab34"50	3 matches (at 1ab34"50)
	1345	No match

\s - Matches where a string contains any whitespace character. Equivalent to [\t\n\r\f\v].

Expression	String	Matched?
\s	Python RegEx	1 match

	PythonRegEx	No match

\S - Matches where a string contains any non-whitespace character. Equivalent to [^ \t\n\r\f\v].

Expression	String	Matched?
\S	a b	2 matches (at a b)
		No match

\w - Matches any alphanumeric character (digits and alphabets). Equivalent to [a-zA-Z0-9_]. By the way, underscore _ is also considered an alphanumeric character.

Expression	String	Matched?
\w	12&": ;c	3 matches (at 12&": ;c)
	%"> !	No match

\W - Matches any non-alphanumeric character. Equivalent to [^a-zA-Z0-9_]

Expression	String	Matched?
\W	1a2%c	1 match (at 1a2%c)
	Python	No match

\Z - Matches if the specified characters are at the end of a string.

Expression	String	Matched?
\ZPython	I like Python	1 match
	I like Python	No match
	Python is fun.	No match

Python RegEx

Python has a module named re to work with regular expressions. To use it, we need to import the module.

```
import re
```

The module defines several functions and constants to work with RegEx.

---

re.findall()

The re.findall() method returns a list of strings containing all matches.

Example 1: re.findall()

```

# Program to extract numbers from a string

import re

string = 'hello 12 hi 89. Howdy 34'
pattern = '\d+'

result = re.findall(pattern, string)
print(result)

# Output: ['12', '89', '34']
```

If the pattern is no found, re.findall() returns an empty list.

---

re.split()

The re.split method splits the string where there is a match and returns a list of strings where the splits have occurred.

Example 2: re.split()

```

import re
```

```python

string = 'Twelve:12 Eighty nine:89.'
pattern = '\d+'

result = re.split(pattern, string)
print(result)

# Output: ['Twelve:', ' Eighty nine:', '.']
```

If the pattern is no found, re.split() returns a list containing an empty string.

You can pass maxsplit argument to the re.split() method. It's the maximum number of splits that will occur.

```python

import re

string = 'Twelve:12 Eighty nine:89 Nine:9.'
pattern = '\d+'

# maxsplit = 1
# split only at the first occurrence
result = re.split(pattern, string, 1)
print(result)

# Output: ['Twelve:', ' Eighty nine:89 Nine:9.']
```

By the way, the default value of maxsplit is 0; meaning all possible splits.

---

re.sub()

The syntax of re.sub() is:

```python
re.sub(pattern, replace, string)
```

The method returns a string where matched occurrences are replaced with the content of replace variable.

Example 3: re.sub()

```python

# Program to remove all whitespaces
```

```
import re

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

# empty string
replace = ''

new_string = re.sub(pattern, replace, string)
print(new_string)

# Output: abc12de23f456
```

If the pattern is no found, re.sub() returns the original string.

You can pass count as a fourth parameter to the re.sub() method. If omited, it results to 0. This will replace all occurrences.

```

import re

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'
replace = ''

new_string = re.sub(r'\s+', replace, string, 1)
print(new_string)

# Output:
# abc12de 23
# f45 6
```

---

re.subn()

The re.subn() is similar to re.sub() expect it returns a tuple of 2 items containing the new string and the number of substitutions made.

Example 4: re.subn()

```

# Program to remove all whitespaces
import re

# multiline string
string = 'abc 12\
de 23 \n f45 6'

# matches all whitespace characters
pattern = '\s+'

# empty string
replace = ''

new_string = re.subn(pattern, replace, string)
print(new_string)

# Output: ('abc12de23f456', 4)
```

---

re.search()

The re.search() method takes two arguments: a pattern and a string. The method looks for the first location where the RegEx pattern produces a match with the string.
If the search is successful, re.search() returns a match object; if not, it returns None.

```
match = re.search(pattern, str)
```

Example 5: re.search()

```

import re
```

```

string = "Python is fun"

# check if 'Python' is at the beginning
match = re.search('\APython', string)

if match:
 print("pattern found inside the string")
else:
 print("pattern not found")

# Output: pattern found inside the string
```

Here, match contains a match object.

Match object

You can get methods and attributes of a match object using dir() function.
Some of the commonly used methods and attributes of match objects are:

---

match.group()

The group() method returns the part of the string where there is a match.

Example 6: Match object

```

import re

string = '39801 356, 2102 1111'

# Three digit number followed by space followed by two digit number
pattern = '(\d{3}) (\d{2})'

# match variable contains a Match object.
match = re.search(pattern, string)

if match:
 print(match.group())
else:
 print("pattern not found")
```

```

# Output: 801 35
```

Here, match variable contains a match object.
Our pattern (\d{3}) (\d{2}) has two subgroups (\d{3}) and (\d{2}). You can get the part of the string of these parenthesized subgroups. Here's how:

```
>>> match.group(1)
'801'

>>> match.group(2)
'35'
>>> match.group(1, 2)
('801', '35')

>>> match.groups()
('801', '35')
```

---

match.start(), match.end() and match.span()

The start() function returns the index of the start of the matched substring. Similarly, end() returns the end index of the matched substring.

```
>>> match.start()
2
>>> match.end()
8
```

The span() function returns a tuple containing start and end index of the matched part.

```
>>> match.span()
(2, 8)
```

---

match.re and match.string

The re attribute of a matched object returns a regular expression object. Similarly, string attribute returns the passed string.

```
>>> match.re
re.compile('(\\d{3}) (\\d{2})')

>>> match.string
```

```
'39801 356, 2102 1111'
```

Using r prefix before RegEx

When r or R prefix is used before a regular expression, it means raw string. For example, '\n' is a new line whereas r'\n' means two characters: a backslash \ followed by n.
Backlash \ is used to escape various characters including all metacharacters. However, using r prefix makes \ treat as a normal character.

Example 7: Raw string using r prefix

```

import re

string = '\n and \r are escape sequences.'

result = re.findall(r'[\n\r]', string)
print(result)

# Output: ['\n', '\r']
```

What is Numpy?

Numpy is a Python library that supports multi-dimensional arrays and matrix. It also provides many basic and high-level mathematical functions that can be applied on these multi-dimensional arrays and matrices with less code footprint.

Why Numpy?

There are many reasons why Numpy package has been used by data scientist and analysts, machine learning experts, deep learning libraries, etc. We will go through some of the most basic advantages of Numpy over regular lists or arrays in Python.

- The code that involves arrays with Numpy package is precise to apply transformations or operations for each element of the multidimensional arrays unlike a Python List.
- Since n-dimensional arrays of Numpy use a single datatype and contiguous memory for storage, they take relatively lesser memory read and write times.

- The most useful features of Numpy package is the compact datatypes that it offers, like unsigned integers of 8 bits, 16 bits size and signed integers of different bit sizes, different floating point precisions, etc.

Numpy Datatypes

Choosing a numpy datatype for your multidimensional array is very important. If an appropriate datatype is chosen based on the requirement and data characteristics, then it can ease a large amount of memory and ofcourse the array operations can be faster.

Data type	Description
bool_	Boolean (True or False) stored as a byte
int_	Default integer type (same as C long; normally either int64 or int32)
intc	Identical to C int (normally int32 or int64)
intp	Integer used for indexing (same as C ssize_t; normally either int32 or int64)
int8	Byte (-128 to 127)
int16	Integer (-32768 to 32767)
int32	Integer (-2147483648 to 2147483647)
int64	Integer (-9223372036854775808 to 9223372036854775807)
uint8	Unsigned integer (0 to 255)
uint16	Unsigned integer (0 to 65535)
uint32	Unsigned integer (0 to 4294967295)
uint64	Unsigned integer (0 to 18446744073709551615)
float_	Shorthand for float64.
float16	Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
float32	Single precision float: sign bit, 8 bits exponent, 23 bits mantissa

float64	Double precision float: sign bit, 11 bits exponent, 52 bits mantissa
complex_	Shorthand for complex128.
complex64	Complex number, represented by two 32-bit floats (real and imaginary components)
complex128	Complex number, represented by two 64-bit floats (real and imaginary components)

In this Numpy Tutorial, we will use some of these datatypes, and whenever appropriate, we shall explain why a particular datatype is selected.

Import Numpy

We have already used import numpy statement while verifying the installation of numpy package using pip. This import is like any python package import. To use the functions of numpy, the package has to be imported at the start of the program.
The syntax of importing numpy package is:

```
import numpy
```

Python community usally uses the numpy package with an alias np.

```
import numpy as np
```

Now, you can use np to call all numpy functions. Going further, we will use this numpy alias version np in code for numpy.

Create a Basic One-dimensional Numpy Array

There are many ways to create an array using numpy. We go through each one of them with examples.

1. **array()**

```
>>> import numpy as np
>>> a = np.array([5, 8, 12])
>>> a
array([ 5, 8, 12])

```

np.array() function accepts a list and creates a numpy array.

7. **arange()** Note that its not arrange but **a range**. numpy.arange function acceps the starting and ending elements of a range, followed by the interval.

```
8.    >>> import numpy as np
9.    >>> a = np.arange(1, 15, 2)
10.   >>> a
11.   array([ 1, 3, 5, 7, 9, 11, 13])
12.
```

In the above example, 1 is the starting of the range and 15 is the ending but 15 is excluded. The interval is 2 and therefore the interval between adjacent elements of the array is 2.

13. **linspace()** This function creates a floating point array with linearly spaced values. numpy.linspace() function accepts starting and ending elements of the array, followed by number of elements.

```
14.   >>> import numpy as np
15.   >>> a = np.linspace(1, 15, 7)
16.   >>> a
17.   array([ 1. , 3.33333333, 5.66666667, 8. , 10.33333333, 12.66666667, 15. ])
18.
```

In the above example, 1 is the starting, 15 is the ending and 7 is the number of elements in the array.

Create Two Dimensional Numpy Array

In the previous section, we have learned to create a one dimensional array. Now we will take a step forward and learn how to reshape this one dimensional array to a two dimensional array. numpy.reshape() is the method used to reshape an array. reshape() function takes shape or dimension of the target array as the argument. In the following example the shape of target array is (3, 2). As we are creating a 2D array, we provided only two values in the shape. You can provide multiple dimensions as required in the shape, separated by comma.

```
>>> import numpy as np
>>> a = np.array([8, 2, 3, 7, 9, 1])
>>> a
array([8, 2, 3, 7, 9, 1])
>>> a = a.reshape(3, 2)
>>> a
array([[8, 2],
       [3, 7],
       [9, 1]])
>>>
```

The product of number of rows and number of columns should equal the size of the array. If the product of dimensions and the size of the array do not match, you will get an error as shown below:

```
>>> a.reshape(2, 4)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of size 6 into shape (2,4)
```

Now the array a is of shape [3,2]. You can reshape this to [2,3] or [1,6] by simply calling the reshape function on the array a.

```
>>> a = a.reshape(3, 2)
>>> a
array([[8, 2],
       [3, 7],
       [9, 1]])
>>> a = a.reshape(2, 3)
>>> a
array([[8, 2, 3],
       [7, 9, 1]])
>>> a = a.reshape(1, 6)
>>> a
array([[8, 2, 3, 7, 9, 1]])
>>>
```

In this Numpy Tutorial, we will go through some of the functions numpy provide to create and empty N-Dimensional array and initialize it zeroes, ones or some random values.

Create Numpy Array with all zeros

If you would like to create a numpy array of a specific size with all elements initialized to zero, you can use zeros() function. The zeros() function takes the shape of the array as argument.

```
>>> import numpy as np
>>> a = np.zeros((4, 5))
>>> a
array([[0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.],
       [0., 0., 0., 0., 0.]])
>>>
```

The default datatype of each element in the numpy array of zeros is numpy.float64.

```
>>> a.dtype
dtype('float64')
```

You can change the datatype of the elements by providing dtype argument to the zeros function. Let us change to the datatype numpy.int16.

```
>>> a = np.zeros((4, 5), np.int16)
>>> a
array([[0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0],
       [0, 0, 0, 0, 0]], dtype=int16)
```

Observe that the datatype int16 belongs to numpy package, hence np.int16.

Create Numpy Array with all ones

Similar to zeros() function, we have ones() function in numpy package. It creates ones with the specified array shape.

```
>>> import numpy as np
>>> a = np.ones((3,7))
>>> a
array([[1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1.],
       [1., 1., 1., 1., 1., 1., 1.]])
>>> a.dtype
dtype('float64')
>>>
```

The default datatype for each element of the array is numpy.float64. You can change this to another numpy datatype, by providing the dtype argument.

```
>>> a = np.ones((3,7), dtype=np.uint8)
>>> a
array([[1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1],
       [1, 1, 1, 1, 1, 1, 1]], dtype=uint8)
>>> a.dtype
dtype('uint8')
>>>
```

The datatype of the array created in the above example is numpy.uint8.

Create Numpy Array with random values

You can create an array with random values and specific dimensions using random function.

```
>>> a = np.random.random((3,2))
>>> a
array([[0.08832623, 0.80635251],
        [0.28991211, 0.9976203 ],
        [0.5372018 , 0.53011979]])
>>> a.dtype
dtype('float64')
>>>
```

In the above example, there are two random keywords. The first random is the sub-package of numpy, while second random is the function. There are other functions like randint(), etc., used to create array with random integers picked from a specific range.

```
>>> a = np.random.randint(0,8,12)
>>> a
array([5, 3, 4, 5, 1, 7, 3, 6, 5, 5, 2, 6])
>>>
```

Numpy Array Size

Array size is independent of its shape or dimensions. size pointer returns the total number of elements in the array.

```
>>> a
array([[8, 2],
        [3, 7],
        [9, 1]])
>>> a.size
6
>>>
```

The returned value is of type int.

Numpy Array Shape

You can get the shape or dimensions of the array using shape pointer to the array.

```
>>> a = np.zeros((4, 5), np.int16)
>>> a
array([[0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0],
        [0, 0, 0, 0, 0]], dtype=int16)
>>> a.shape
(4, 5)
>>>
```

The example holds good for any dimensional array. Let us create a four dimensional array and find its shape.

```
>>> a = np.zeros((4, 5, 2, 2), np.int16)
>>> a
array([[[[0, 0],
         [0, 0]],

        [[0, 0],
         [0, 0]],

        [[0, 0],
         [0, 0]],

        [[0, 0],
         [0, 0]],

        [[0, 0],
         [0, 0]]],


         [[[0, 0],
         [0, 0]],

        [[0, 0],
         [0, 0]],

        [[0, 0],
         [0, 0]],

        [[0, 0],
         [0, 0]],

        [[0, 0],
         [0, 0]]],


         [[[0, 0],
         [0, 0]],

        [[0, 0],
         [0, 0]],

        [[0, 0],
        [0, 0]],

      [[0, 0],
        [0, 0]],

      [[0, 0],
        [0, 0]]],


        [[[0, 0],
        [0, 0]],

      [[0, 0],
        [0, 0]],

      [[0, 0],
        [0, 0]],

      [[0, 0],
        [0, 0]],

      [[0, 0],
        [0, 0]]]], dtype=int16)
>>> a.shape
(4, 5, 2, 2)
>>>
```

Individual values of the shape can be accessed using index.

```
>>> a.shape
(4, 5, 2, 2)
>>> a.shape[1]
5
>>> a.shape[2]
2
```

Numpy Array level Relational Operators

In this section, you can glimpse the power of Numpy. You can apply relational operators to the whole array in a single statement.
In the following example, we have an array a, and we will check if each element of the array is greater than 4.

```
>>> a = np.random.randint(1,8,20).reshape(4,5)
>>> a
array([[1, 5, 5, 7, 7],
       [5, 5, 5, 6, 4],
       [4, 5, 2, 2, 1],
       [6, 6, 7, 4, 6]])
>>> b = a<5
>>> b
array([[ True, False, False, False, False],
       [False, False, False, False, True],
       [ True, False, True, True, True],
       [False, False, False, True, False]])
>>>
```

The output of a<5 is another array with the same shape of a, but with the boolean values representing if the element is less than 5.

Numpy Array level Arithmetic Operators

You can perform Arithmetic Operations as well at array level with ease and less code.
For example, if a is a numpy array, then

- a/4 divides all the elements of the array with 4 and returns the resulting array.
- a*3 multiplies all the elements of the array with 3 and returns the resulting array.

In the following example, we add 4 to each of the element in numpy array a using a single statement.

```
>>> a = np.random.randint(1,8,20).reshape(4,5)
>>> a
array([[1, 4, 2, 7, 1],
       [6, 2, 3, 7, 3],
       [4, 7, 7, 5, 7],
       [1, 2, 4, 4, 6]])
>>> a+4
array([[ 5, 8, 6, 11, 5],
       [10, 6, 7, 11, 7],
       [ 8, 11, 11, 9, 11],
       [ 5, 6, 8, 8, 10]])
>>>
```

Note that the arithmetic operator returns the resulting array, but does not modify the actual array. To modify the actual array, you have to use the modifier operators like +=, /=, *=, etc.

```
>>> a = np.random.randint(1,8,20).reshape(4,5)
>>> a
array([[1, 4, 2, 7, 1],
        [6, 2, 3, 7, 3],
        [4, 7, 7, 5, 7],
        [1, 2, 4, 4, 6]])
>>> a+4
array([[ 5, 8, 6, 11, 5],
        [10, 6, 7, 11, 7],
        [ 8, 11, 11, 9, 11],
        [ 5, 6, 8, 8, 10]])
>>> a
array([[1, 4, 2, 7, 1],
        [6, 2, 3, 7, 3],
        [4, 7, 7, 5, 7],
        [1, 2, 4, 4, 6]])
>>> a += 4
>>> a
array([[ 5, 8, 6, 11, 5],
        [10, 6, 7, 11, 7],
        [ 8, 11, 11, 9, 11],
        [ 5, 6, 8, 8, 10]])
>>>
```

Numpy Mathematical Functions

While introducing numpy to you, we have gone through the point that Numpy is created for Numerical Analysis in Python. Hence, you might expect that Numpy provides a huge collection of Mathematical Functions. And it is true. Numpy provides statistical functions, trigonometric functions, linear algebra functions, etc.
In this Numpy Tutorial, we will go through some of the basic mathematical functions provided by Numpy.

Numpy Sum Function – numpy.sum()

In this example, we will take an array and find the sum of the elements in it. numpy.sum() function not only allows us to calculate the sum of all elements in the array, but also along a specific axis as well.

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.sum(a)
10
>>> np.sum(a, axis=0)
array([4, 6])
>>> np.sum(a, axis=1)
array([3, 7])
```

Numpy Mean Function – numpy.mean()

In this example, we will take an array and find the mean. Mean is the average of elements of an array. numpy.mean() function not only allows us to calculate the mean of the complete array, but also along a specific axis as well.

```
>>> a = np.array([[1, 2], [3, 4]])
>>> np.mean(a)
2.5
>>> np.mean(a, axis=0)
array([ 2., 3.])
>>> np.mean(a, axis=1)
array([ 1.5, 3.5])
```

Numpy libraries used above
Mean

numpy.ma.mean(*self*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*) = *<numpy.ma.core._frommethod object>*
Returns the average of the array elements along given axis.
Masked entries are ignored, and result elements which are not finite will be masked.
Weighted average.
Examples

```
>>>
>>> a = np.ma.array([1,2,3], mask=[False, False, True])
>>> a
masked_array(data = [1 2 --],
             mask = [False False  True],
       fill_value = 999999)
>>> a.mean()
1.5
```

Sum
numpy.sum(*a*, *axis=None*, *dtype=None*, *out=None*, *keepdims=<no value>*, *initial=<no value>*
Sum of array elements over a given axis.

Parameters:

a : array_like
Elements to sum.

axis : None or int or tuple of ints, optional
Axis or axes along which a sum is performed. The default, axis=None, will sum all of the elements of the input array. If axis is negative it counts from the last to the first axis.
New in version 1.7.0.
If axis is a tuple of ints, a sum is performed on all of the axes specified in the tuple instead of a single axis or all the axes as before.

dtype : dtype, optional
The type of the returned array and of the accumulator in which the elements are summed. The dtype of *a* is used by default unless *a* has an integer dtype of less precision than the default platform integer. In that case, if *a* is signed then the platform integer is used while if *a* is unsigned then an unsigned integer of the same precision as the platform integer is used.

out : ndarray, optional
Alternative output array in which to place the result. It must have the same shape as the expected output, but the type of the output values will be cast if necessary.

keepdims : bool, optional
If this is set to True, the axes which are reduced are left in the result as dimensions with size one. With this option, the result will broadcast correctly against the input array.
If the default value is passed, then *keepdims* will not be passed through to the sum method of sub-classes of ndarray, however any non-default value will be. If the sub-class’ method does not implement *keepdims* any exceptions will be raised.

initial : scalar, optional
Starting value for the sum.

Returns:	sum_along_axis : ndarray An array with the same shape as *a*, with the specified axis removed. If *a* is a 0-d array, or if *axis* is None, a scalar is returned. If an output array is specified, a reference to *out* is returned.

Random

numpy.random.randint(*low*, *high=None*, *size=None*, *dtype='l'*)

Return random integers from *low* (inclusive) to *high* (exclusive).

Return random integers from the “discrete uniform” distribution of the specified dtype in the “half-open” interval [*low*, *high*). If *high* is None (the default), then results are from [0, *low*).

Parameters:	low : int Lowest (signed) integer to be drawn from the distribution (unless high=None, in which case this parameter is one above the *highest* such integer). high : int, optional If provided, one above the largest (signed) integer to be drawn from the distribution (see above for behavior if high=None). size : int or tuple of ints, optional Output shape. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn. Default is None, in which case a single value is returned. dtype : dtype, optional Desired dtype of the result. All dtypes are determined by their name, i.e., ‘int64’, ‘int’, etc, so byteorder is not available and a specific precision may have different C types depending on the platform. The default value is ‘np.int’. New in version 1.11.0.

Returns: out : int or ndarray of ints
size-shaped array of random integers from the appropriate distribution, or a single such random int if *size* not provided.

See also
random.random_integers
similar to randint, only for the closed interval [*low*, *high*], and 1 is the lowest value if *high* is omitted. In particular, this other one is the one to use to generate uniformly distributed discrete non-integers.
Examples

```
>>>
>>> np.random.randint(2, size=10)
array([1, 0, 0, 0, 1, 1, 0, 0, 1, 0])
>>> np.random.randint(1, size=10)
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0])
```

Generate a 2 x 4 array of ints between 0 and 4, inclusive:

```
>>>
>>> np.random.randint(5, size=(2, 4))
array([[4, 0, 2, 1],
       [3, 2, 2, 0]])
```