# Python_oops

March 11, 2021

## 1 Object Oriented Programming

Object Oriented Programming (OOP) tends to be one of the major obstacles for beginners when they are first starting to learn Python.

There are many, many tutorials and lessons covering OOP so feel free to Google search other lessons, and I have also put some links to other useful tutorials online at the bottom of this Notebook.

For this lesson we will construct our knowledge of OOP in Python by building on the following topics:

- Objects
- Using the *class* keyword
- Creating class attributes
- Creating methods in a class
- Learning about Inheritance
- Learning about Polymorphism
- Learning about Special Methods for classes

---

What Is **Object-Oriented Programming** (OOP)?

Object-oriented Programming, or OOP for short, is a programming paradigm which provides a means of structuring programs so that properties and behaviors are bundled into individual objects.

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`

Lets start the lesson by remembering about the Basic Python Objects. For example:

[1]:
```
lst = [1,2,3]
```

Remember how we could call methods on a list?

[2]:
```
lst.count(2)
```

[2]:
```
1
```

### 1.0.1 Objects

An **object** is an instance of a class. A unique instance of a data structure that's defined by its class. An object comprises both data members (class variables and instance variables) and methods.

In Python, *everything is an object*. Remember from previous lectures we can use type() to check the type of object something is:

[3]:
```
print(type(1))
print(type([]))
print(type(()))
print(type({}))
```

```
<class 'int'>
<class 'list'>
<class 'tuple'>
<class 'dict'>
```

So we know all these things are objects, so how can we create our own Object types? That is where the class keyword comes in.

### 1.0.2 Class

**A class is a blueprint for the object**. User defined objects are created using the class keyword. Classes are used to create new user-defined data structures that contain arbitrary information about something. The class is a blueprint that defines the nature of a future object. From classes we can construct instances. An instance is a specific object created from a particular class. In the case of an animal, we could create an Animal() class to track properties about the Animal like the name and age. For example, above we created the object lst which was an instance of a list object.

It's important to note that a class just provides structure—it's a blueprint for how something should be defined, but it doesn't actually provide any real content itself. Let see how we can use class:

[4]:
```
# Create a new object type called Sample
class Sample:
    pass

# Instance of Sample
x = Sample()

print(type(x))
```

```
<class '__main__.Sample'>
```

By convention we give classes a name that starts with a capital letter. Note how x is now the reference to our new instance of a Sample class. In other words, we **instantiate** the Sample class.

Inside of the class we currently just have pass. But we can define class attributes and methods.

### 1.0.3 Constructors in Python

The **constructor** method is used to initialize data. It is run as soon as an object of a class is instantiated. Also known as the **init** method, it will be the first definition of a class.

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.

An **attribute** is a characteristic of an object. A **method** is an operation we can perform with the object.

For example, we can create a class called Dog. An attribute of a dog may be its breed or its name, while a method of a dog may be defined by a .bark() method which returns a sound.

Let's get a better understanding of attributes through an example. The syntax for creating an attribute is:

`self.attribute = something`

There is a special method called:

`__init__()`

This method is used to initialize the attributes of an object. For example:

[5]:
```
class Dog:
    def __init__(self,breed):
        self.breed = breed

sam = Dog(breed='Lab')
frank = Dog(breed='Huskie')
```

Lets break down what we have above.The special method

`__init__()`

is called automatically right after the object has been created:

`def __init__(self, breed):`

Each attribute in a class definition begins with a reference to the instance object. It is by convention named self. The breed is the argument. The value is passed during the class instantiation.

`self.breed = breed`

Now we have created two instances of the Dog class. With two breed types, we can then access these attributes like this:

[6]:
```
sam.breed
```

[6]:
```
'Lab'
```

[7]:
```
frank.breed
```

[7]:
```
'Huskie'
```

Note how we don't have any parentheses after breed; this is because it is an attribute and doesn't take any arguments.

In Python there are also *class object attributes*. These Class Object Attributes are the same for any instance of the class. For example, we could create the attribute *species* for the Dog class. Dogs, regardless of their breed, name, or other attributes, will always be mammals. We apply this logic in the following manner:

[8]:
```
class Dog:

    # Class Object Attribute
    species = 'mammal'

    def __init__(self,breed,name):
        self.breed = breed
        self.name = name
```

[9]:
```
sam = Dog('Lab','Sam')
```

[10]:
```
sam.name
```

[10]:
```
'Sam'
```

Note that the Class Object Attribute is defined outside of any methods in the class. Also by convention, we place them first before the init.

[11]:
```
sam.species
```

[11]:
```
'mammal'
```

### 1.0.4 Class Variables

Class variables are defined within the class construction. Because they are owned by the class itself, class variables are shared by all instances of the class. They therefore will generally have the same value for every instance unless you are using the class variable to initialize a variable.

Defined outside of all the methods, class variables are, by convention, typically placed right below the class header and before the constructor method and other methods.

[12]:
```
class Employee():

    raise_amount =1.04
```

```
    def __init__(self,first,last,pay):
        self.first  = first
        self.last = last
        self.pay = pay
        self.email = first+'.'+last+'@company.com'

    def fullname(self):
        return self.first+' '+self.last

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)
        return self.pay
```

[13]:
```
emp_1 = Employee('Sudhir','Kumar',200)
emp_2 = Employee('Latha','Shet',200)

print(emp_1.__dict__)
print(emp_2.fullname())
print(emp_1.apply_raise())
```

```
{'first': 'Sudhir', 'last': 'Kumar', 'pay': 200, 'email':
'Sudhir.Kumar@company.com'}
Latha Shet
208
```

### 1.0.5 Instance Variables

Instance variables are owned by instances of the class. This means that for each object or instance of a class, the instance variables are different.

Unlike class variables, instance variables are defined within methods.

[14]:
```
print(emp_1.first)
print(emp_1.email)
```

```
Sudhir
Sudhir.Kumar@company.com
```

## 1.1 Methods

Methods are functions defined inside the body of a class. They are used to perform operations with the attributes of our objects. Methods are a key concept of the OOP paradigm. They are essential to dividing responsibilities in programming, especially in large applications.

You can basically think of methods as functions acting on an Object that take the Object itself into account through its *self* argument.

Let's go through an example of creating a Circle class:

[15]:
```
class Circle:
    pi = 3.14

    # Circle gets instantiated with a radius (default is 1)
    def __init__(self, radius=1):
        self.radius = radius
        self.area = radius * radius * Circle.pi

    # Method for resetting Radius
    def setRadius(self, new_radius):
        self.radius = new_radius
        self.area = new_radius * new_radius * self.pi

    # Method for getting Circumference
    def getCircumference(self):
        return self.radius * self.pi * 2


c = Circle()

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  1
Area is:  3.14
Circumference is:  6.28
```

In the ___init___ method above, in order to calculate the area attribute, we had to call Circle.pi. This is because the object does not yet have its own .pi attribute, so we call the Class Object Attribute pi instead. In the setRadius method, however, we'll be working with an existing Circle object that does have its own pi attribute. Here we can use either Circle.pi or self.pi. Now let's change the radius and see how that affects our Circle object:

[16]:
```
c.setRadius(2)

print('Radius is: ',c.radius)
print('Area is: ',c.area)
print('Circumference is: ',c.getCircumference())
```

```
Radius is:  2
Area is:  12.56
Circumference is:  12.56
```

Great! Notice how we used self. notation to reference attributes of the class within the method calls. Review how the code above works and try creating your own method.

## 1.2 Inheritance

Inheritance is a way to form new classes using classes that have already been defined. The newly formed classes are called derived classes, the classes that we derive from are called base classes. Important benefits of inheritance are code reuse and reduction of complexity of a program. The derived classes (descendants) override or extend the functionality of base classes (ancestors).

Inheritance is when a class uses code constructed within another class. If we think of inheritance in terms of biology, we can think of a child inheriting certain traits from their parent. That is, a child can inherit a parent's height or eye color. Children also may share the same last name with their parents. Classes called child classes or subclasses inherit methods and variables from parent classes or base classes. Let's see an example by incorporating our previous work on the Dog class:

[17]:
```
class Animal:
    def __init__(self):
        print("Animal created")

    def whoAmI(self):
        print("Animal")

    def eat(self):
        print("Eating")


class Dog(Animal):
    def __init__(self):
        Animal.__init__(self)
        print("Dog created")

    def whoAmI(self):
        print("Dog")

    def bark(self):
        print("Woof!")
```

[18]:
```
d = Dog()
```

```
Animal created
Dog created
```

[19]:
```
d.whoAmI()
```

```
Dog
```

[20]:
```
d.eat()
```

```
Eating
```

[21]:
```
d.bark()
```

```
Woof!
```

In this example, we have two classes: Animal and Dog. The Animal is the base class, the Dog is the derived class.

The derived class inherits the functionality of the base class.

- It is shown by the eat() method.

The derived class modifies existing behavior of the base class.

- shown by the whoAmI() method.

Finally, the derived class extends the functionality of the base class, by defining a new bark() method.

The **benefits** of inheritance are:

1. It represents real-world relationships well.
2. It provides reusability of a code. We don't have to write the same code again and again. Also, it allows us to add more features to a class without modifying it.
3. It is transitive in nature, which means that if class B inherits from another class A, then all the subclasses of B would automatically inherit from class A.

### 1.3 Multiple Inheritance

The name says it all. One class extending more than one class is called multiple inheritance. This is one of the cool specialties of python which makes it more convenient than java in some cases (Java doesn't support multiple inheritance). Java doesn't have it because at times multiple inheritance may create some ambiguity.

### 1.4 Multiple Inheritance vs Multi-level Inheritance

It may seem confusing if you are familiar with multi-level inheritance before. The main difference between multiple and multi-level inheritance is that, in multi-level inheritance the superclass may also inherit another super class. And in this way, different levels of inheritance can be created among the classes.

When you inherit a child class from more than one base classes, that situation is known as Multiple Inheritance. It, however, exhibits the same behavior as does the single inheritance.

The syntax for Multiple Inheritance is also similar to the single inheritance. By the way, in Multiple Inheritance, the child class claims the properties and methods of all the parent classes.

https://www.python-course.eu/python3_multiple_inheritance.php

[22]:
```
# Parent class 1
class TeamMember(object):
    def __init__(self, name, uid):
        self.name = name
```

```
        self.uid = uid

# Parent class 2
class Worker(object):
    def __init__(self, pay, jobtitle):
        self.pay = pay
        self.jobtitle = jobtitle

# Deriving a child class from the two parent classes
class TeamLeader(TeamMember, Worker):
    def __init__(self, name, uid, pay, jobtitle, exp):
        self.exp = exp
        TeamMember.__init__(self, name, uid)
        Worker.__init__(self, pay, jobtitle)
        print("Name: {}, Pay: {}, Exp: {}".format(self.name, self.pay, self.
 ↪exp))

TL = TeamLeader('Jake', 10001, 250000, 'Scrum Master', 5)
```

```
Name: Jake, Pay: 250000, Exp: 5
```

### 1.4.1 Overriding Methods

When you define a parent class method in the child, then this process is called Overriding.

In other words, a child class can override methods of its parent or superclass by defining a function with the same name.

However, there are some rules for overriding:

- The name of the method should be the same and its parameters as well.
- If the superclass method is private (prefixed with double underscores), then you can't override it.

## 2 Reference

1. multiple-inheritance techbeamers
2. multiple-inheritance journaldev

### 2.0.1 Multi-level Inheritance

When you inherit a class from a derived class, then it's called multilevel inheritance. And, it can go up any levels in Python.

In multilevel inheritance, properties of the parent and the child classes are available to the new class.

Multilevel inheritance is akin to the relationship between grandpa, father, and the child.

[23]:
```
class parent:
    pass
class child(parent):
    pass
class next_child(child):
    pass
```

Python Multiple Inheritance vs. Multi-level Inheritance

The primary differences between Multiple and Multilevel Inheritance are as follows:

- Multiple Inheritance denotes a scenario when a class derives from more than one base classes.
- Multilevel Inheritance means a class derives from a subclass making that subclass a parent for the new class.
- Multiple Inheritance is more complex and hence not used widely.
- Multilevel Inheritance is a more typical case and hence used frequently.
- Multiple Inheritance has two classes in the hierarchy, i.e., a base class and its subclass.
- Multilevel Inheritance requires three levels of classes, i.e., a base class, an intermediate class, and the subclass.

### 2.0.2 Method Resolution Order

Method Resolution Order (MRO) is an approach that a programming language takes to resolve the variables or methods of a class.

Python has a built-in base class named as the object. So, any other in-built or user-defined class which you define will eventually inherit from it.

Now, let's talk about how the method resolution order (MRO) takes place in Python.

- In the multiple inheritance use case, the attribute is first looked up in the current class. If it fails, then the next place to search is in the parent class, and so on.
- If there are multiple parent classes, then the preference order is depth-first followed by a left-right path, i.e., DLR.
- MRO ensures that a class always precedes its parents and for multiple parents, keeps the order as the tuple of base classes.

[24]:
```
class Heap:
    def create(self):
        print(" Creating Heap")
class Node(Heap):
    def create(self):
        print(" Creating Node")

node = Node()
node.create()
```

```
 Creating Node
```

Methods for Method Resolution Order(MRO)

You can check the Method Resolution Order of a class. Python provides a **mro** attribute and the mro() method. With these, you can get the resolution order.

[25]:
```
Node.mro()
```

[25]:
```
[__main__.Node, __main__.Heap, object]
```

### 2.0.3 super() Function

At a high level super() gives you access to methods in a superclass from the subclass that inherits from it. super() alone returns a temporary object of the superclass that then allows you to call that superclass's methods. Calling the previously built methods with super() saves you from needing to rewrite those methods in your subclass, and allows you to swap out superclasses with minimal code changes.

[26]:
```
class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def area(self):
        return self.length * self.width

    def perimeter(self):
        return 2 * self.length + 2 * self.width

# Here we declare that the Square class inherits from the Rectangle class
class Square(Rectangle):
    def __init__(self, length):
        super().__init__(length, length)

square = Square(4)
square.area()
```

[26]:
```
16
```

### 2.0.4 Multiple Inheritance with argument

[27]:
```
# multiple inheritance with argument
class First:
    def  __init__(self,first,**kwargs):
        print('INIT of first ...')
        super().__init__(**kwargs)
        self.first = first
```

```
class Second:
    def __init__(self,second,**kwargs):
        print('INTI of second ...')
        super().__init__(**kwargs)
        self.second =second

class Combine(First,Second):
    def __init__(self,first,second,**kwargs):
        super().__init__(first=first,second=second)


c = Combine('Sudhir','Kumar')
```

```
INIT of first …
INTI of second …
```

## 2.1 Polymorphism

We've learned that while functions can take in different arguments, methods belong to the objects they act on. In Python, *polymorphism* refers to the way in which different object classes can share the same method name, and those methods can be called from the same place even though a variety of different objects might be passed in.

Polymorphism is an important feature of class definition in Python that is utilized when you have commonly named methods across classes or subclasses. This allows functions to use objects of any of these polymorphic classes without needing to be aware of distinctions across the classes.

Polymorphism can be carried out through inheritance, with subclasses making use of base class methods or overriding them. The best way to explain this is by example:

[28]:
```
class Dog:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Woof!'

class Cat:
    def __init__(self, name):
        self.name = name

    def speak(self):
        return self.name+' says Meow!'

niko = Dog('Niko')
felix = Cat('Felix')

print(niko.speak())
```

```
print(felix.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Here we have a Dog class and a Cat class, and each has a `.speak()` method. When called, each object's `.speak()` method returns a result unique to the object.

There a few different ways to demonstrate polymorphism. First, with a for loop:

[29]:
```
for pet in [niko,felix]:
    print(pet.speak())
```

```
Niko says Woof!
Felix says Meow!
```

Another is with functions:

[30]:
```
def pet_speak(pet):
    print(pet.speak())

pet_speak(niko)
pet_speak(felix)
```

```
Niko says Woof!
Felix says Meow!
```

In both cases we were able to pass in different object types, and we obtained object-specific results from the same mechanism.

A more common practice is to use abstract classes and inheritance. An abstract class is one that never expects to be instantiated. For example, we will never have an Animal object, only Dog and Cat objects, although Dogs and Cats are derived from Animals:

[31]:
```
class Animal:
    def __init__(self, name):    # Constructor of the class
        self.name = name

    def speak(self):              # Abstract method, defined by convention only
        raise NotImplementedError("Subclass must implement abstract method")


class Dog(Animal):

    def speak(self):
        return self.name+' says Woof!'

class Cat(Animal):

    def speak(self):
        return self.name+' says Meow!'
```

```
fido = Dog('Fido')
isis = Cat('Isis')

print(fido.speak())
print(isis.speak())
```

```
Fido says Woof!
Isis says Meow!
```

Real life examples of polymorphism include: * opening different file types - different tools are needed to display Word, pdf and Excel files * adding different objects - the + operator performs arithmetic and concatenation

## 2.2 Encapsulation of Data

Another important advantage of OOP consists in the encapsulation of data. We can say that object-oriented programming relies heavily on encapsulation. The terms encapsulation and abstraction (also data hiding) are often used as synonyms. They are nearly synonymous, i.e. abstraction is achieved though encapsulation. Data hiding and encapsulation are the same concept, so it's correct to use them as synonyms. Generally speaking encapsulation is the mechanism for restricting the access to some of an object's components, this means that the internal representation of an object can't be seen from outside of the objects definition. Access to this data is typically only achieved through special methods: Getters and Setters. By using solely get() and set() methods, we can make sure that the internal data cannot be accidentally set into an inconsistent or invalid state.

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 " ___".

[32]:
```
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()
```

```
Selling Price: 900
```

[33]:
```
# change the price
c.__maxprice = 1000
```

```
c.sell()
```

```
Selling Price: 900
```

[34]:
```
# using setter function
c.setMaxPrice(1000)
c.sell()
```

```
Selling Price: 1000
```

### 2.3 Special Methods

Finally let's go over special methods. Classes in Python can implement certain operations with special method names. These methods are not actually called directly but by Python specific language syntax. For example let's create a Book class:

[35]:
```
class Book:
    def __init__(self, title, author, pages):
        print("A book is created")
        self.title = title
        self.author = author
        self.pages = pages

    def __str__(self):
        return "Title: %s, author: %s, pages: %s" %(self.title, self.author,␣
↪self.pages)

    def __len__(self):
        return self.pages

    def __del__(self):
        print("A book is destroyed")
```

[36]:
```
book = Book("Python Rocks!", "Jose Portilla", 159)

#Special Methods
print(book)
print(len(book))
del book
```

```
A book is created
Title: Python Rocks!, author: Jose Portilla, pages: 159
159
A book is destroyed
```

The __init__(), __str__(), __len__() and __del__() methods

These special methods are defined by their use of underscores. They allow us to use Python specific functions on objects created through our class.

**Great! After this lecture you should have a basic understanding of how to create your own objects with class in Python. You will be utilizing this heavily in your next milestone project!**

For more great resources on this topic, check out:

Jeff Knupp's Post

Mozilla's Post

Tutorial's Point

Official Documentation

Special Method

### 2.3.1 class method

A class method is a method that is bound to a class rather than its object. It doesn't require creation of a class instance, much like staticmethod.

The difference between a static method and a class method is: * Static method knows nothing about the class and just deals with the parameters * Class method works with the class since its parameter is always the class itself.

The class method can be called both by the class and its object.

```
Class.classmethod()
Or even
Class().classmethod()
```

```
@classmethod
def set_raise_amount(cls, amount):
    cls.raise_amount = amount
```

[37]:
```
class Employee():

    raise_amount =1.04
    def __init__(self,first,last,pay):
        self.first  = first
        self.last = last
        self.pay = pay
        self.email = first+'.'+last+'@company.com'

    def fullname(self):
        return self.first+' '+self.last

    def apply_raise(self):
        self.pay = int(self.pay * self.raise_amount)
```

```
        return self.pay

    @classmethod
    def set_raise_amount(cls, amount):
        cls.raise_amount = amount

    @classmethod
    def from_string(cls,emp_str):
        " class method used as constructor"
        first,last,pay = emp_str.split('-')
        return cls(first,last,pay)

    @staticmethod
    def is_workday(day):
        if day.weekday() ==5 or day.weekday() ==6:
            return False
        return True
```

[38]:
```
emp_1 = Employee('Sudhir','Kumar',2000)
emp_2 = Employee('Latha','Shet',4000)

print(emp_1.raise_amount)
Employee.set_raise_amount(1.1)
print(emp_1.raise_amount)
```

```
1.04
1.1
```

class method can be used as alternative constructor. we use class method in order to provide multiple ways of creating our objects.

```
@classmethod
def from_string(cls,emp_str):
    " class method used as constructor"
    first,last,pay = emp_str.split('-')
    return cls(first,last,pay)
```

[39]:
```
# class method used as constructor
emp_str_1 = 'John-Doe-2000'
new_emp = Employee.from_string(emp_str_1)
print(new_emp.fullname())
```

```
John Doe
```

### 2.3.2 Static method

Static methods, much like class methods, are methods that are bound to a class rather than its object. They do not require a class instance creation. So, they are not dependent on the state of

the object.

The difference between a static method and a class method is: * Static method knows nothing about the class and just deals with the parameters. * Class method works with the class since its parameter is always the class itself.

They can be called both by the class and its object.

```
Class.staticmethodFunc()
or even
Class().staticmethodFunc()

@staticmethod
def is_workday(day):
    if day.weekday() ==5 or day.weekday() ==6:
        return False
    return True
```

[40]:
```
import datetime
my_day = datetime.date(2018,11,21)
Employee.is_workday(my_day)
```

[40]:
```
True
```

### 2.3.3 Operator Overloading

Python operators work for built-in classes. But the same operator behaves differently with different types. For example, the + operator will perform arithmetic addition on two numbers, merge two lists, or concatenate two strings.

This feature in Python that allows the same operator to have different meaning according to the context is called operator overloading.

[41]:
```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

try:

    p1 = Point(1, 2)
    p2 = Point(2, 3)
    print(p1+p2)
except:
    print('unsupported operand type(s) for +')
```

```
unsupported operand type(s) for +
```

[42]:
```
class Point:
    def __init__(self, x=0, y=0):
        self.x = x
        self.y = y

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

    def __str__(self):
        return f"Point({self.x},{self.y})"

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

```
Point(3,5)
```

[43]:
```
# second example
class Employee():

    def __init__(self,first,last,pay):
        self.first  = first
        self.last = last
        self.pay = pay

    def __add__(self,other):
        pay = self.pay + other.pay
        return pay
```

[44]:
```
emp_1 = Employee('sudhi','kumar',20000)
emp_2 = Employee('test','user',10200)
```

[45]:
```
emp_1 + emp_2
```

[45]:
```
30200
```

### 2.3.4 @property decorator

Python programming provides us with a built-in @property decorator which makes usage of getter and setters much easier in Object-Oriented Programming.

- Getters: These are the methods used in Object-Oriented Programming (OOPS) which helps to access the private attributes from a class.

- Setters: These are the methods used in OOPS feature which helps to set the value to private attributes in a class.

[46]:
```
#Class Without Getters and Setters
class Celsius:
    def __init__(self,temperature=0):
        self.temperature = temperature
    def to_fahrenheit(self):
        return (self.temperature *1.8) + 32

#
human = Celsius()
human.temperature = 37
print(human.temperature)
print(human.to_fahrenheit())
```

```
37
98.60000000000001
```

[47]:
```
class Celsius:
    def __init__(self,temperature=0):
        self.set_temperature(temperature)

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

    def get_temperature(self):
        return self._temperature

    def set_temperature(self,value):
        if value < -273.15:
            raise ValueError('Temperature below -273.15 is not possible')
        self._temperature = value
```

[48]:
```
human = Celsius(37)
print(human.get_temperature())
print(human.to_fahrenheit())
try:
    human.set_temperature(-300)
except:
    print('error ----------------')

# Get the to_fahreheit method
print(human.to_fahrenheit())
```

```
37
98.60000000000001
error ----------------
```

```
98.60000000000001
```

However, the bigger problem with the above update is that all the programs that implemented our previous class have to modify their code from obj.temperature to obj.get_temperature() and all expressions like obj.temperature = val to obj.set_temperature(val).

This refactoring can cause problems while dealing with hundreds of thousands of lines of codes.

All in all, our new update was not backwards compatible. This is where @property comes to rescue.

The property Class

A pythonic way to deal with the above problem is to use the property class. Here is how we can update our code:

[49]:
```
# using property class
class Celsius:
    def __init__(self, temperature=0):
        self.temperature = temperature

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

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

    # setter
    def set_temperature(self, value):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273.15 is not possible")
        self._temperature = value

    # creating a property object
    temperature = property(get_temperature, set_temperature)
```

[50]:
```
human.get_temperature()
```

[50]:
```
37
```

[51]:
```
human = Celsius(37)
print(human.temperature)
print(human.to_fahrenheit())
try:
    human.temperature = -300
except:
    print('error -----')
```

```
Setting value…
Getting value…
37
Getting value…
98.60000000000001
Setting value…
error -----
```

The @property Decorator

In Python, property() is a built-in function that creates and returns a property object. The syntax 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
- doc is a string (like a comment)

[53]:
```python
# Using @property decorator
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):
        print("Setting value...")
        if value < -273.15:
            raise ValueError("Temperature below -273 is not possible")
        self._temperature = value


# create an object
human = Celsius(37)

print(human.temperature)

print(human.to_fahrenheit())
```

```
try:
    coldest_thing = Celsius(-300)
except:
    print('error')
```

```
Setting value…
Getting value…
37
Getting value…
98.60000000000001
Setting value…
error
```

### 2.3.5 if name == 'main'

stackoverflow

Whenever the Python interpreter reads a source file, it does two things:

- it sets a few special variables like **name**, and then
- it executes all of the code found in the file.

Let's see how this works and how it relates to your question about the **name** checks we always see in Python scripts.

## 3 Code Sample

Let's use a slightly different code sample to explore how imports and scripts work. Suppose the following is in a file called foo.py.

```
# Suppose this is foo.py.

print("before import")
import math

print("before functionA")
def functionA():
    print("Function A")

print("before functionB")
def functionB():
    print("Function B {}".format(math.sqrt(100)))

print("before __name__ guard")
if __name__ == '__main__':
    functionA()
```

```
    functionB()
print("after __name__ guard")
```

### 3.0.1 Special Variables

When the Python interpeter reads a source file, it first defines a few special variables. In this case, we care about the **name** variable.

## 3.1 When Your Module Is the Main Program

If you are running your module (the source file) as the main program, e.g.

```
python foo.py
```

the interpreter will assign the hard-coded string "**main**" to the **name** variable, i.e.

```
# It's as if the interpreter inserts this at the top
# of your module when run as the main program.
__name__ = "__main__"
```

### 3.1.1 When Your Module Is Imported By Another

On the other hand, suppose some other module is the main program and it imports your module. This means there's a statement like this in the main program, or in some other module the main program imports:

```
# Suppose this is in some other main program.
import foo
```

The interpreter will search for your foo.py file (along with searching for a few other variants), and prior to executing that module, it will assign the name "foo" from the import statement to the **name** variable, i.e.

```
# It's as if the interpreter inserts this at the top
# of your module when it's imported from another module.
__name__ = "foo"
```

### 3.1.2 Executing the Module's Code

After the special variables are set up, the interpreter executes all the code in the module, one statement at a time. You may want to open another window on the side with the code sample so you can follow along with this explanation.

Always

1. It prints the string "before import" (without quotes).
2. It loads the math module and assigns it to a variable called math. This is equivalent to replacing import math with the following (note that **import** is a low-level function in Python that takes a string and triggers the actual import):

```
# Find and load a module given its string name, "math",
# then assign it to a local variable called math.
math = __import__("math")
```

3. It prints the string "before functionA".
4. It executes the def block, creating a function object, then assigning that function object to a variable called functionA.
5. It prints the string "before functionB".
6. It executes the second def block, creating another function object, then assigning it to a variable called functionB.
7. It prints the string "before **name** guard".

Only When Your Module Is the Main Program

8. If your module is the main program, then it will see that **name** was indeed set to "**main**" and it calls the two functions, printing the strings "Function A" and "Function B 10.0".

Only When Your Module Is Imported by Another

8. (instead) If your module is not the main program but was imported by another one, then **name** will be "foo", not "**main**", and it'll skip the body of the if statement.

Always

9. It will print the string "after **name** guard" in both situations.

### 3.1.3 Why Does It Work This Way?

You might naturally wonder why anybody would want this. Well, sometimes you want to write a .py file that can be both used by other programs and/or modules as a module, and can also be run as the main program itself. Examples:

- Your module is a library, but you want to have a script mode where it runs some unit tests or a demo.
- Your module is only used as a main program, but it has some unit tests, and the testing framework works by importing .py files like your script and running special test functions. You don't want it to try running the script just because it's importing the module.
- Your module is mostly used as a main program, but it also provides a programmer-friendly API for advanced users.

Beyond those examples, it's elegant that running a script in Python is just setting up a few magic variables and importing the script. "Running" the script is a side effect of importing the script's module.

[54]:
```
# Suppose this is foo.py.

print("before import")
import math

print("before functionA")
def functionA():
```

```
    print("Function A")

print("before functionB")
def functionB():
    print("Function B {}".format(math.sqrt(100)))

print("before __name__ guard")
if __name__ == '__main__':
    functionA()
    functionB()
print("after __name__ guard")
```

```
before import
before functionA
before functionB
before __name__ guard
Function A
Function B 10.0
after __name__ guard
```

## 3.2 Modules and Packages

In this section we briefly: * code out a basic module and show how to import it into a Python script * run a Python script from a Jupyter cell * show how command line arguments can be passed into a script

Check out the video lectures for more info and resources for this.

The best online resource is the official docs: https://docs.python.org/3/tutorial/modules.html#packages

But I really like the info here: https://python4astronomers.github.io/installation/packages.html

### 3.2.1 Writing modules

[55]:
```
%%writefile file1.py
def myfunc(x):
    return [num for num in range(x) if num%2==0]
list1 = myfunc(11)
```

```
Overwriting file1.py
```

**file1.py** is going to be used as a module.

Note that it doesn't print or return anything, it just defines a function called *myfunc* and a variable called *list1*. ### Writing scripts

[56]:
```
%%writefile file2.py
import file1
```

```
file1.list1.append(12)
print(file1.list1)
```

```
Overwriting file2.py
```

**file2.py** is a Python script.

First, we import our **file1** module (note the lack of a .py extension) Next, we access the *list1* variable inside **file1**, and perform a list method on it. `.append(12)` proves we're working with a Python list object, and not just a string. Finally, we tell our script to print the modified list. ### Running scripts

[57]:
```
! python file2.py
```

```
[0, 2, 4, 6, 8, 10, 12]
```

Here we run our script from the command line. The exclamation point is a Jupyter trick that lets you run command line statements from inside a jupyter cell.

[58]:
```
import file1
print(file1.list1)
```

```
[0, 2, 4, 6, 8, 10]
```

The above cell proves that we never altered **file1.py**, we just appended a number to the list *after* it was brought into **file2**.

### 3.2.2 Passing command line arguments

Python's `sys` module gives you access to command line arguments when calling scripts.

[59]:
```
%%writefile file3.py
import sys
import file1
num = int(sys.argv[1])
print(file1.myfunc(num))
```

```
Overwriting file3.py
```

Note that we selected the second item in the list of arguments with `sys.argv[1]`. This is because the list created with `sys.argv` always starts with the name of the file being used.

[60]:
```
! python file3.py 21
```

```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20]
```

Here we're passing 21 to be the upper range value used by the *myfunc* function in **list1.py**

### 3.2.3 Understanding modules

Modules in Python are simply Python files with the .py extension, which implement a set of functions. Modules are imported from other modules using the import command.

To import a module, we use the import command. Check out the full list of built-in modules in the Python standard library here.

The first time a module is loaded into a running Python script, it is initialized by executing the code in the module once. If another module in your code imports the same module again, it will not be loaded twice but once only - so local variables inside the module act as a "singleton" - they are initialized only once.

If we want to import the math module, we simply import the name of the module:

[61]:
```
# import the library
import math
```

[62]:
```
# use it (ceiling rounding)
math.ceil(2.4)
```

[62]:
```
3
```

### 3.2.4 Exploring built-in modules

Two very important functions come in handy when exploring modules in Python - the dir and help functions.

We can look for which functions are implemented in each module by using the dir function:

[63]:
```
print(dir(math))
```

```
['__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__',
'acos', 'acosh', 'asin', 'asinh', 'atan', 'atan2', 'atanh', 'ceil', 'comb',
'copysign', 'cos', 'cosh', 'degrees', 'dist', 'e', 'erf', 'erfc', 'exp',
'expm1', 'fabs', 'factorial', 'floor', 'fmod', 'frexp', 'fsum', 'gamma', 'gcd',
'hypot', 'inf', 'isclose', 'isfinite', 'isinf', 'isnan', 'isqrt', 'ldexp',
'lgamma', 'log', 'log10', 'log1p', 'log2', 'modf', 'nan', 'perm', 'pi', 'pow',
'prod', 'radians', 'remainder', 'sin', 'sinh', 'sqrt', 'tan', 'tanh', 'tau',
'trunc']
```

When we find the function in the module we want to use, we can read about it more using the help function, inside the Python interpreter:

[64]:
```
help(math.ceil)
```

```
Help on built-in function ceil in module math:

ceil(x, /)
    Return the ceiling of x as an Integral.
```

```
This is the smallest integer >= x.
```

### 3.2.5 Writing modules

Writing Python modules is very simple. To create a module of your own, simply create a new .py file with the module name, and then import it using the Python file name (without the .py extension) using the import command.

### 3.2.6 Writing packages

Packages are name-spaces which contain multiple packages and modules themselves. They are simply directories, but with a twist.

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

If we create a directory called foo, which marks the package name, we can then create a module inside that package called bar. We also must not forget to add the ***__init__*.py** file inside the foo directory.

To use the module bar, we can import it in two ways:

[65]:
```
ls
```

```
file1.py  file3.py  newmyfile.txt  Python_basics.ipynb  testfile
file2.py  image/    __pycache__/   Python_oops.ipynb
test.txt
```

```
Just an example, this won't work
import foo.bar

# OR could do it this way
from foo import bar
```

In the first method, we must use the foo prefix whenever we access the module bar. In the second method, we don't, because we import the module to our module's name-space.

The ***__init__*.py** file can also decide which modules the package exports as the API, while keeping other modules internal, by overriding the ***__all__*** variable, like so:

```
__init__.py:

__all__ = ["bar"]
```

## 3.3 Errors and Exception Handling

In this lecture we will learn about Errors and Exception Handling in Python. You've definitely already encountered errors by this point in the course. For example:

[66]:
```
print('Hello')
```

```
Hello
```

Note how we get a SyntaxError, with the further description that it was an EOL (End of Line Error) while scanning the string literal. This is specific enough for us to see that we forgot a single quote at the end of the line. Understanding these various error types will help you debug your code much faster.

This type of error and description is known as an Exception. Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called exceptions and are not unconditionally fatal.

You can check out the full list of built-in exceptions here. Now let's learn how to handle errors and exceptions in our own code.

### 3.3.1 try and except

The basic terminology and syntax used to handle errors in Python are the try and except statements. The code which can cause an exception to occur is put in the try block and the handling of the exception is then implemented in the except block of code. The syntax follows:

```
try:
   You do your operations here...
   ...
except ExceptionI:
   If there is ExceptionI, then execute this block.
except ExceptionII:
   If there is ExceptionII, then execute this block.
   ...
else:
   If there is no exception then execute this block.
```

We can also just check for any exception with just using except: To get a better understanding of all this let's check out an example: We will look at some code that opens and writes a file:

[67]:
```
try:
    f = open('testfile','w')
    f.write('Test write this')
except IOError:
    # This will only check for an IOError exception and then execute this print␣
 ↪statement
    print("Error: Could not find file or read data")
else:
    print("Content written successfully")
```

```
    f.close()
```

```
Content written successfully
```

Now let's see what would happen if we did not have write permission (opening only with 'r'):

[68]:
```
try:
    f = open('testfile','r')
    f.write('Test write this')
except IOError:
    # This will only check for an IOError exception and then execute this print␣
 ↪statement
    print("Error: Could not find file or read data")
else:
    print("Content written successfully")
    f.close()
```

```
Error: Could not find file or read data
```

Great! Notice how we only printed a statement! The code still ran and we were able to continue doing actions and running code blocks. This is extremely useful when you have to account for possible input errors in your code. You can be prepared for the error and keep running code, instead of your code just breaking as we saw above.

We could have also just said except: if we weren't sure what exception would occur. For example:

[69]:
```
try:
    f = open('testfile','r')
    f.write('Test write this')
except:
    # This will check for any exception and then execute this print statement
    print("Error: Could not find file or read data")
else:
    print("Content written successfully")
    f.close()
```

```
Error: Could not find file or read data
```

Great! Now we don't actually need to memorize that list of exception types! Now what if we kept wanting to run code after the exception occurred? This is where finally comes in. ### finally The finally: block of code will always be run regardless if there was an exception in the try code block. The syntax is:

```
try:
   Code block here
   ...
   Due to any exception, this code may be skipped!
finally:
   This code block would always be executed.
```

For example:

[70]:
```
try:
    f = open("testfile", "w")
    f.write("Test write statement")
    f.close()
finally:
    print("Always execute finally code blocks")
```

```
Always execute finally code blocks
```

We can use this in conjunction with except. Let's see a new example that will take into account a user providing the wrong input:

[71]:
```
def askint():
    try:
        val = int(input("Please enter an integer: "))
    except:
        print("Looks like you did not enter an integer!")

    finally:
        print("Finally, I executed!")
    print(val)
```

[72]:
```
askint()
```

```
Please enter an integer:  32

Finally, I executed!
32
```

[74]:
```
askint()
```

```
Please enter an integer:  3

Finally, I executed!
3
```

Notice how we got an error when trying to print val (because it was never properly assigned). Let's remedy this by asking the user and checking to make sure the input type is an integer:

[75]:
```
def askint():
    try:
        val = int(input("Please enter an integer: "))
    except:
        print("Looks like you did not enter an integer!")
        val = int(input("Try again-Please enter an integer: "))
    finally:
        print("Finally, I executed!")
    print(val)
```

[76]:
```
askint()
```

```
Please enter an integer:  r

Looks like you did not enter an integer!

Try again-Please enter an integer:  57

Finally, I executed!
57
```

Hmmm…that only did one check. How can we continually keep checking? We can use a while loop!

[77]:
```
def askint():
    while True:
        try:
            val = int(input("Please enter an integer: "))
        except:
            print("Looks like you did not enter an integer!")
            continue
        else:
            print("Yep that's an integer!")
            break
        finally:
            print("Finally, I executed!")
        print(val)
```

[78]:
```
askint()
```

```
Please enter an integer:  2

Yep that's an integer!
Finally, I executed!
```

So why did our function print "Finally, I executed!" after each trial, yet it never printed `val` itself? This is because with a try/except/finally clause, any continue or break statements are reserved until *after* the try clause is completed. This means that even though a successful input of **3** brought us to the else: block, and a break statement was thrown, the try clause continued through to finally: before breaking out of the while loop. And since print(val) was outside the try clause, the break statement prevented it from running.

Let's make one final adjustment:

[79]:
```
def askint():
    while True:
        try:
            val = int(input("Please enter an integer: "))
        except:
            print("Looks like you did not enter an integer!")
            continue
        else:
            print("Yep that's an integer!")
            print(val)
```

```
            break
        finally:
            print("Finally, I executed!")
```

[80]:
```
askint()
```

```
Please enter an integer:  5

Yep that's an integer!
5
Finally, I executed!
```

**Great! Now you know how to handle errors and exceptions in Python with the try, except, else, and finally notation!**

## 3.4 Built in function

### 3.4.1 map()

map() is a built-in Python function that takes in two or more arguments: a function and one or more iterables, in the form:

`map(function, iterable, ...)`

map() returns an *iterator* - that is, map() returns a special object that yields one result at a time as needed. We will learn more about iterators and generators in a future lecture. For now, since our examples are so small, we will cast map() as a list to see the results immediately.

When we went over list comprehensions we created a small expression to convert Celsius to Fahrenheit. Let's do the same here but use map:

[81]:
```
def fahrenheit(celsius):
    return (9/5)*celsius + 32

temps = [0, 22.5, 40, 100]
```

Now let's see map() in action:

[82]:
```
F_temps = map(fahrenheit, temps)

#Show
list(F_temps)
```

[82]:
```
[32.0, 72.5, 104.0, 212.0]
```

### 3.4.2 map() with multiple iterables

map() can accept more than one iterable. The iterables should be the same length - in the event that they are not, map() will stop as soon as the shortest iterable is exhausted.

For instance, if our function is trying to add two values **x** and **y**, we can pass a list of **x** values and another list of **y** values to map(). The function (or lambda) will be fed the 0th index from each list, and then the 1st index, and so on until the n-th index is reached.

Let's see this in action with two and then three lists:

[83]:
```
a = [1,2,3,4]
b = [5,6,7,8]
c = [9,10,11,12]

list(map(lambda x,y:x+y,a,b))
```

[83]:
```
[6, 8, 10, 12]
```

[84]:
```
# Now all three lists
list(map(lambda x,y,z:x+y+z,a,b,c))
```

[84]:
```
[15, 18, 21, 24]
```

We can see in the example above that the parameter **x** gets its values from the list **a**, while **y** gets its values from **b** and **z** from list **c**. Go ahead and play with your own example to make sure you fully understand mapping to more than one iterable.

Great job! You should now have a basic understanding of the map() function.

### 3.4.3 reduce()

Many times students have difficulty understanding reduce() so pay careful attention to this lecture. The function reduce(function, sequence) continually applies the function to the sequence. It then returns a single value.

If seq = [ s1, s2, s3, … , sn ], calling reduce(function, sequence) works like this:

- At first the first two elements of seq will be applied to function, i.e. func(s1,s2)
- The list on which reduce() works looks now like this: [ function(s1, s2), s3, … , sn ]
- In the next step the function will be applied on the previous result and the third element of the list, i.e. function(function(s1, s2),s3)
- The list looks like this now: [ function(function(s1, s2),s3), … , sn ]
- It continues like this until just one element is left and return this element as the result of reduce()

Let's see an example:

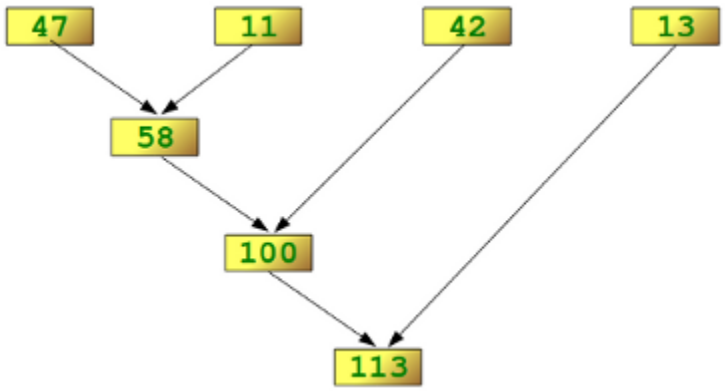
[85]:
```
from functools import reduce

lst =[47,11,42,13]
reduce(lambda x,y: x+y,lst)
```

[85]:
```
113
```

Lets look at a diagram to get a better understanding of what is going on here:

[86]:
```
from IPython.display import Image
Image('http://www.python-course.eu/images/reduce_diagram.png')
```

[86]:




Note how we keep reducing the sequence until a single final value is obtained. Lets see another example:

[87]:
```
#Find the maximum of a sequence (This already exists as max())
max_find = lambda a,b: a if (a > b) else b
```

[88]:
```
#Find max
reduce(max_find,lst)
```

[88]: 47

Hopefully you can see how useful reduce can be in various situations. Keep it in mind as you think about your code projects!

### 3.4.4 filter

The function filter(function, list) offers a convenient way to filter out all the elements of an iterable, for which the function returns True.

The function filter(function,list) needs a function as its first argument. The function needs to return a Boolean value (either True or False). This function will be applied to every element of the iterable. Only if the function returns True will the element of the iterable be included in the result.

Like map(), filter() returns an *iterator* - that is, filter yields one result at a time as needed. Iterators and generators will be covered in an upcoming lecture. For now, since our examples are so small, we will cast filter() as a list to see our results immediately.

Let's see some examples:

[89]:
```
#First let's make a function
def even_check(num):
    if num%2 ==0:
        return True
```

Now let's filter a list of numbers. Note: putting the function into filter without any parentheses might feel strange, but keep in mind that functions are objects as well.

[90]:
```
lst =range(20)

list(filter(even_check,lst))
```

[90]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

filter() is more commonly used with lambda functions, because we usually use filter for a quick job where we don't want to write an entire function. Let's repeat the example above using a lambda expression:

[91]:
```
list(filter(lambda x: x%2==0,lst))
```

[91]:
```
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

### 3.4.5 zip

zip() makes an iterator that aggregates elements from each of the iterables.

Returns an iterator of tuples, where the i-th tuple contains the i-th element from each of the argument sequences or iterables. The iterator stops when the shortest input iterable is exhausted. With a single iterable argument, it returns an iterator of 1-tuples. With no arguments, it returns an empty iterator.

zip() is equivalent to:

```
def zip(*iterables):
    # zip('ABCD', 'xy') --> Ax By
    sentinel = object()
    iterators = [iter(it) for it in iterables]
    while iterators:
        result = []
        for it in iterators:
            elem = next(it, sentinel)
            if elem is sentinel:
                return
            result.append(elem)
        yield tuple(result)
```

zip() should only be used with unequal length inputs when you don't care about trailing, unmatched values from the longer iterables.

Let's see it in action in some examples:

[92]:
```
x = [1,2,3]
y = [4,5,6]

# Zip the lists together
list(zip(x,y))
```

[92]:
```
[(1, 4), (2, 5), (3, 6)]
```

### 3.4.6 enumerate()

In this lecture we will learn about an extremely useful built-in function: enumerate(). Enumerate allows you to keep a count as you iterate through an object. It does this by returning a tuple in the form (count,element). The function itself is equivalent to:

```
def enumerate(sequence, start=0):
    n = start
    for elem in sequence:
        yield n, elem
        n += 1
```

enumerate() becomes particularly useful when you have a case where you need to have some sort of tracker. For example:

[93]:
```
lst = ['a','b','c']
for count,item in enumerate(lst):
    if count >= 2:
        break
    else:
        print(item)
```

```
a
b
```

enumerate() takes an optional "start" argument to override the default value of zero:

[94]:
```
months = ['March','April','May','June']

list(enumerate(months,start=3))
```

[94]:
```
[(3, 'March'), (4, 'April'), (5, 'May'), (6, 'June')]
```

### 3.4.7 all() and any()

all() and any() are built-in functions in Python that allow us to conveniently check for boolean matching in an iterable. all() will return True if all elements in an iterable are True. It is the same as this function code:

```
def all(iterable):
    for element in iterable:
        if not element:
            return False
    return True
```

any() will return True if any of the elements in the iterable are True. It is equivalent to the following function code:

```
def any(iterable):
    for element in iterable:
        if element:
            return True
    return False
```

Let's see a few examples of these functions. They should be fairly straightforward:

[95]:
```
lst = [True,True,False,True]
```

[96]:
```
all(lst)
```

[96]:
```
False
```

Returns False because not all elements are True.

[97]:
```
any(lst)
```

[97]:
```
True
```

Returns True because at least one of the elements in the list is True

### 3.4.8 complex()

complex() returns a complex number with the value real + imag*1j or converts a string or number to a complex number.

If the first parameter is a string, it will be interpreted as a complex number and the function must be called without a second parameter. The second parameter can never be a string. Each argument may be any numeric type (including complex). If imag is omitted, it defaults to zero and the constructor serves as a numeric conversion like int and float. If both arguments are omitted, returns 0j.

If you are doing math or engineering that requires complex numbers (such as dynamics, control systems, or impedance of a circuit) this is a useful tool to have in Python.

Let's see some examples:

[98]:
```
# Create 2+3j
complex(2,3)
```

[98]:
```
(2+3j)
```

We can also pass strings:

[99]:
```
complex('12+2j')
```

[99]:
```
(12+2j)
```

# 4 Decorators

Decorators are very powerful and useful tool in Python since it allows programmers to modify the behavior of function,methods or class. Decorators allow us to wrap another function in order to extend the behavior of wrapped function, without permanently modifying it.

Decorators can be thought of as functions which modify the *functionality* of another function. They help to make your code shorter and more "Pythonic".

In Python, functions are the first class objects, which means that –

- Functions are objects; they can be referenced to, passed to a variable and returned from other functions as well.
- Functions can be defined inside another function and can also be passed as argument to another function.

To properly explain decorators we will slowly build up from functions. Make sure to run every cell in this Notebook for this lecture to look the same on your own computer.So let's break down the steps:

### 4.0.1 Functions Review

[100]:
```
def func():
    return 1
func()
```

[100]:
```
1
```

**Scope Review**

Remember from the nested statements lecture that Python uses Scope to know what a label is referring to. For example:

[101]:
```
s = 'Global Variable'

def check_for_locals():
    print(locals())
```

Remember that Python functions create a new scope, meaning the function has its own namespace to find variable names when they are mentioned within the function. We can check for local variables and global variables with the locals() and globals() functions. For example:

[102]:
```
print(globals()['__name__'],globals()['__doc__'])
```

```
__main__ Automatically created module for IPython interactive environment
```

Here we get back a dictionary of all the global variables, many of them are predefined in Python. So let's go ahead and look at the keys:

[103]:
```
print(globals().keys())
```

```
dict_keys(['__name__', '__doc__', '__package__', '__loader__', '__spec__',
'__builtin__', '__builtins__', '_ih', '_oh', '_dh', 'In', 'Out', 'get_ipython',
'exit', 'quit', '_', '__', '___', '_i', '_ii', '_iii', '_i1', 'lst', '_i2',
'_2', '_i3', '_i4', 'Sample', 'x', '_i5', 'Dog', 'sam', 'frank', '_i6', '_6',
'_i7', '_7', '_i8', '_i9', '_i10', '_10', '_i11', '_11', '_i12', 'Employee',
'_i13', 'emp_1', 'emp_2', '_i14', '_i15', 'Circle', 'c', '_i16', '_i17',
'Animal', '_i18', 'd', '_i19', '_i20', '_i21', '_i22', 'TeamMember', 'Worker',
'TeamLeader', 'TL', '_i23', 'parent', 'child', 'next_child', '_i24', 'Heap',
'Node', 'node', '_i25', '_25', '_i26', 'Rectangle', 'Square', 'square', '_26',
'_i27', 'First', 'Second', 'Combine', '_i28', 'Cat', 'niko', 'felix', '_i29',
'pet', '_i30', 'pet_speak', '_i31', 'fido', 'isis', '_i32', 'Computer', '_i33',
'_i34', '_i35', 'Book', '_i36', '_i37', '_i38', '_i39', 'emp_str_1', 'new_emp',
'_i40', 'datetime', 'my_day', '_40', '_i41', 'Point', 'p1', 'p2', '_i42',
'_i43', '_i44', '_i45', '_45', '_i46', 'Celsius', 'human', '_i47', '_i48',
'_i49', '_i50', '_50', '_i51', '_i52', '_i53', '_i54', 'math', 'functionA',
'functionB', '_i55', '_i56', '_i57', '_exit_code', '_i58', 'file1', '_i59',
'_i60', '_i61', '_i62', '_62', '_i63', '_i64', '_i65', '_i66', '_i67', 'f',
'_i68', '_i69', '_i70', '_i71', 'askint', '_i72', '_i73', '_i74', '_i75',
'_i76', '_i77', '_i78', '_i79', '_i80', '_i81', 'fahrenheit', 'temps', '_i82',
'F_temps', '_82', '_i83', 'a', 'b', '_83', '_i84', '_84', '_i85', 'reduce',
'_85', '_i86', 'Image', '_86', '_i87', 'max_find', '_i88', '_88', '_i89',
'even_check', '_i90', '_90', '_i91', '_91', '_i92', 'y', '_92', '_i93', 'count',
'item', '_i94', 'months', '_94', '_i95', '_i96', '_96', '_i97', '_97', '_i98',
'_98', '_i99', '_99', '_i100', 'func', '_100', '_i101', 's', 'check_for_locals',
'_i102', '_i103'])
```

Note how **s** is there, the Global Variable we defined as a string:

[104]:
```
globals()['s']
```

[104]:
```
'Global Variable'
```

Now let's run our function to check for local variables that might exist inside our function (there shouldn't be any)

[105]:
```
check_for_locals()
```

```
{}
```

Great! Now lets continue with building out the logic of what a decorator is. Remember that in Python **everything is an object**. That means functions are objects which can be assigned labels

and passed into other functions. Lets start with some simple examples:

[106]:
```
def hello(name='Jose'):
    return 'Hello '+name

hello()
```

[106]:
```
'Hello Jose'
```

Assign another label to the function. Note that we are not using parentheses here because we are not calling the function **hello**, instead we are just passing a function object to the **greet** variable.

[107]:
```
greet = hello

greet
```

[107]:
```
<function __main__.hello(name='Jose')>
```

[108]:
```
greet()
```

[108]:
```
'Hello Jose'
```

So what happens when we delete the name **hello**?

[109]:
```
del hello
```

[110]:
```
try:
    hello()
except:
    print('not defind')
```

```
not defind
```

[111]:
```
greet()
```

[111]:
```
'Hello Jose'
```

Even though we deleted the name **hello**, the name **greet** *still points to* our original function object. It is important to know that functions are objects that can be passed to other objects!

## 4.1 Creating a Decorator

In the previous example we actually manually created a Decorator. Here we will modify it to make its use case clear:

[112]:
```
def new_decorator(func):

    def wrap_func():
```

```
        print("Code would be here, before executing the func")

        func()

        print("Code here will execute after the func()")

    return wrap_func

def func_needs_decorator():
    print("This function is in need of a Decorator")
```

[113]:
```
func_needs_decorator()
```

```
This function is in need of a Decorator
```

## 5 Reassign func_needs_decorator

func_needs_decorator = new_decorator(func_needs_decorator)

[114]:
```
func_needs_decorator()
```

```
This function is in need of a Decorator
```

So what just happened here? A decorator simply wrapped the function and modified its behavior. Now let's understand how we can rewrite this code using the @ symbol, which is what Python uses for Decorators:

[115]:
```
@new_decorator
def func_needs_decorator():
    print("This function is in need of a Decorator")
```

[116]:
```
func_needs_decorator()
```

```
Code would be here, before executing the func
This function is in need of a Decorator
Code here will execute after the func()
```

**Great! You've now built a Decorator manually and then saw how we can use the @ symbol in Python to automate this and clean our code. You'll run into Decorators a lot if you begin using Python for Web Development, such as Flask or Django!**

[117]:
```
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
```

[118]:
```
divide(2,5)
```

```
I am going to divide 2 and 5
```

[118]: 0.4

[119]:
```
divide(2,0)
```

```
I am going to divide 2 and 0
Whoops! cannot divide
```

## 5.1 Iterators and Generators

In this section of the course we will be learning the difference between iteration and generation in Python and how to construct our own Generators with the *yield* statement. Generators allow us to generate as we go along, instead of holding everything in memory.

We've touched on this topic in the past when discussing certain built-in Python functions like **range()**, **map()** and **filter()**.

Let's explore a little deeper. We've learned how to create functions with def and the return statement. Generator functions allow us to write a function that can send back a value and then later resume to pick up where it left off. This type of function is a generator in Python, allowing us to generate a sequence of values over time. The main difference in syntax will be the use of a yield statement.

In most aspects, a generator function will appear very similar to a normal function. The main difference is when a generator function is compiled they become an object that supports an iteration protocol. That means when they are called in your code they don't actually return a value and then exit. Instead, generator functions will automatically suspend and resume their execution and state around the last point of value generation. The main advantage here is that instead of having to compute an entire series of values up front, the generator computes one value and then suspends its activity awaiting the next instruction. This feature is known as *state suspension*.

To start getting a better understanding of generators, let's go ahead and see how we can create some.

[120]:
```
# Generator function for the cube of numbers (power of 3)
def gencubes(n):
    for num in range(n):
        yield num**3
```

[121]:
```
for x in gencubes(10):
    print(x)
```

```
0
1
8
27
64
125
216
343
512
729
```

Great! Now since we have a generator function we don't have to keep track of every single cube we created.

Generators are best for calculating large sets of results (particularly in calculations that involve loops themselves) in cases where we don't want to allocate the memory for all of the results at the same time.

Let's create another example generator which calculates fibonacci numbers:

[122]:
```
def genfibon(n):
    """
    Generate a fibonnaci sequence up to n
    """
    a = 1
    b = 1
    for i in range(n):
        yield a
        a,b = b,a+b
```

[123]:
```
for num in genfibon(10):
    print(num)
```

```
1
1
2
3
5
8
13
21
34
55
```

What if this was a normal function, what would it look like?

[124]:
```
def fibon(n):
    a = 1
    b = 1
    output = []

    for i in range(n):
        output.append(a)
        a,b = b,a+b

    return output
```

[125]:
```
fibon(10)
```

[125]:
```
[1, 1, 2, 3, 5, 8, 13, 21, 34, 55]
```

Notice that if we call some huge value of n (like 100000) the second function will have to keep track of every single result, when in our case we actually only care about the previous result to generate the next one!

### 5.1.1 next() and iter() built-in functions

A key to fully understanding generators is the next() function and the iter() function.

The next() function allows us to access the next element in a sequence. Lets check it out:

[126]:
```
def simple_gen():
    for x in range(3):
        yield x

# Assign simple_gen
g = simple_gen()
```

[127]:
```
for i in range(3):
    print(next(g))
```

```
0
1
2
```

[128]:
```
try:
    print(next(g))
except:
    print('error')
```

```
error
```

After yielding all the values next() caused a StopIteration error. What this error informs us of is that all the values have been yielded.

You might be wondering that why don't we get this error while using a for loop? A for loop automatically catches this error and stops calling next().

Let's go ahead and check out how to use iter(). You remember that strings are iterables:

[129]:
```
s = 'hello'

#Iterate over string
for let in s:
    print(let)
```

```
h
e
l
l
o
```

But that doesn't mean the string itself is an *iterator*! We can check this with the next() function:

[130]:
```
try:
    next(s)
except:
    print('error')
```

```
error
```

Interesting, this means that a string object supports iteration, but we can not directly iterate over it as we could with a generator function. The iter() function allows us to do just that!

[131]:
```
s_iter = iter(s)
```

[132]:
```
next(s_iter)
```

[132]:
```
'h'
```

[133]:
```
next(s_iter)
```

[133]:
```
'e'
```

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.

## 5.2 Iterators

Python iterator objects are required to support two methods while following the iterator protocol.

`__iter__`

returns the iterator object itself. This is used in for and in statements.

`__next__`

method returns the next value from the iterator. If there is no more items to return then it should raise StopIteration exception.

[134]:
```
class Counter(object):
    def __init__(self, low, high):
        self.current = low
        self.high = high

    def __iter__(self):
        'Returns itself as an iterator object'
        return self

    def __next__(self):
        'Returns the next value till current is lower than high'
        if self.current > self.high:
            raise StopIteration
        else:
            self.current += 1
            return self.current - 1
```

[135]:
```
c = Counter(5,10)
for i in c: print(i, end=' ')
```

```
5 6 7 8 9 10
```

[136]:
```
c = Counter(5,10)
next(c)
```

[136]:
```
5
```

Great! Now you know how to convert objects that are iterable into iterators themselves!

The main takeaway from this lecture is that using the yield keyword at a function will cause the function to become a generator. This change can save you a lot of memory for large use cases. For more information on generators check out:

Stack Overflow Answer

Another StackOverflow Answer