Python Class/Object, and Modules/Packages

FPT Education

FPT UNIVERSITY

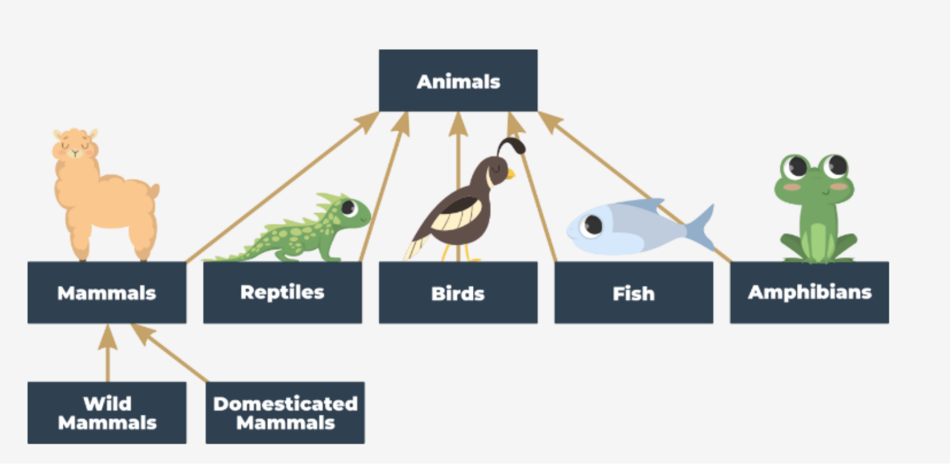
- Class Constructor, Inheritance and Polymorphism
- Module and Package Construction
- Problem Solving: Object Oriented Titanic Survival Classification

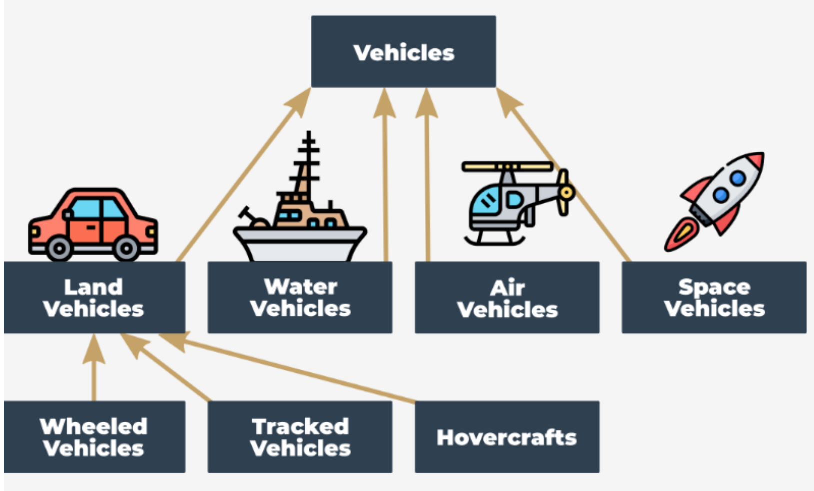
Procedural vs. the object-oriented approach

- In the procedural approach, it's possible to distinguish two different and completely separate worlds: the world of data, and the world of code.
  - Functions are able to use data, but not vice versa.
- The object approach suggests a completely different way of thinking. The data and the code are enclosed together in the same world, divided into classes.
  - Every class is like a recipe which can be used when you want to create a useful object (this is where the name of the approach comes from). You may produce as many objects as you need to solve your problem.
  - Objects are incarnations of ideas expressed in classes, like a cheesecake on your plate is an incarnation of the idea expressed in a recipe printed in an old cookbook

Class hierarchies

- The vehicles class is very broad. Too broad. We have to define some more specialized classes, then. The specialized classes are the subclasses. The vehicles class will be a superclass for them all



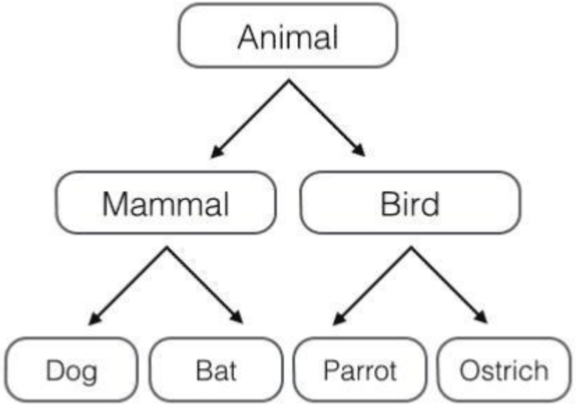


Class Constructor, Inheritance and Polymorphism

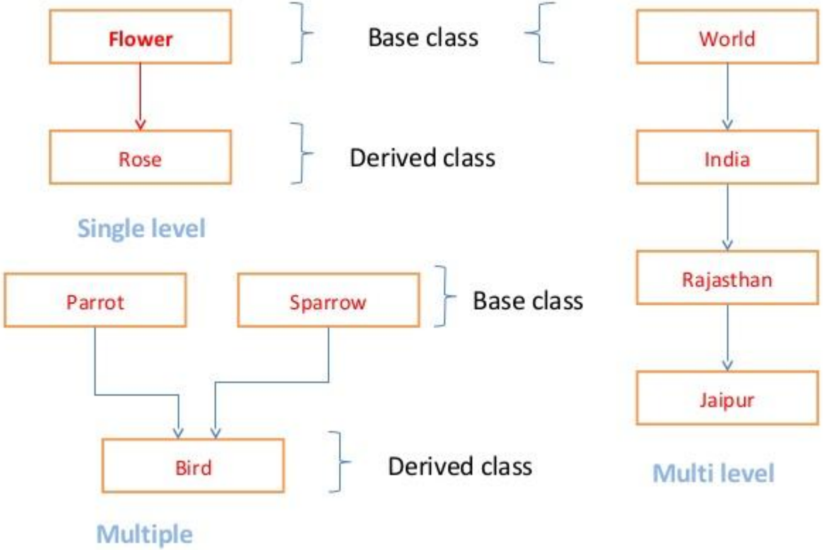
- Python is a multi-paradigm programming language. It supports different programming approaches. One of the popular approaches to solve a programming problem is by creating objects. This is known as Object-Oriented Programming (OOP).
- An object has two characteristics: attributes and behavior. The concept of OOP in Python focuses on creating reusable code. This concept is also known as DRY (Don't Repeat Yourself).
- Example:
- A parrot is an object, as it has the following properties:
  - name, age, color as attributes
  - singing, dancing as behavior

Class Constructor, Inheritance and Polymorphism

Object Hierachy



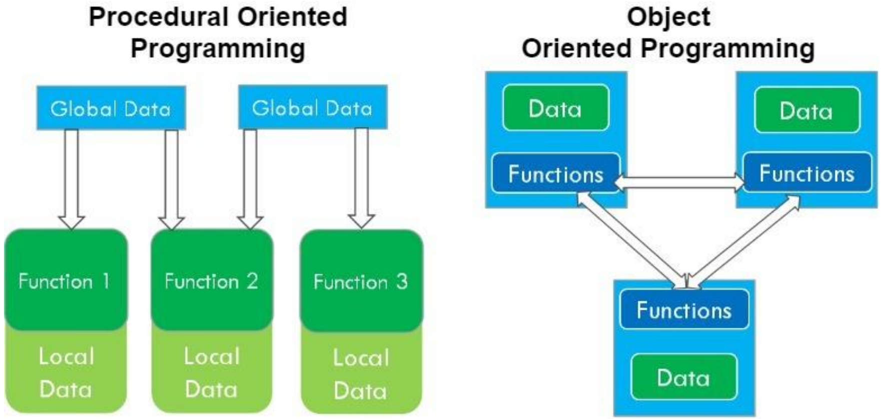
Derived Class



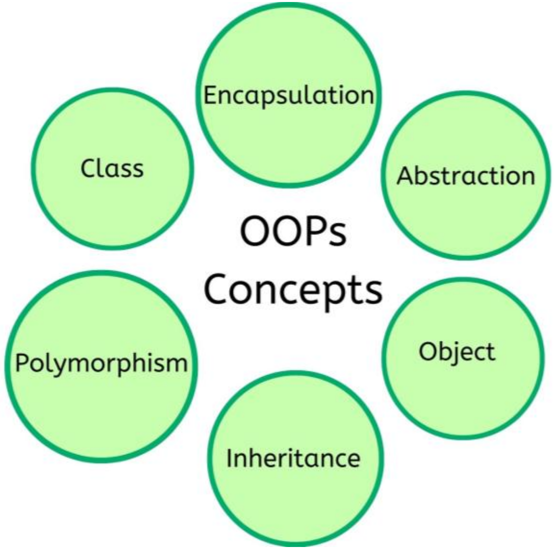
Class Constructor, Inheritance and Polymorphism

FPT Education
FPT UNIVERSITY

POP vs OOP



OOPs Concepts

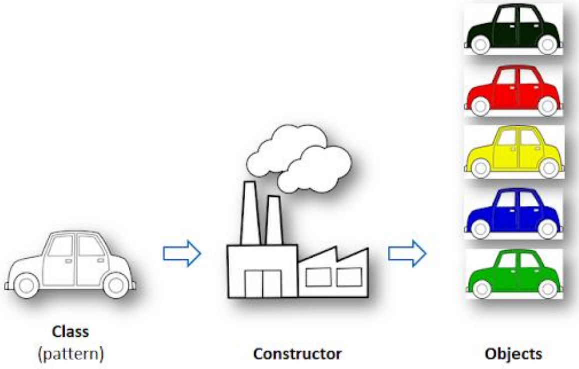


Class Constructor, Inheritance and Polymorphism

- A class attribute is a Python variable that belongs to a class rather than a particular object. It is shared between all the objects of this class and it is defined outside the constructor function, __init__(self,...), of the class.
- Instance attributes are owned by the specific instances of a class. This means for two different instances the instance attributes are usually different. This variable is only accessible in the scope of this object and it is defined inside the constructor function, __init__(self,..) of the class.

Class Constructor, Inheritance and Polymorphism

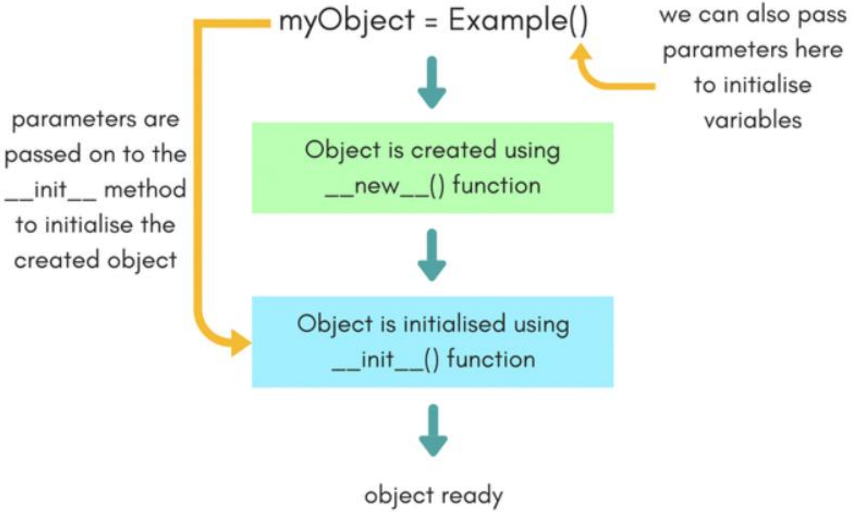
- Example
- class Car:
  - number_of_instances = 0 #class attribute
  - def __init__(self, width = 0, height = 0):
  -     self.height = height # instances attribute
  -     self.width = width   # instances attribute
  -     Car.number_of_instances += 1



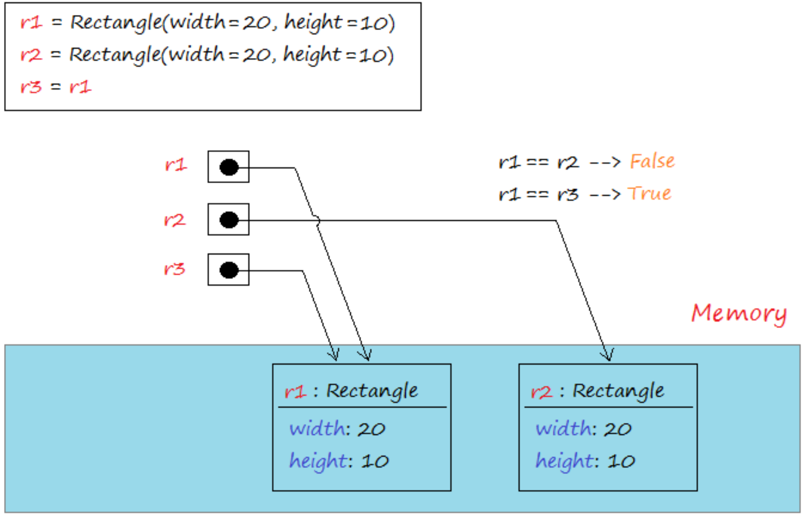
Class Constructor, Inheritance and Polymorphism

-

Object Initialization


myObject = Example()
we can also pass parameters here to initialise variables
parameters are passed on to the __init__ method to initialise the created object
Object is created using __new__() function
Object is initialised using __init__() function
object ready

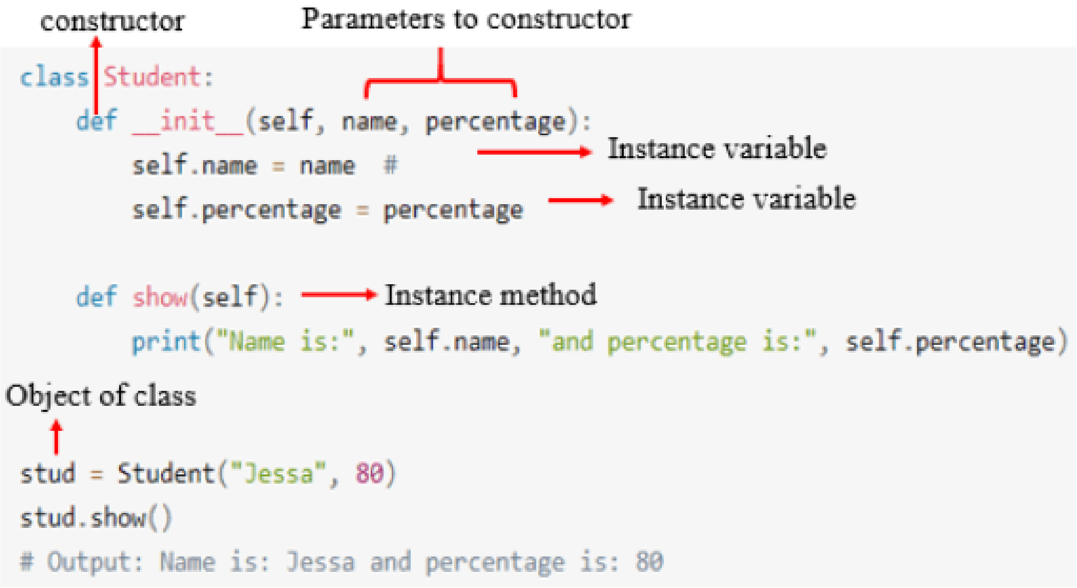
Memory Initialization



```
r1 = Rectangle(width=20, height=10)
r2 = Rectangle(width=20, height=10)
r3 = r1
```

Class Constructor, Inheritance and Polymorphism

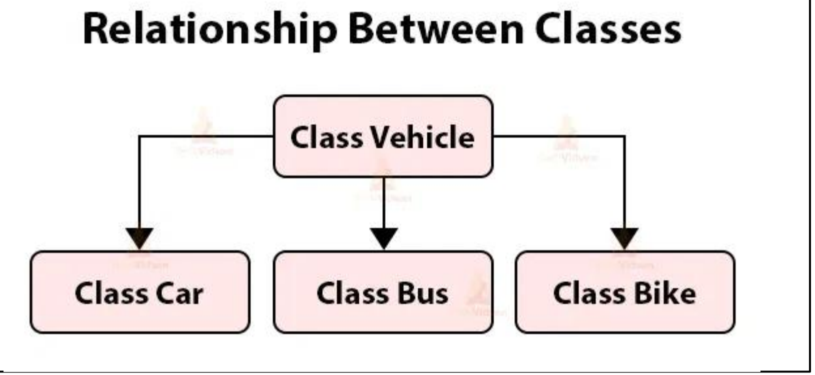
Example of Class Student



```
class Student:
    def __init__(self, name, percentage):
        self.name = name  #
        self.percentage = percentage

    def show(self):
        print("Name is:", self.name, "and percentage is:", self.percentage)

stud = Student("Jessa", 80)
stud.show()
# Output: Name is: Jessa and percentage is: 80
```

Class Constructor, Inheritance and Polymorphism

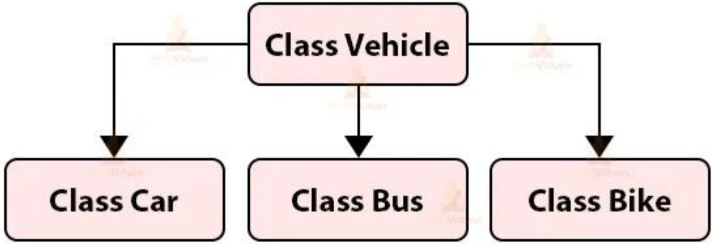
- Inheritance in object-oriented programming is inspired by the real-world inheritance
- In Python, inheritance is the capability of a class to pass some of its properties or methods to it’s derived class(child class).
- With inheritance, we build a relationship between classes based on how they are derived.



Class Constructor, Inheritance and Polymorphism

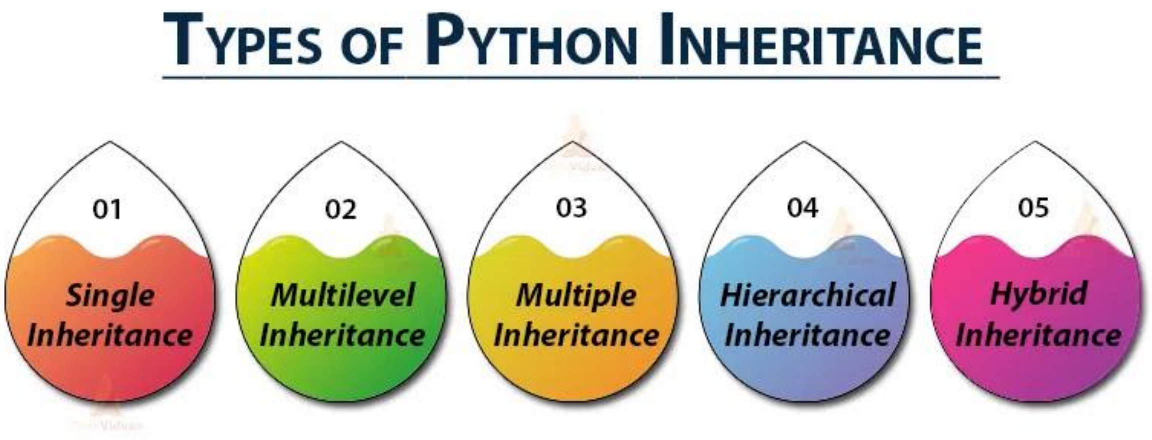
- Example:
- Every car, bus, bikes are vehicles
- > Here, the Vehicle will be called parent or
- base class while the car, bus, and bike
- are its child or derived class.

Relationship Between Classes

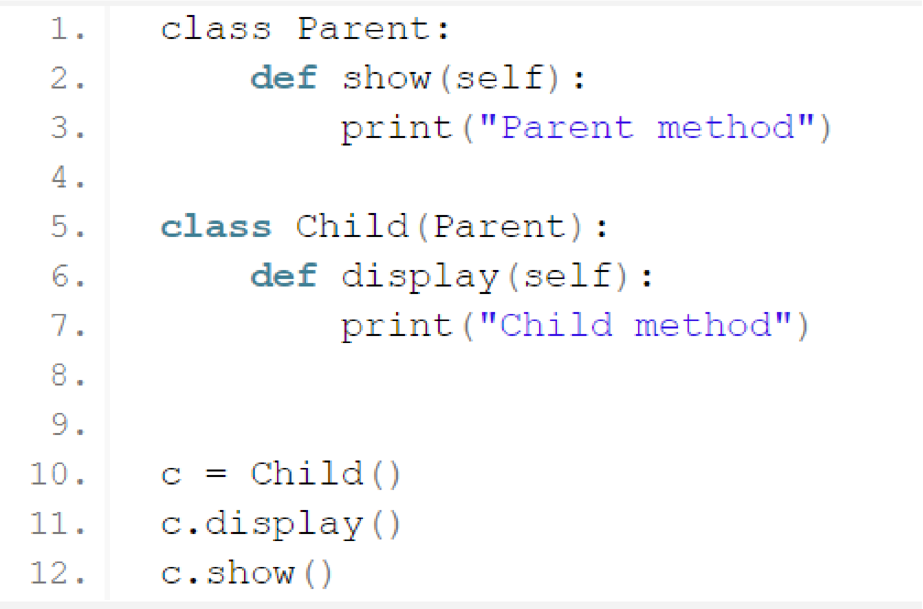


Class Constructor, Inheritance and Polymorphism

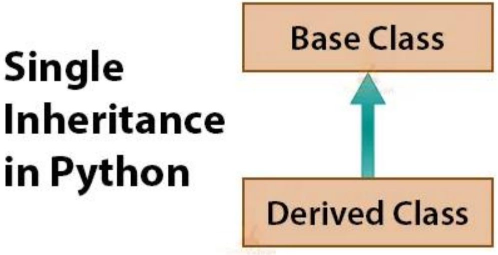
Inheritance Types



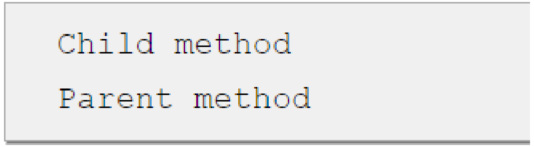
Class Constructor, Inheritance and Polymorphism



```python
class Parent:
    def show(self):
        print("Parent method")

class Child(Parent):
    def display(self):
        print("Child method")



c = Child()
c.display()
c.show()
```

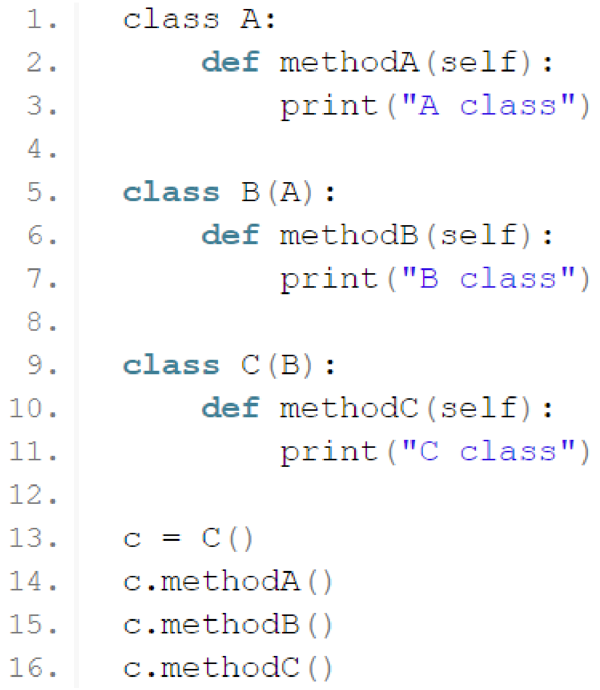


Output:

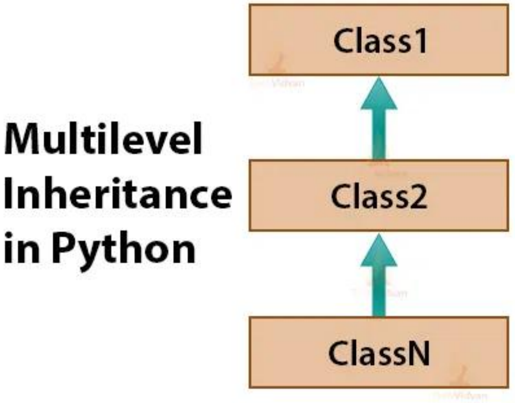


```
Child method
Parent method
```

Class Constructor, Inheritance and Polymorphism

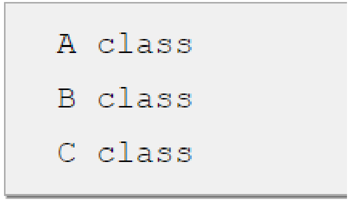


```python
class A:
    def methodA(self):
        print("A class")

class B(A):
    def methodB(self):
        print("B class")

class C(B):
    def methodC(self):
        print("C class")

c = C()
c.methodA()
c.methodB()
c.methodC()
```



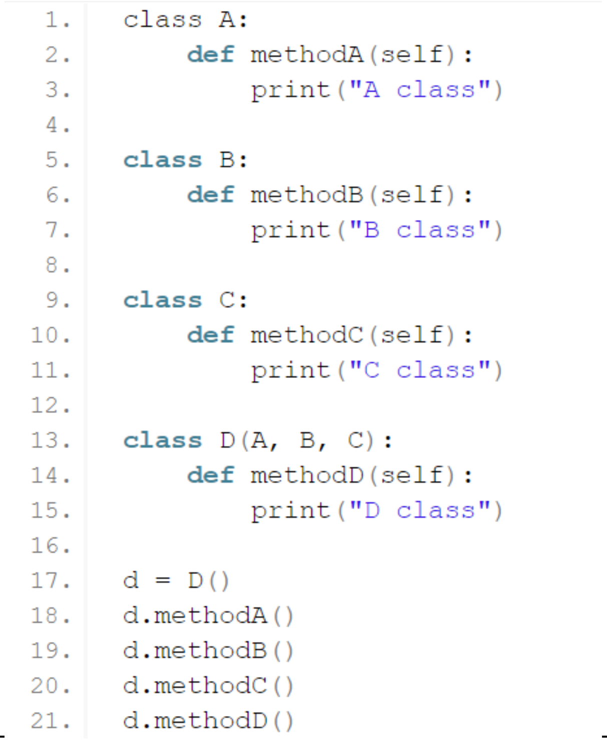
Multilevel Inheritance in Python

Output:

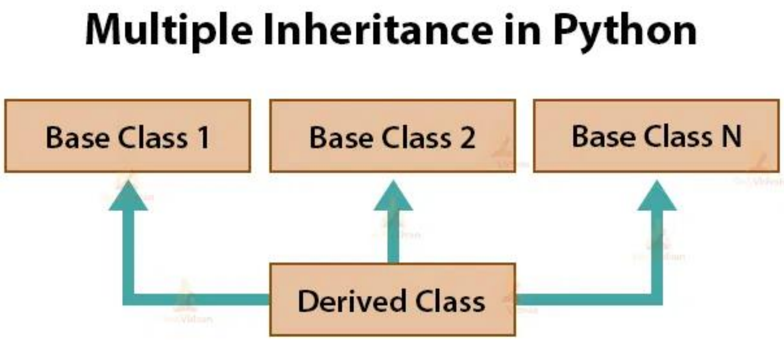


```
A class
B class
C class
```

Class Constructor, Inheritance and Polymorphism

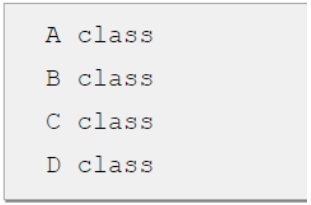


```python
class A:
    def methodA(self):
        print("A class")

class B:
    def methodB(self):
        print("B class")

class C:
    def methodC(self):
        print("C class")

class D(A, B, C):
    def methodD(self):
        print("D class")

d = D()
d.methodA()
d.methodB()
d.methodC()
d.methodD()
```



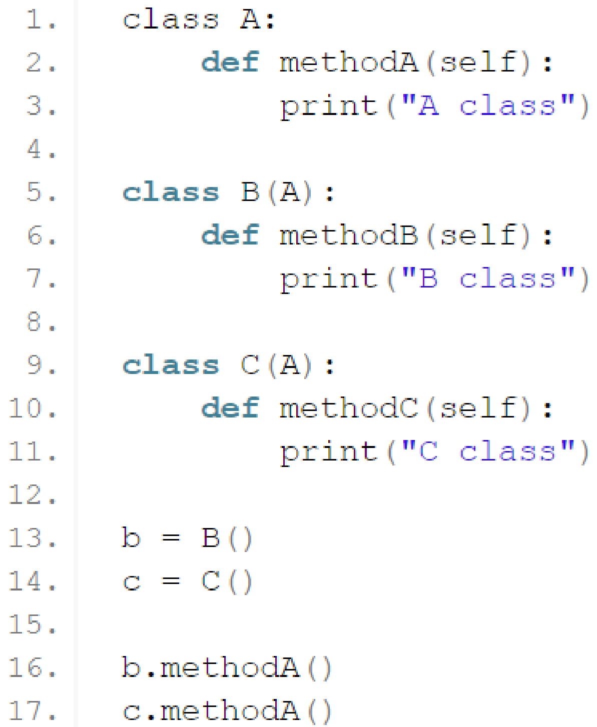
Multiple Inheritance in Python

Output:

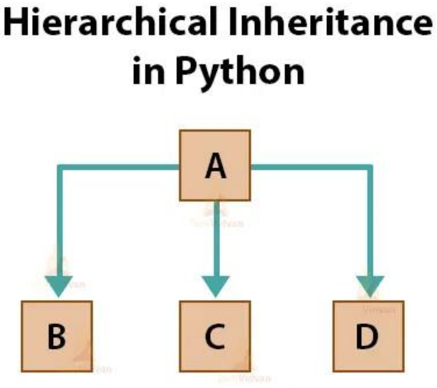


```
A class
B class
C class
D class
```

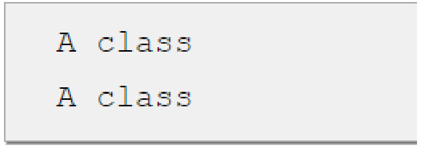
Class Constructor, Inheritance and Polymorphism



```python
class A:
    def methodA(self):
        print("A class")

class B(A):
    def methodB(self):
        print("B class")

class C(A):
    def methodC(self):
        print("C class")

b = B()
c = C()

b.methodA()
c.methodA()
```

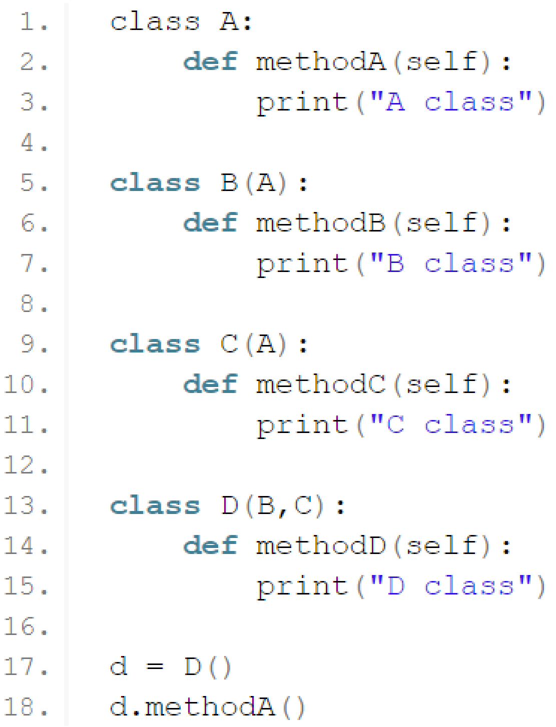


Output:



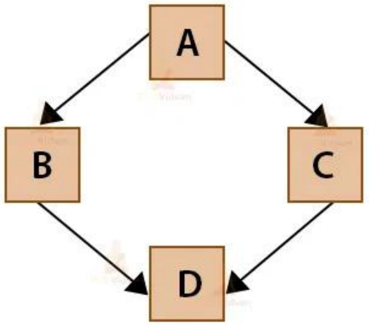
```
A class
A class
```

Class Constructor, Inheritance and Polymorphism



```python
class A:
    def methodA(self):
        print("A class")

class B(A):
    def methodB(self):
        print("B class")

class C(A):
    def methodC(self):
        print("C class")

class D(B,C):
    def methodD(self):
        print("D class")

d = D()
d.methodA()
```

Hybrid Inheritance in Python

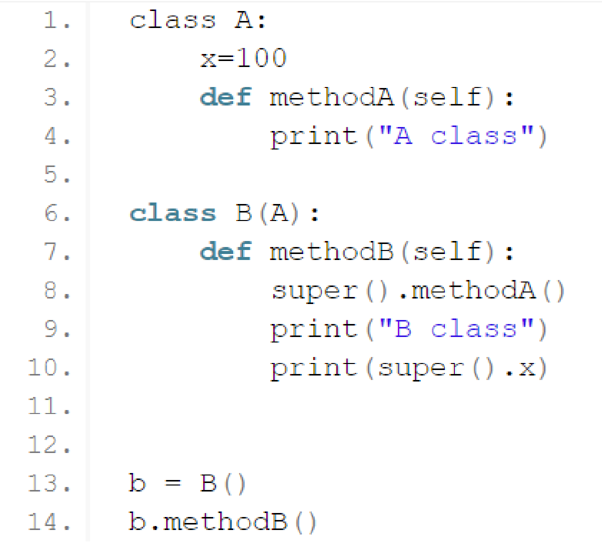


Output:

```
A class
```

Class Constructor, Inheritance and Polymorphism

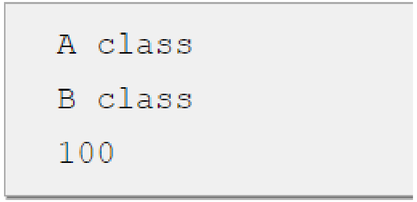
- **Example Inheritance with super() function**



```
class A:
    x=100
    def methodA(self):
        print("A class")

class B(A):
    def methodB(self):
        super().methodA()
        print("B class")
        print(super().x)


b = B()
b.methodB()
```

Super() is a proxy object which is used to refer to the parent object.
We can call super() method to access the properties or methods of the parent class.

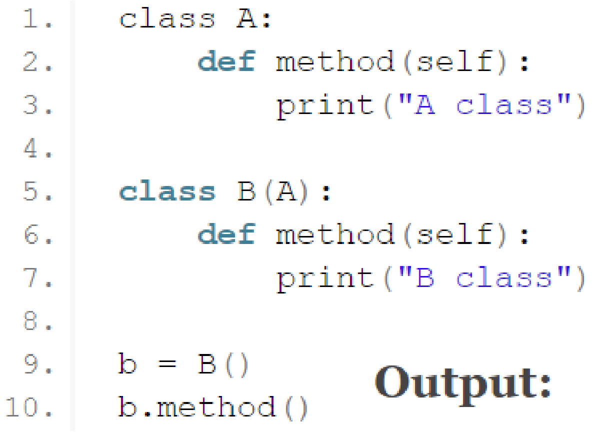
Output:



```
A class
B class
100
```

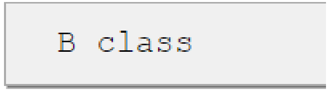
Class Constructor, Inheritance and Polymorphism

- ## Example Inheritance with Overriding Methods



```
class A:
    def method(self):
        print("A class")

class B(A):
    def method(self):
        print("B class")

b = B()
b.method()
```

Output:



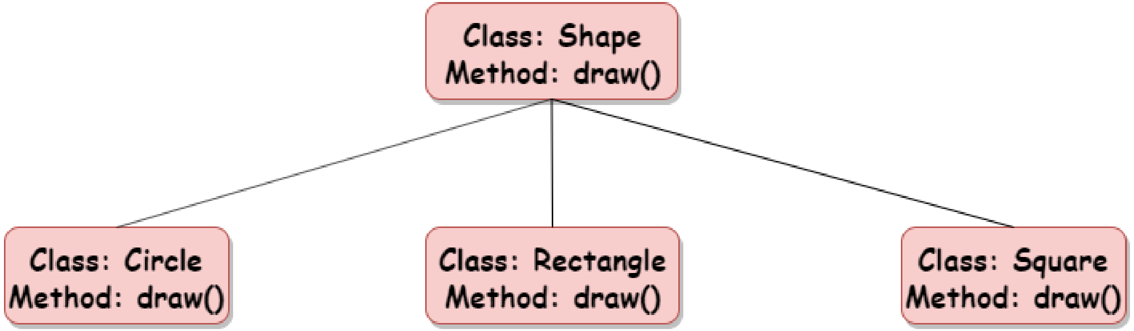
```
B class
```

- Method overriding is an important concept in object-oriented programming.
- Method overriding allows us to redefine a method by overriding it

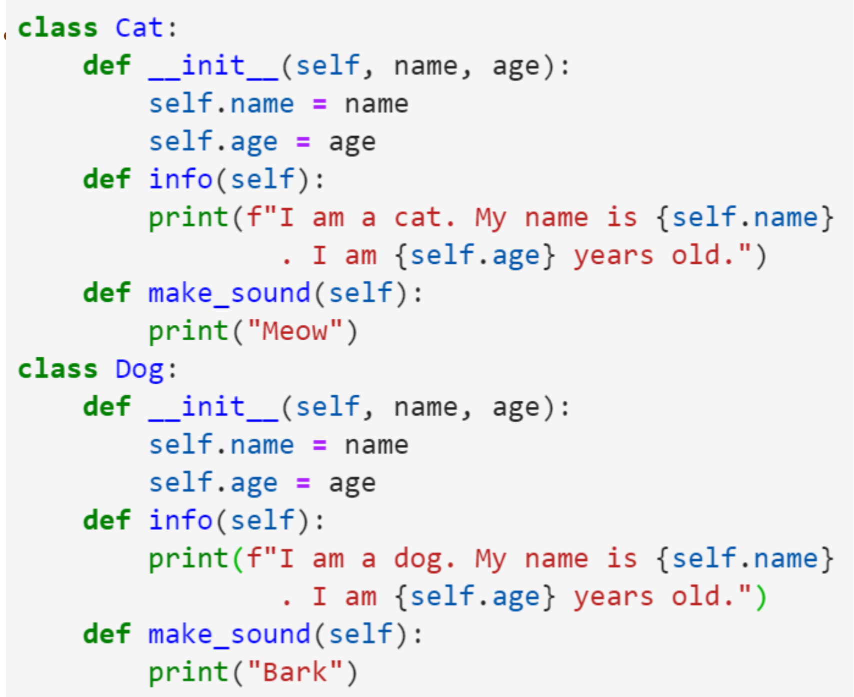
For method overriding, we must satisfy two conditions:
1. There should be a parent-child relationship between the classes.
Inheritance is a must.
2. The name of the method and the parameters should be the same in the base and derived class in order to override it.

Class Constructor, Inheritance and Polymorphism

- Polymorphism is a very important concept in Object-Oriented Programming.
- We can use the concept of polymorphism while creating class methods as Python allows different classes to have methods with the same name.
- Example

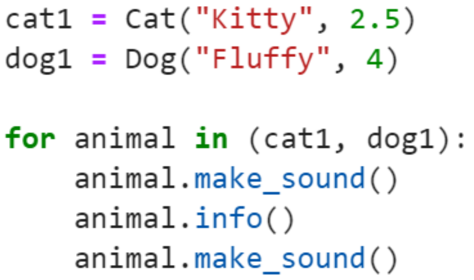


Class Constructor, Inheritance and Polymorphism

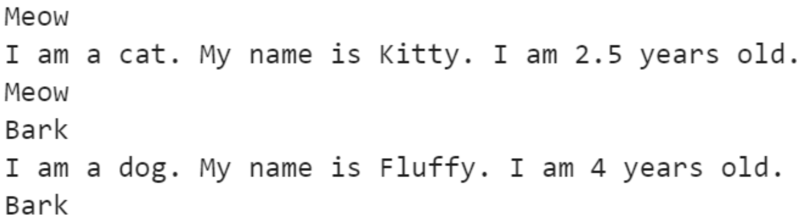


```
class Cat:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def info(self):
        print(f"I am a cat. My name is {self.name}
                . I am {self.age} years old.")
    def make_sound(self):
        print("Meow")
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def info(self):
        print(f"I am a dog. My name is {self.name}
                . I am {self.age} years old.")
    def make_sound(self):
        print("Bark")
```

Class Polymorphism

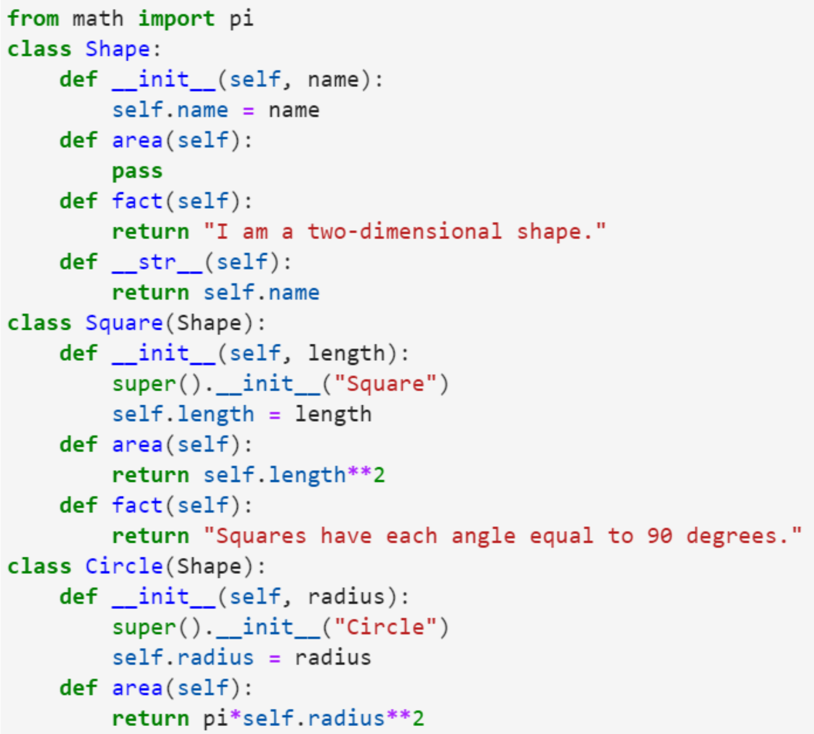


```
cat1 = Cat("Kitty", 2.5)
dog1 = Dog("Fluffy", 4)

for animal in (cat1, dog1):
    animal.make_sound()
    animal.info()
    animal.make_sound()
```



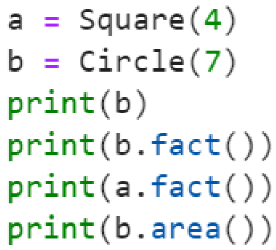
```
Meow
I am a cat. My name is Kitty. I am 2.5 years old.
Meow
Bark
I am a dog. My name is Fluffy. I am 4 years old.
Bark
```

Class Constructor, Inheritance and Polymorphism

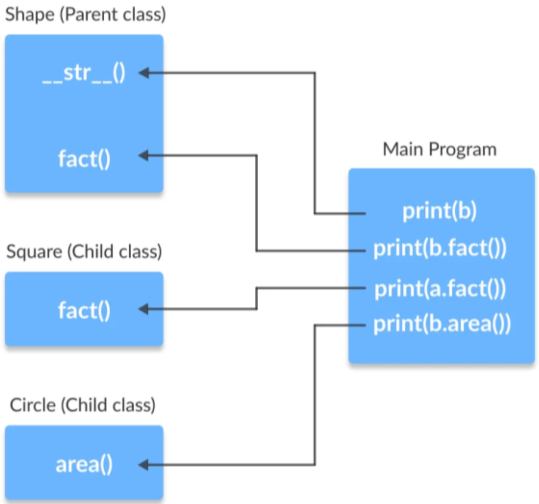


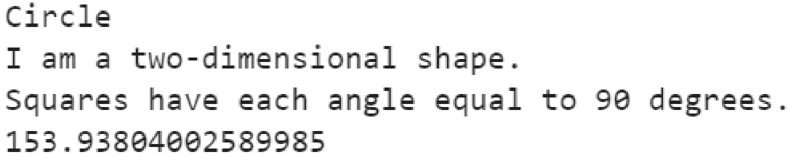
```python
from math import pi
class Shape:
    def __init__(self, name):
        self.name = name
    def area(self):
        pass
    def fact(self):
        return "I am a two-dimensional shape."
    def __str__(self):
        return self.name
class Square(Shape):
    def __init__(self, length):
        super().__init__("Square")
        self.length = length
    def area(self):
        return self.length**2
    def fact(self):
        return "Squares have each angle equal to 90 degrees."
class Circle(Shape):
    def __init__(self, radius):
        super().__init__("Circle")
        self.radius = radius
    def area(self):
        return pi*self.radius**2
```

Method Overriding



```python
a = Square(4)
b = Circle(7)
print(b)
print(b.fact())
print(a.fact())
print(b.area())
```





```
Circle
I am a two-dimensional shape.
Squares have each angle equal to 90 degrees.
153.93804002589985
```

Class Constructor, Inheritance and Polymorphism

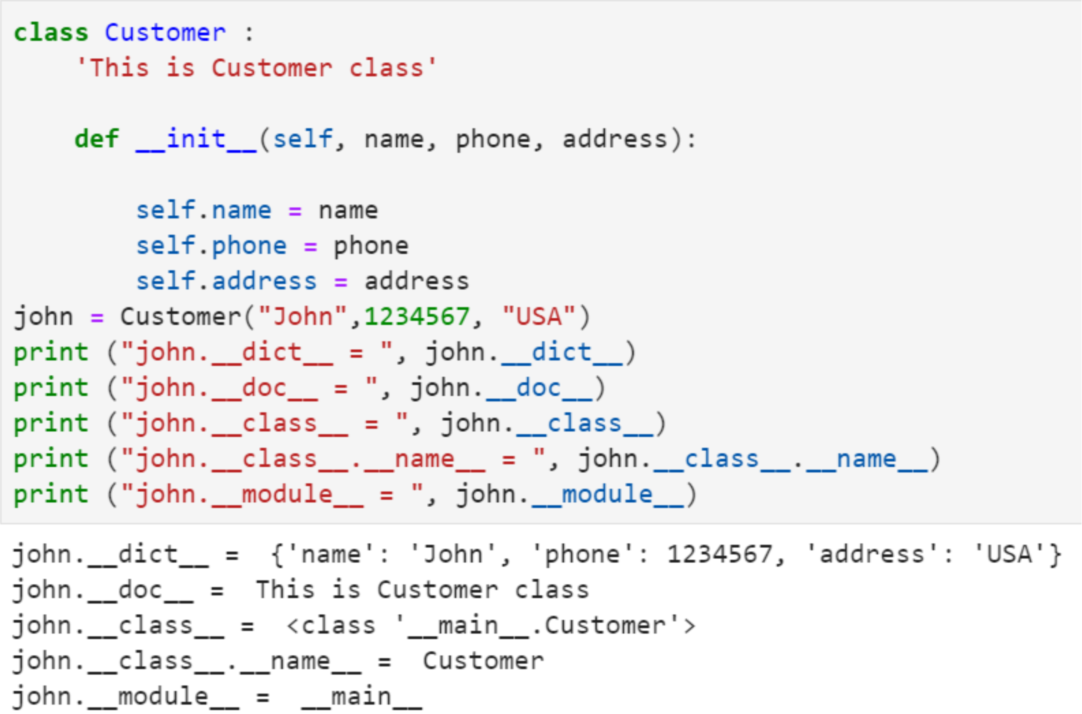
- Python classes are descendants of the object class. So they inherit the following attributes:

Attribute	Description
__dict__	Giving information about this class in a short, easy to understand, as one dictionary
__doc__	Returns a string describing the class, or returns None if it is not defined
__class__	Returns an object, containing information about the class, which has many useful attributes, including the __name__ attribute.
module	Returns the 'module' name of the class, or returns "__main__" if that class is defined in the module being run.

Class Constructor, Inheritance and Polymorphism

-

Built-in Attributes

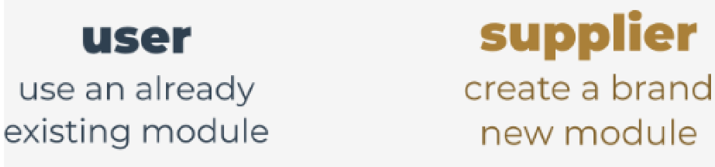


```
class Customer :
    'This is Customer class'

    def __init__(self, name, phone, address):

        self.name = name
        self.phone = phone
        self.address = address
john = Customer("John",1234567, "USA")
print ("john.__dict__ = ", john.__dict__)
print ("john.__doc__ = ", john.__doc__)
print ("john.__class__ = ", john.__class__)
print ("john.__class__.__name__ = ", john.__class__.__name__)
print ("john.__module__ = ", john.__module__)
```

```
john.__dict__ =  {'name': 'John', 'phone': 1234567, 'address': 'USA'}
john.__doc__ =  This is Customer class
john.__class__ =  <class '__main__.Customer'>
john.__class__.__name__ =  Customer
john.__module__ =  __main__
```

Module and Package Construction

- Computer code has a tendency to grow
- A code which is not able to respond to users' needs will be forgotten quickly
- A larger code always means tougher maintenance.
- Searching for bugs is always easier where the code is smaller
  - =>divide all the tasks among the developers;
  - =>join all the created parts into one working whole.
- How to make use of a module?



user

use an already existing module

supplier

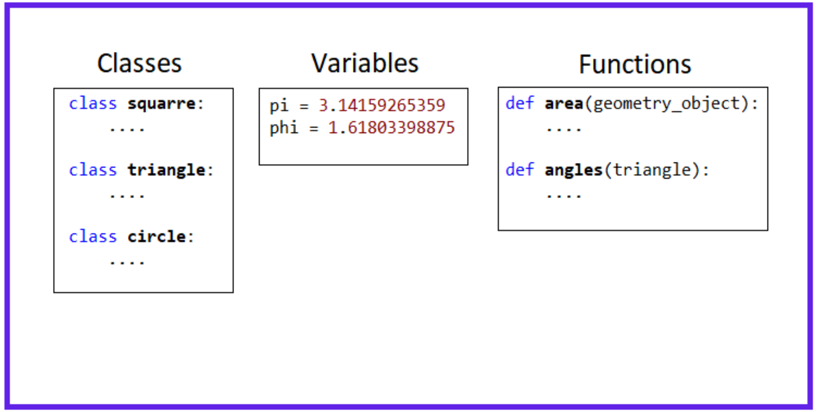
create a brand new module

  - Each module consists of entities (like a book consists of chapters).
  - These entities can be functions, variables, constants, classes, and objects.
  - can make use of any of the entities it stores.

Module and Package Construction

- Exiting the interpreter destroys all functions and variables we created. But when we want a longer program, we create a script.
- With Python, we can put such definitions in a file, and use them in a script, or in an interactive instance of the interpreter. Such a file is a module.
- Example: import geometry
- sq = geometry.square(4)
- tri = geometry.triangle(3, 6, 5)
- print(geometry.pi) # Output : 3.1415
- geometry.area(sq) # Output: 16

geometry.py



```
class squarre:
    ....

class triangle:
    ....

class circle:
    ....
```

```
pi  = 3.14159265359
phi = 1.61803398875
```

```
def area(geometry_object):
    ....

def angles(triangle):
    ....
```

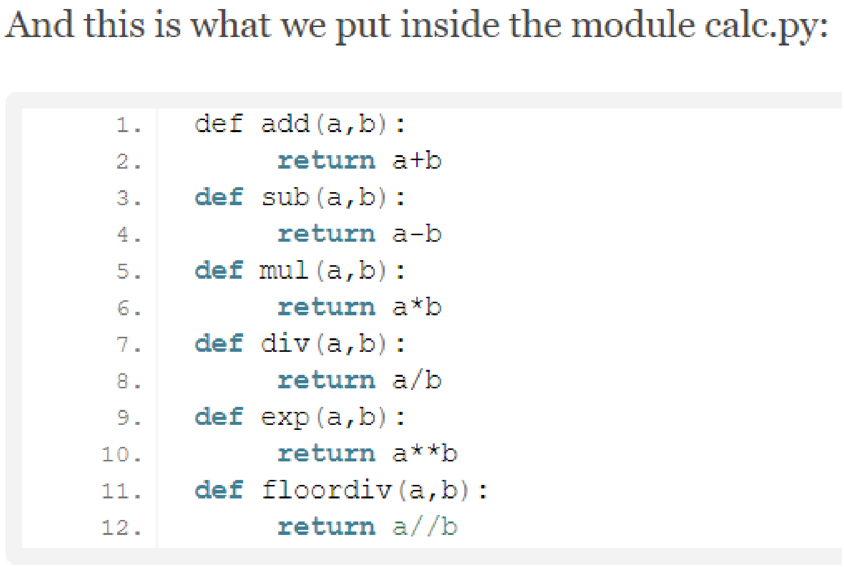
Module and Package Construction

- Example 1:

```
C:\Users\lifei>cd Desktop
C:\Users\lifei\Desktop>mkdir calc
C:\Users\lifei\Desktop>cd calc
C:\Users\lifei\Desktop\calc>echo >__init__.py
C:\Users\lifei\Desktop\calc>echo >calc.py
C:\Users\lifei\Desktop\calc>
```

Module and Package Construction

- Example 1:
- import os #Apply Modules calc
- os.chdir('C:\\Users\\lifei\\Desktop
- \\calc')
- import calc
- fd=calc.floordiv
- fd(5.5,4)
- Output: 1.0

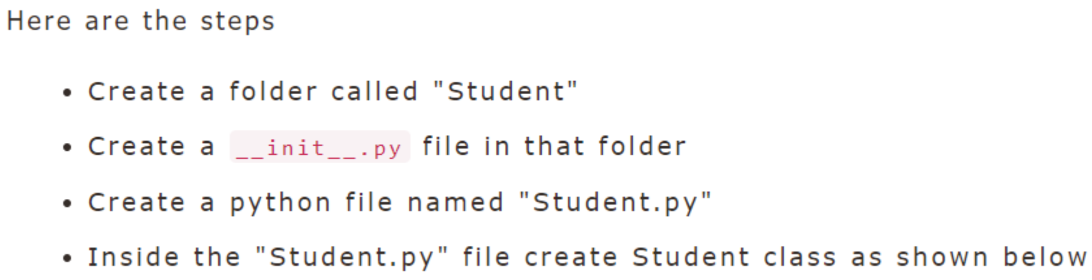


And this is what we put inside the module calc.py:

```
def add(a,b):
        return a+b
def sub(a,b):
        return a-b
def mul(a,b):
        return a*b
def div(a,b):
        return a/b
def exp(a,b):
        return a**b
def floordiv(a,b):
        return a//b
```

Module and Package Construction

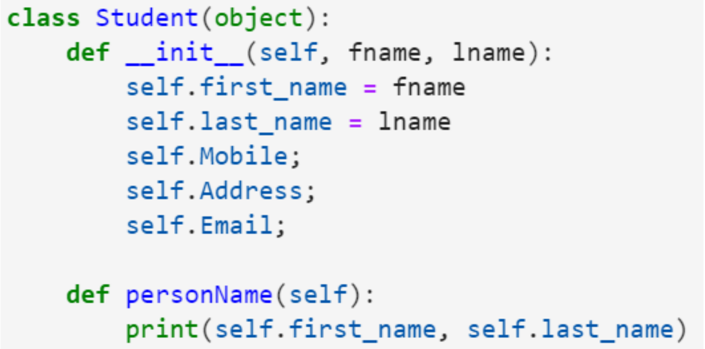
- Example 2:



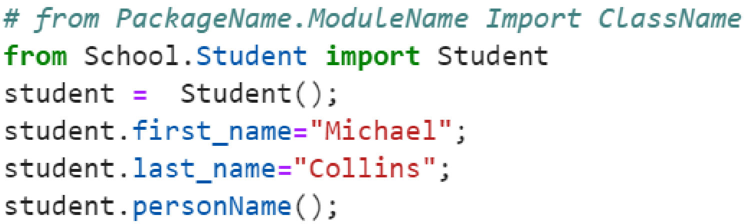
Here are the steps

- Create a folder called "Student"
- Create a `__init__.py` file in that folder
- Create a python file named "Student.py"
- Inside the "Student.py" file create Student class as shown below

-



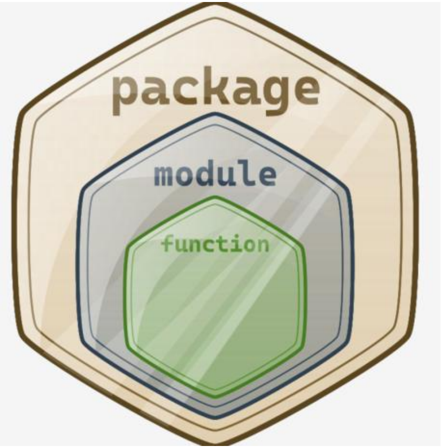
```python
class Student(object):
    def __init__(self, fname, lname):
        self.first_name = fname
        self.last_name = lname
        self.Mobile;
        self.Address;
        self.Email;

    def personName(self):
        print(self.first_name, self.last_name)
```



```python
# from PackageName.ModuleName Import ClassName
from School.Student import Student
student =  Student();
student.first_name="Michael";
student.last_name="Collins";
student.personName();
```

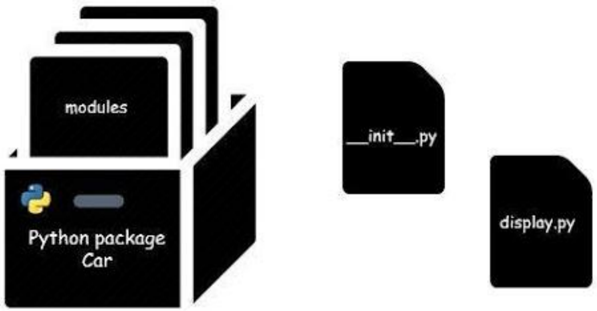
Module and Package Construction

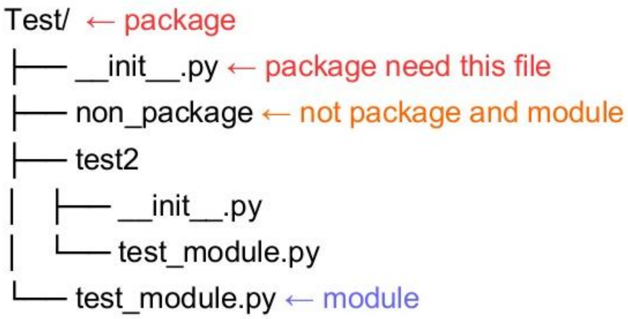
- a module is a kind of container filled with functions
- making many modules may cause a little mess - sooner or later you'll want to group your modules exactly in the same way as you've previously grouped functions
- in the world of modules, a package plays a similar role to a folder/directory in the world of files.
- A Python file named __init__.py is implicitly run when a package containing it is subject to import, and is used to initialize a package and/or its sub-packages (if any). The file may be empty, but must not be absent.



Module and Package Construction

- Packages in Python are similar to directories or folders. Just like a directory that can contain subdirectories and folders and sub-folders, a Python package can have sub-packages and modules (modules are similar to files, they have .py extension).

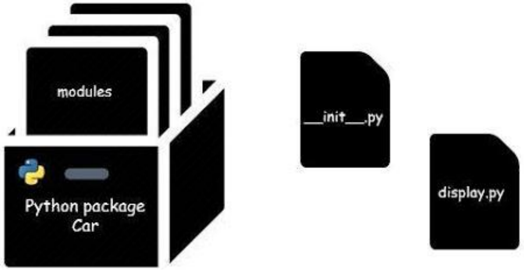




```
Test/ ← package
├── __init__.py ← package need this file
├── non_package ← not package and module
├── test2
│   ├── __init__.py
│   └── test_module.py
└── test_module.py ← module
```

Module and Package Construction

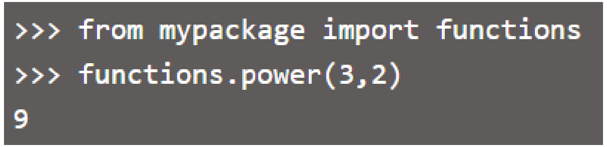
- Python has a hierarchical directory structure, with multiple sub-packages, sub-sub packages
- A directory in python (package) must contain a file named __init__.py in order for Python to consider it as a package. This file can be left empty but we usually prefer to place the initialization code for that package in this file.



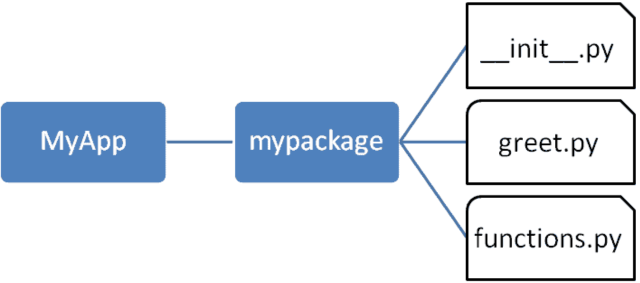
```
Test/ ← package
├── __init__.py ← package need this file
├── non_package ← not package and module
├── test2
│   ├── __init__.py
│   └── test_module.py
└── test_module.py ← module
```

Module and Package Construction

- Example:
- Let's create a package named mypackage, using the following steps:
- Create a new folder named D:\MyApp.
- Inside MyApp, create a subfolder with the name 'mypackage'.
- Create an empty __init__.py file in the mypackage folder.
- Create modules greet.py and functions.py with the following code:

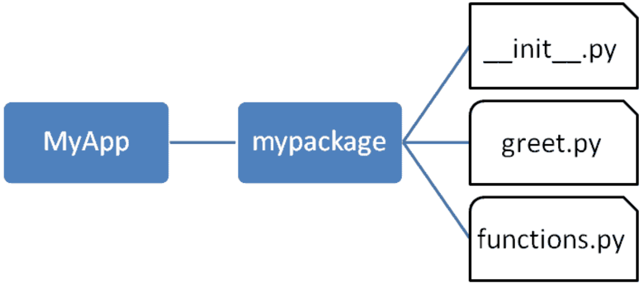


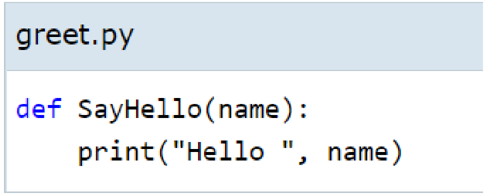
```
>>> from mypackage import functions
>>> functions.power(3,2)
9
```



Module and Package Construction

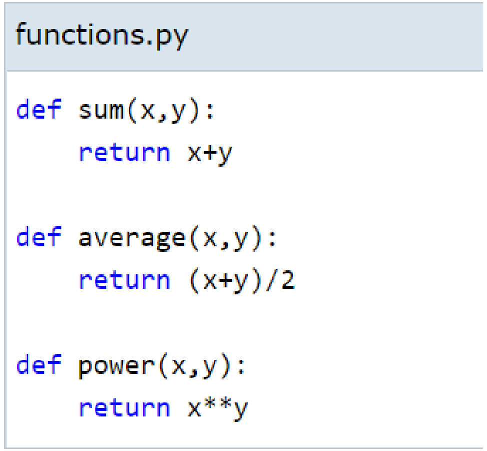
- Example:





greet.py

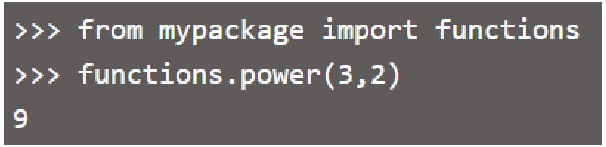
```
def SayHello(name):
    print("Hello ", name)
```



functions.py

```
def sum(x,y):
    return x+y

def average(x,y):
    return (x+y)/2

def power(x,y):
    return x**y
```

-



```
>>> from mypackage import functions
>>> functions.power(3,2)
9
```

Module and Package Construction

- How to access packages and modules in a python program?
- To access the module 'hello', simply call:

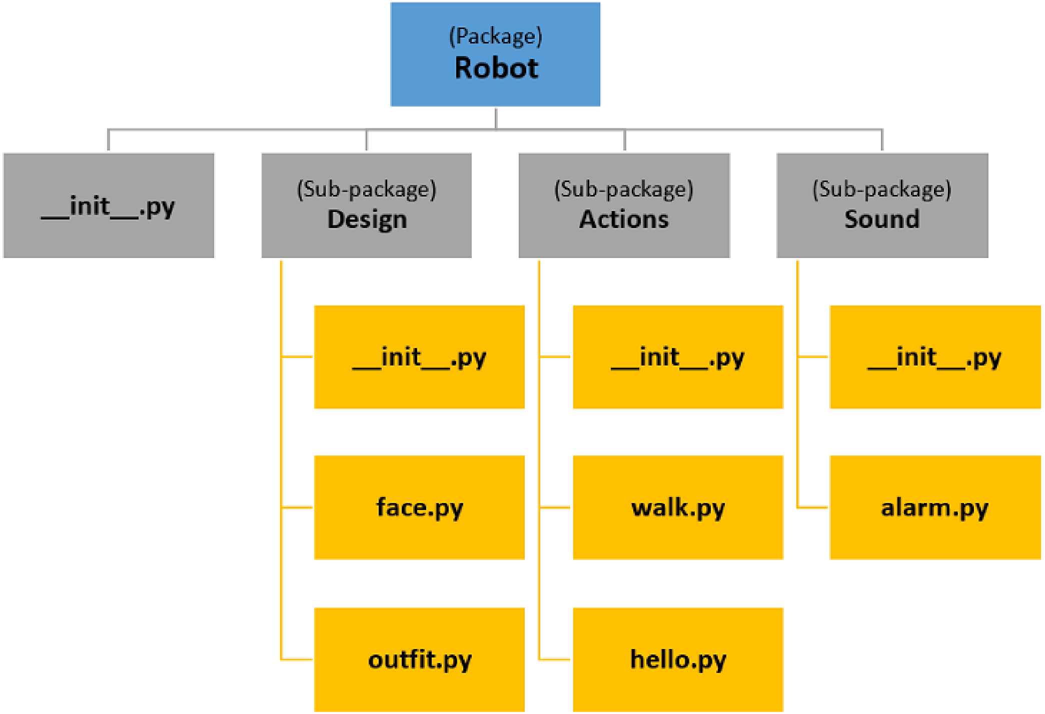
import Robot.Actions.hello

from Robot.Actions import hello

- To access all the module and sub-packages
  - from Robot import *
- How to create a Package in Python?
  - Step 1: Create a Directory (Package). We have created 'Robot'.
  - Step 2: Create a file __init__.py in the directory
  - Step 3: Create subdirectories or modules in the main directory.

Module and Package Construction

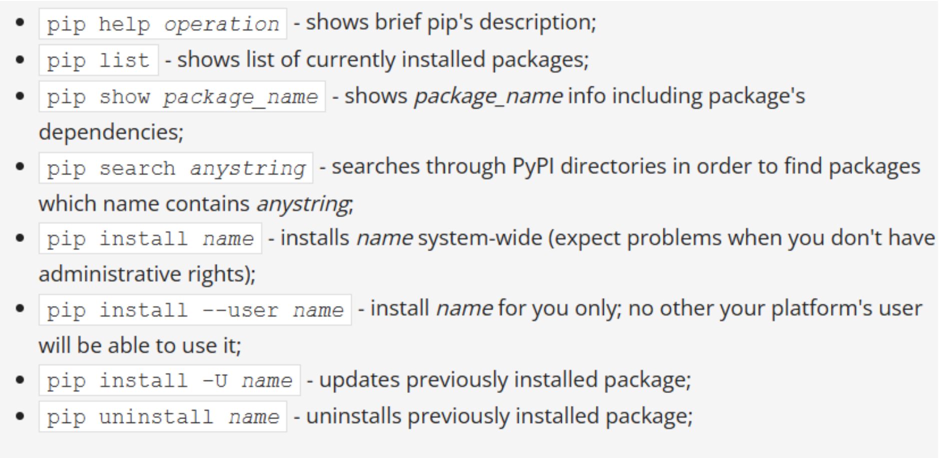
-



PIP

- A repository (or repo for short) designed to collect and share free Python code exists and works under the name Python Package Index (PyPI) although it's also likely that you come across a very niche name The Cheese Shop. The Shop's website is available at https://pypi.org/.
- To make use of The Cheese Shop the specialized tool has been created and its name is pip (pip installs packages)

https://pypi.org/



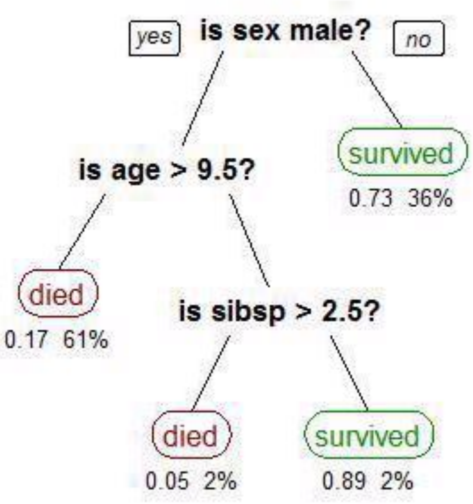
- `pip help operation` - shows brief pip's description;
- `pip list` - shows list of currently installed packages;
- `pip show package_name` - shows *package_name* info including package's dependencies;
- `pip search anystring` - searches through PyPI directories in order to find packages which name contains *anystring*;
- `pip install name` - installs *name* system-wide (expect problems when you don't have administrative rights);
- `pip install --user name` - install *name* for you only; no other your platform's user will be able to use it;
- `pip install -U name` - updates previously installed package;
- `pip uninstall name` - uninstalls previously installed package;

Problem Solving

- Object Oriented Titanic Survival Classification
- The Challenge
- On April 15, 1912, during her maiden voyage, the widely considered “unsinkable” RMS Titanic sank after colliding with an iceberg. Unfortunately, there weren’t enough lifeboats for everyone onboard, resulting in the death of 1502 out of 2224 passengers and crew.
- While there was some element of luck involved in surviving, it seems some groups of people were more likely to survive than others.
- In this challenge, we ask you to build a predictive model that answers the question: “what sorts of people were more likely to survive?” using passenger data (ie name, age, gender, socio-economic class, etc).

Problem Solving

-



Problem Solving

- Variable Definition Key
- survival Survival0 = No, 1 = Yes
- pclass Ticket class 1 = 1st, 2 = 2nd, 3 = 3rd
- sex Sex
- AgeAge in years
- sibsp # of siblings / spouses aboard the Titanic
- parch # of parents / children aboard the Titanic
- ticket Ticket number
- farePassenger fare

Problem Solving

cabin Cabin number

embarked Port of Embarkation C = Cherbourg, Q = Queenstown, S = Southampton

pclass: A proxy for socio-economic status (SES)

1st = Upper

2nd = Middle

3rd = Lower

age: Age is fractional if less than 1. If the age is estimated, is it in the form of xx.5

sibsp: The dataset defines family relations in this way...

Problem Solving

- Sibling = brother, sister, stepbrother, stepsister
- Spouse = husband, wife (mistresses and fiancés were ignored)
- parch: The dataset defines family relations in this way...
- Parent = mother, father
- Child = daughter, son, stepdaughter, stepson
- Some children travelled only with a nanny, therefore parch=0 for them.