PYTHON APPLICATION PROGRAMMING

Module4

Introduction: Object-oriented programming (OOP) is a programming paradigm that models real world entities as software objects, which have data and methods. OOP is also, an approach used for creating neat and reusable code instead of a redundant one. The program is divided into self-contained objects or several mini-programs. Every Individual object represents a different part of the application having its own logic and data to communicate within themselves.

Difference between Object-Oriented and Procedural Oriented Programming

Object-Oriented Programming (OOP)	**Procedural-Oriented Programming (POP)**
It is a bottom-up approach	It is a top-down approach
Program is divided into objects	Program is divided into functions
Makes use of *Access modifiers* 'public', private', protected'	Doesn't use *Access modifiers*
It is more secure	It is less secure
Object can move freely within member functions	Data can move freely from function to function within programs
It supports inheritance	It does not support inheritance

Major OOP (object-oriented programming) concepts in Python include Class, Object, Method, Inheritance, Polymorphism, Data Abstraction, and Encapsulation.

1. Classes and Objects

Class is an abstract data type which can be defined as a template or blueprint that describes the common attributes and behavior / state of the entity the programmer has to model in the program.

Examples:

a. House plan containing the information like size, floors, doors, windows, locations of rooms, lawn, parking space and so on.

b. Blue print of the car containing the generic information of the car like name, size, shape, model, color and so on.

Defining a Class in Python

Class definition begins with a class keyword followed by name and colon. The first string inside the class is called doctstring and has a brief description about the class. The simplest form of class definition looks like this :

```
class    className :
         <statement-1>
         < statement-2>
         .
         .
         .
         < statement-n>
```

Example1:

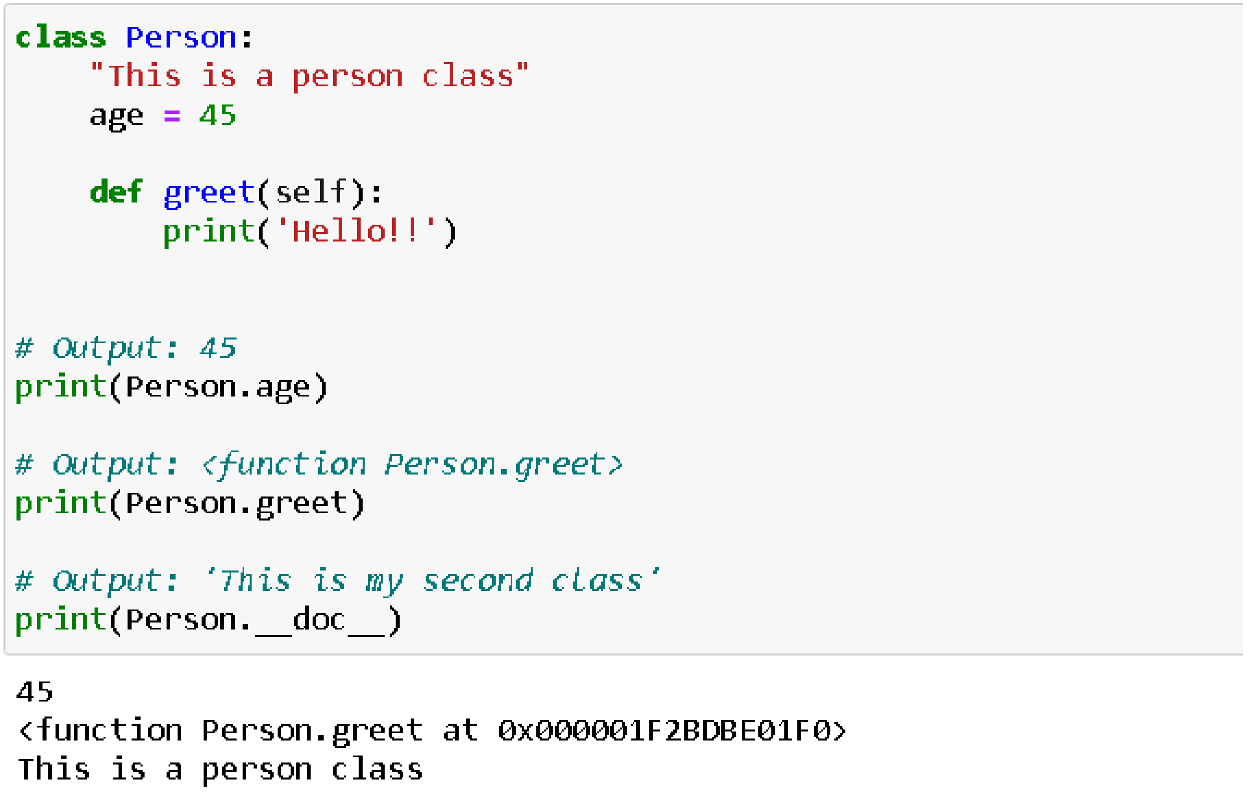
```
class MyClass: # Header
    '''This is a docstring. I have created a new class''' # Body
    pass
```

```
print(MyClass.__doc__) # To print the doc string
```

```
This is a docstring. I have created a new class
```

The header indicates that the new class is called My Class , The body is a docstring that explains what the class is for. Also, we used the Python keyword pass here. This is very often used as a place holder where code will eventually go. It allows us to run this code without throwing an error.
You can define variables and methods inside a class dentition as illustrated in the following examples:

Example 2:



```
class Person:
    "This is a person class"
    age = 45

    def greet(self):
        print('Hello!!')


# Output: 45
print(Person.age)

# Output: <function Person.greet>
print(Person.greet)

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

```
45
<function Person.greet at 0x000001F2BDBE01F0>
This is a person class
```

Creating an Object in Python:

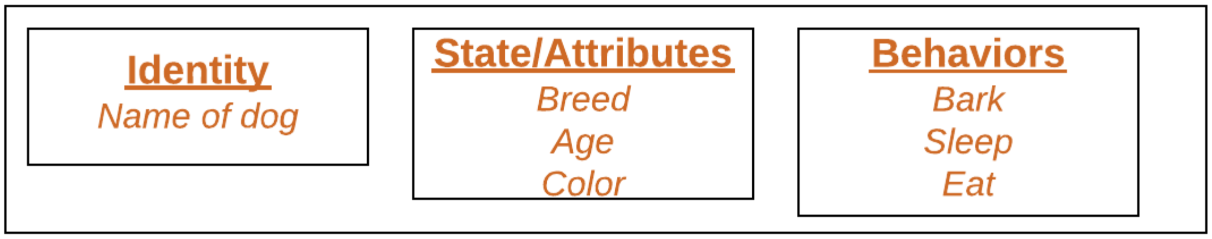
Class only provides a blue print. Class instances or objects actually implement the class. To use functions and data defined inside a class one should create instances or objects of the class. A class can have multiple objects. Each object will have their own copy of data and methods.

An Object is an instance of a Class. **It is an entity that has state and behavior. In a nutshell, it is an instance of a class that can access the data.**
Syntax: obj = class1()
Here obj is the "object " of class1.

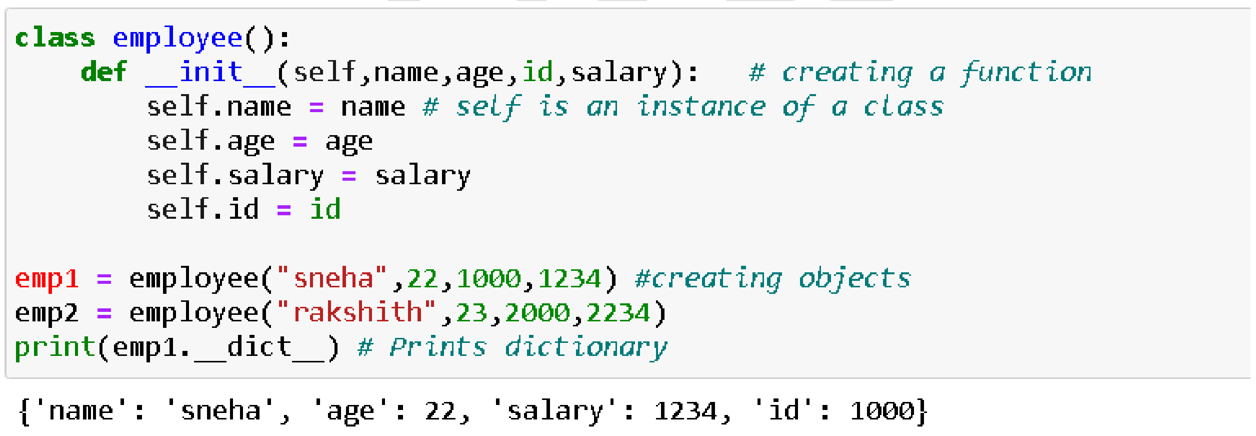
A class is like a blueprint while an instance is a copy of the class with *actual values*. It's not an idea anymore, it's an actual dog, like a dog of breed pug who's seven years old. You can have many dogs to create many different

instances, but without the class as a guide, you would be lost, not knowing what information is required.
An object consists of :

- **State :** It is represented by attributes of an object. It also reflects the properties of an object.
- **Behavior :** It is represented by methods of an object. It also reflects the response of an object with other objects.
- **Identity :** It gives a unique name to an object and enables one object to interact with other objects.



Example :



```
class employee():
    def __init__(self,name,age,id,salary):   # creating a function
        self.name = name # self is an instance of a class
        self.age = age
        self.salary = salary
        self.id = id

emp1 = employee("sneha",22,1000,1234) #creating objects
emp2 = employee("rakshith",23,2000,2234)
print(emp1.__dict__) # Prints dictionary
```

```
{'name': 'sneha', 'age': 22, 'salary': 1234, 'id': 1000}
```

Attributes: All classes create objects, and all objects contain characteristics called attributes. Attributes represents the state of the object. Attributes may be instance attributes and class attributes.

Instance Attributes: Attributes that are specific to the objects are called instance attributes.

Class Attributes: Attributes that are same for all objects are called class attributes.

Constructors in Python: Class functions that begin with double underscore __ are called special functions with special meaning. A class constructor in Python is a special function **__init__( )** that is used to initialize an objects initial attributes by giving them their default value (or state). This method must have at least one argument as well as the self variable, which refers to the object itself as illustrated below :

```
class Dog:

    kind = 'canine'         # class variable shared by all instances

    def __init__(self, name):
        self.name = name    # instance variable unique to each instance
```

```
d = Dog('Fido')
e = Dog('Buddy')
```

```
print(d.kind ) # shared by all dogs
print(e.kind)  # shared by all dogs
print(d.name)  # unique to d
print(e.name) # unique to e
```

```
canine
canine
Fido
Buddy
```

Note: __init__() method
The __init__() method is similar to constructors in C++ and Java. Constructors are used to initialize the object's state. Like methods, a constructor also contains a collection of statements (i.e. instructions) that are executed at the time of Object creation. It is run as soon as an object of a class is instantiated. The method is useful to do any initialization you want to do with your object.

```
# A Sample class with init method
class Person:

    # init method or constructor
    def __init__(self, name):
        self.name = name

    # Sample Method
    def say_hi(self):
        print('Hello, my name is', self.name)

p = Person('Nikhil')
p.say_hi()
```

```
Hello, my name is Nikhil
```

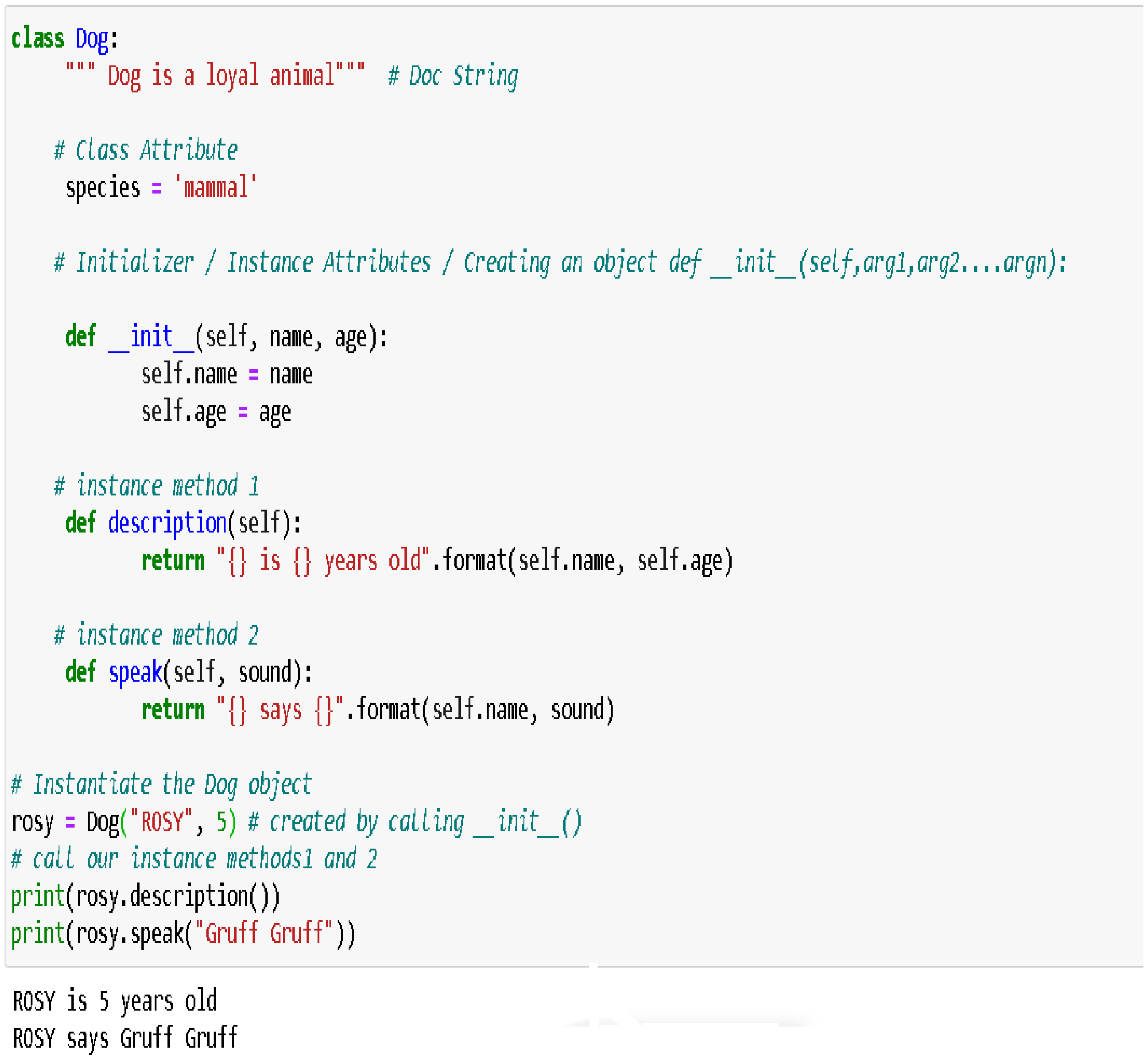
Instantiation: The process of creating this object is called instantiation. When an object of a class is created, the class is said to be instantiated. All the instances share the attributes and the behavior of the class. But the values of those ttributes, i.e. the state are unique for each object. A single class may have any number of instances.

```
# Python program to demonstrate instantiating of class
class Dog:

    # A simple class
    # attribute
    attr1 = "mamal"
    attr2 = "dog"

    # A sample method
    def fun(self):
        print("I'm a", self.attr1)
        print("I'm a", self.attr2)

# Driver code
# Object instantiation
Beagle = Dog()

# Accessing class attributes
# and method through objects
print(Beagle.attr1)
Beagle.fun()
```

```
mamal
I'm a mamal
I'm a dog
```

Instance Methods:

Instance methods are defined inside a class and are used to get the contents of an instance. They can also be used to perform operations with the attributes of our objects. Like the **init** method, the first argument is always self:



```
class Dog:
     """ Dog is a loyal animal"""  # Doc String

    # Class Attribute
     species = 'mammal'

    # Initializer / Instance Attributes / Creating an object def __init__(self,arg1,arg2....argn):

     def __init__(self, name, age):
            self.name = name
            self.age = age

    # instance method 1
     def description(self):
            return "{} is {} years old".format(self.name, self.age)

    # instance method 2
     def speak(self, sound):
            return "{} says {}".format(self.name, sound)

# Instantiate the Dog object
rosy = Dog("ROSY", 5) # created by calling __init__()
# call our instance methods1 and 2
print(rosy.description())
print(rosy.speak("Gruff Gruff"))
```

```
ROSY is 5 years old
ROSY says Gruff Gruff
```

Example 2

```python
class Employee:
    'Common base class for all employees'
    empCount = 0

    # Initializer
    def __init__(self,name,salary):
        self.name = name
        self.salary = salary
        Employee.empCount +=1

    # Instance Method
    def displayCount(self):
        print("Total Employee %d" %Employee.empCount)

    # Instance Method
    def displayEmployee(self):
        print("Name : ",self.name,"Salary: ", self.salary)
```

```python
# Creating Instance Objects
emp1 = Employee("Raju",2000)
emp2 = Employee("Swamy",5000)
emp3 = Employee("Swamy1",15000)
emp4 = Employee("Swamy2",25000)
emp5 = Employee("Swamy3",35000)
```

```python
# Accessing Attributes
emp1.displayEmployee()
emp2.displayEmployee()
emp3.displayEmployee()
emp4.displayEmployee()
emp5.displayEmployee()
```

```
Name :  Raju Salary:  2000
Name :  Swamy Salary:  5000
Name :  Swamy1 Salary:  15000
Name :  Swamy2 Salary:  25000
Name :  Swamy3 Salary:  35000
```

```
print("Total Employee %d"%Employee.empCount)
```

```
Total Employee 5
```

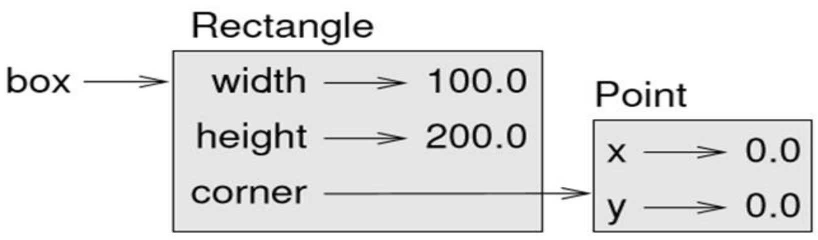
Objects / Instances as Return values:

Example: Construction of Rectangle

Rectangle can be created using following two possibilities:

1. Using one corner of the rectangle (or the center), the width, and the height.
2. Using two opposing corners.

By considering the first option the object diagram of rectangle is illustrated as below:



Object diagram.

The class definition for Rectangle ,Point and function to print Point values is as illustrated below:

```
class Rectangle:
    """ Rectangle with attributes:width ,height , corner"""
```

```
class Point:
    """ Represents a point (x,y) in 2D space"""
```

```
def print_point(P):
    print('(%d ,%d)'%(P.x,P.y))
```

Corner is the origin point with values (0,0) and width , height are numbers . Rectangle object is instantiated with values width = 100, height = 200 , corner.x = 0 and corner .y = 0 as illustrated below :

```
box   = Rectangle()
box.width = 50.00
box.height = 100.00
box.corner = Point()
box.corner.x = 0.0
box.corner.y = 0.0
```

Functions can return instances/objects. For example, find_center takes a Rectangle as an argument and returns a Point that contains the coordinates of the center of the Rectangle:

```
def find_center(rect):
    p = Point()
    p.x = rect.corner.x + rect.width/2
    p.y = rect.corner.y + rect.height/2
    return p
```

Here is an example that passes box as an argument and assigns the resulting Point to center:

```
center = find_center(box)
print_point(center)
```

```
(25 ,50)
```

Objects are Mutable

You can change the state of an object by making an assignment to one of its attributes.

For example, to change the size of a rectangle without changing its position, you can modify the values of width and height:

```
box.width = box.width + 100
box.height = box.height + 200
```

```
print(box.width)
print(box.height)
```

```
200.0
400.0
```

Modifying through Functions

```
def  grow_rectangle(rect, dwidth, dheight):
    rect.width += dwidth
    rect.height += dheight
```

```
grow_rectangle(box,200,300)
print(box.width)
print(box.height)
```

```
400.0
700.0
```

Copying

Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object. Consider a point P1 with x = 3 and y = 4 :

```
P1 = Point()
P1.x= 5.0
P1.y = 6.0
```

Copying an object is often an alternative to aliasing. The copy module contains a function called copy that can duplicate any object .A copy of point p1 can be assigned to point p2 by using copy method as illustrated below :

```
import copy
P2 = copy.copy(P1)
print_point(P1)
print_point(P2)
```

```
(5 ,6)
(5 ,6)
```

```
p1 is p2
```

```
False
```

```
p1 == p2
```

```
False
```

The **is** and **==** operator indicates that p1 and p2 are not the same object .

If you use ***copy.copy*** to duplicate a Rectangle, you will find that it copies the Rectangle object but not the embedded Point.

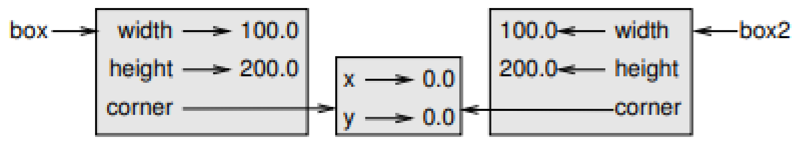


Figure shows what the object diagram looks like. This operation is called a shallow copy because it copies the object and any references it contains, but not the embedded objects.

```
# Shallow  Copy
box2 = copy.copy(box)
box2 is box
```

```
False
```

```
box2.corner is box.corner
```

```
True
```

Fortunately, the copy module provides a method named deepcopy that copies not only the object but also the objects it refers to, and the objects they refer to, and so on. You will not be surprised to learn that this operation is called a deep copy.

```
# Deep Copy
box3 = copy.deepcopy(box)
box3 is box
```

```
False
```

```
box3.corner is box.corner
```

```
False
```

Example for Modifying Attributes

```
class Email:
    def __init__(self): # Initiator
        self.is_sent = False  # Initial Value

    def send_email(self):
        self.is_sent = True  # Modified Value
```

```
my_email = Email()
my_email.is_sent  # Initial Value
```

```
False
```

```
my_email.send_email()
my_email.is_sent  # Modified Value
```

```
True
```

Deleting Attributes and Objects

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

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

    def get_data(self):
        print(f'{self.real}+{self.imag}j')
```

```
num1 = ComplexNumber(2,3)
```

```
print(num1.real,num1.imag)
```

```
2 3
```

```
del num1.imag
num1.get_data()
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-49-3f1747e66ca4> in <module>
      1 del num1.imag
----> 2 num1.get_data()

<ipython-input-44-4f23deca1c28> in get_data(self)
      5 
      6     def get_data(self):
----> 7         print(f'{self.real}+{self.imag}j')

AttributeError: 'ComplexNumber' object has no attribute 'imag'
```

2. Classes and functions

In this section we will write Functions (user define functions) that take user defined objects as parameter and return them as results. Consider a class called Time as illustrated below:

```
class Time :
    ''' Represent Time in Hour:Minutes:Seconds'''
    pass
time = Time()
time.hour =   20
time.minute = 30
time.second = 59
```

User defined Functions to display the attribute of object say ob.

```
#User defined function to display  the value of time
def print_time(time):
    print("Hours:",time.hour)
    print("Minutes:",time.minute)
    print("Seconds:",time.second)

print_time(time)
```

```
Hours: 20
Minutes: 30
Seconds: 59
```

Inorder to add time values we will consider following two kinds of functions :

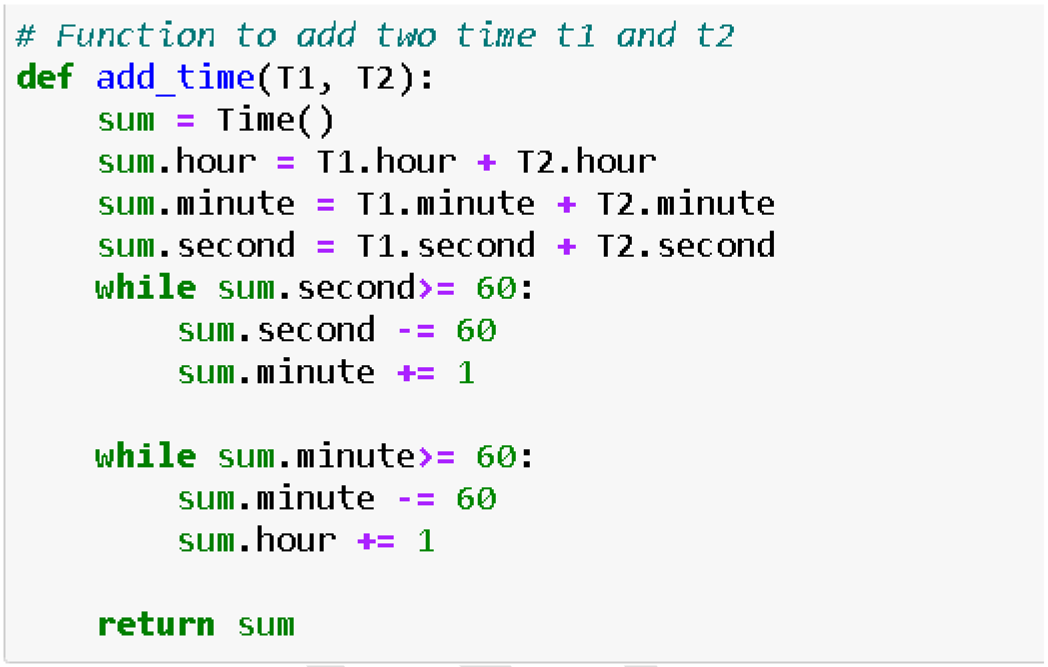
1. Pure Functions

2. Modifiers

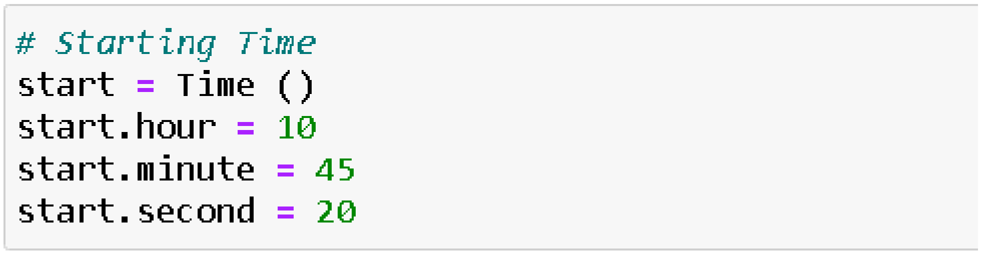
1. Pure Function:

The function which creates new object , initiates its attributes and returns a reference to the new object is called a pure function. It is called pure function because it does not modify any of the objects passed to it as arguments and it has no side effects , such as displaying value or getting user input.

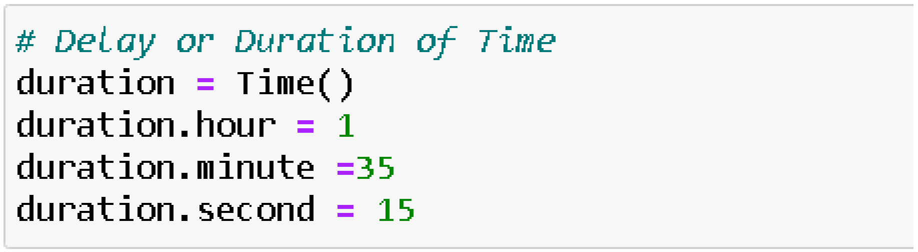
Example:



```python
# Function to add two time t1 and t2
def add_time(T1, T2):
    sum = Time()
    sum.hour = T1.hour + T2.hour
    sum.minute = T1.minute + T2.minute
    sum.second = T1.second + T2.second
    while sum.second>= 60:
        sum.second -= 60
        sum.minute += 1

    while sum.minute>= 60:
        sum.minute -= 60
        sum.hour += 1

    return sum
```



```python
# Starting Time
start = Time ()
start.hour = 10
start.minute = 45
start.second = 20
```



```python
# Delay or Duration of Time
duration = Time()
duration.hour = 1
duration.minute =35
duration.second = 15
```

```
done = add_time (start,duration)
print_time(done)
```

```
Hours: 12
Minutes: 20
Seconds: 35
```

2. Modifiers

The functions which are used to modify one or more of the objects its gets as rguments are called modifiers . Most modifiers are fruitful .

```
# Modifier
def increment (time,seconds):
    time.second = time.second + seconds
    while time.second>= 60:
        time.second -= 60
        time.minute += 1

    while time.minute>= 60:
        time.minute -= 60
        time.hour += 1
    return time
```

```
incr_time = increment(time,200)
print_time(incr_time)
```

```
Hours: 20
Minutes: 34
Seconds: 19
```

Approaches for Program Development

1. Prototype Development
2. Planned Development

1. Prototype Development: This approach can be effective, especially if programmer have a deep understanding of the problem. But incremental corrections can generate code that is unnecessarily complicated – since it deals with many special cases and unreliable since it is hard to find all errors. Prototype development involves

1. *Writing a Rough draft of programs (or prototype)*
2. *Performing a basic calculation*
3. *Testing on a few cases, correcting flaws as we found them.*
4. Prototype development leads code complicated

2. Planned Development: High level insight into the problem can make the programming much easier

Example: A time object is really a three digit number in base 60.[hour: minute(60X60): second(60)] . The second component is the "Ones Column" the minute component is the "sixties column" and the hour component is the thirty six hundred column. When we wrote add_time and increment, we were effectively doing addition in base 60, which is why we had to carry from one column to the next. This observation suggests another approach to the whole problem—we can convert Time objects to integers and take advantage of the fact that the computer knows how to do integer arithmetic. Here is a function that converts Times to integers:

```
def time_to_int(time):
    minutes = time.hour * 60 + time.minute
    seconds = minutes * 60 + time.second
    return seconds
```

And here is the function that converts integers to Times (recall that divmod divides the first argument by the second and returns the quotient and remainder as a tuple).

```python
def int_to_time(seconds):
    time = Time()
    minutes, time.second = divmod(seconds, 60)
    time.hour, time.minute = divmod(minutes, 60)
    return time
```

You might have to think a bit, and run some tests, to convince yourself that these functions are correct. One way to test them is to check that time_to_int (int_to_time(x)) == x for many values of x. This is an example of a consistency check. Once you are convinced they are correct, you can use them to rewrite add_time:

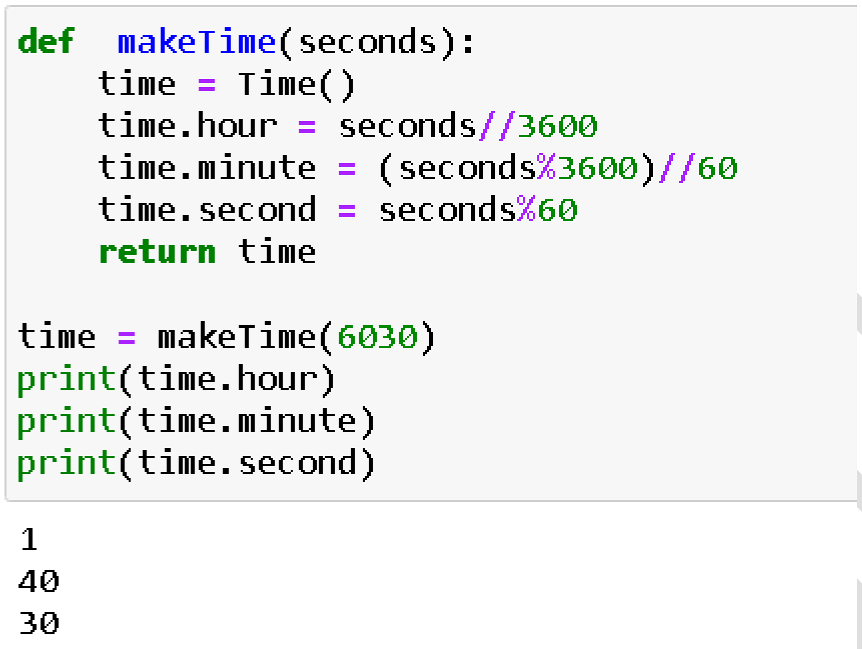
```python
def add_time(t1, t2):
    seconds = time_to_int(t1) + time_to_int(t2)
    return int_to_time(seconds)
```

Example 1: Converting time to seconds

```python
def convertToSeconds(t):
    minutes = t.hour*60 + t.minute
    seconds = t.minute *60 + t.second
    return seconds

time = Time()
time.hour = 10
time.minute = 30
time.second = 50

sec = convertToSeconds(time)
print(sec)
```

```
1850
```

Example2 : Converting Seconds into Hours , Minutes and Seconds



```
def  makeTime(seconds):
    time = Time()
    time.hour = seconds//3600
    time.minute = (seconds%3600)//60
    time.second = seconds%60
    return time

time = makeTime(6030)
print(time.hour)
print(time.minute)
print(time.second)
```

```
1
40
30
```

3. Classes and Methods

Object Oriented Features: Python is an object oriented programming language which deals with declaring Python classes and objects which lays foundation of OOP Concepts. Some of the characteristics that object oriented programming language supports are :

- Programs include class and method definitions.
- Most of the computation is expressed in terms of operations on objects.
- Objects often represent things in the real world, and methods often correspond to the ways things in the real world interact

For example, the Time class defined corresponds to the way people record the time of day, and the functions we defined correspond to the kinds of things people do with times. Similarly, the Point and Rectangle classes correspond to the mathematical concepts of a point and a rectangle.

Methods are functions that are associated with a particular class. In this section we will define methods for programmer defined types. Methods are semantically same as functions with following two differences :

- Methods are define inside a class definition in order to make the relationship between class and the method explicit
- The syntax for invoking a method is different from the syntax for calling a function.

Transforming Function into methods: Functions can be transformed into methods by moving function inside the class definition with proper indentation.

Example1: Time Class with print_time () method

```
class Time:
    """Represents the time of day."""
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
```

```
start = Time()
start.hour = 9
start.minute = 45
start.second = 00
print_time(start)
```

```
Hours: 9
Minutes: 45
Seconds: 0
```

```
Time.print_time(start)
```

```
09:45:00
```

```
start.print_time()
```

```
09:45:00
```

Example 2: Time class with methods *print_time() , increment() and time_to_int().*

```
# inside
class Time:
    """Represents the time of day."""
    def print_time(time):
        print('%.2d:%.2d:%.2d' % (time.hour, time.minute, time.second))
    def increment(self, seconds):
        seconds += self.time_to_int()
        return int_to_time(seconds)
    def time_to_int(time):
        minutes = time.hour * 60 + time.minute
        seconds = minutes * 60 + time.second
        return seconds
```

```
start = Time()
start.hour = 9
start.minute = 45
start.second = 00
print_time(start)
```

```
Hours: 9
Minutes: 45
Seconds: 0
```

```
start.print_time()
```

```
09:45:00
```

```
end = start.increment(1337)
end.print_time()
```

```
10:07:17
```

Optional Arguments

In python we can write user defined functions with optional argument lists.

```
def find(str, ch, start =0): # start is the optional argument
    index = start
    while index <len(str):
        if str[index] == ch:
            return index
        index = index +1
    return -1
```

The third parameter start is optional because a default value 0 is provided. If we invoke find with only two arguments it uses the default value and start from the beginning of the string:

```
find("apple","p")
```

```
1
```

```
find("apple","p",2)
```

```
2
```

Initialization Method : __ init__()

The initialization method is a special method that is invoked when an object is created.The name of this method is :__init__() (two underscore characters followed by init and then two more underscore)

Example : init method for the Time Class

```
# inside
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
    def print_time(time):
        print (str(time.hour) + ":" + str(time.minute) + ":"  + str(time.second))
```

```
#Inovoking a method
currentTime = Time(9,14,30)
currentTime.print_time()
```

```
9:14:30
```

The __str__method :

__str__ is a special method , like __init__ that is supposed to return a string representation of an object.

Example : str method for Time Class

```
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
```

```
time = Time(9, 45)
print(time)
```

```
09:45:00
```

Method Overriding:

If a class provides a method __str__ it overrides the default behavior of the Python built in str function .

```
# Method overriding
class Point:
    def __init__(self, x=0,y=0):
        self.x = x
        self.y = y
    def __str__(self):
        return '('+str(self.x)+','+str(self.y)+')'
```

```
P = Point(3,4)
str(P)
```

```
'(3,4)'
```

Operator Overloading

Operator overloading is one of the feature of OOP language which make possible to change the definition of the build in operators when they are applied to user defined type. It is especially useful when defining new mathematical types. For example one can use a method name __add__ for the Time class to use + operator on Time objects as illustrated below:

```
## Method Overloadig
class Time:
    def __init__(self, hour=0, minute=0, second=0):
        self.hour = hour
        self.minute = minute
        self.second = second
    def __str__(self):
        return '%.2d:%.2d:%.2d' % (self.hour, self.minute, self.second)
    def time_to_int(time):
        minutes = time.hour * 60 + time.minute
        seconds = minutes * 60 + time.second
        return seconds
    def __add__(self, other):
        seconds = self.time_to_int() + other.time_to_int()
        return int_to_time(seconds)
```

```
start = Time(9, 45)
duration = Time(1, 35)
print(start + duration)
```

```
11:20:00
```

Example 2: Addition of two points

```
start = Time(9, 45)
duration = Time(1, 35)
print(start + duration)
```

```
11:20:00
```

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

```
P1 = Point(3,4)
```

```
P2 = Point(5,7)
```

```
P3 = P1 + P2
```

```
print(P3.x,P3.y)
```

```
8 11
```

Polymorphism

A polymorphic function can operate on more than one type .If all the operations in a function can be applied to a type, then the entire function can also be applied to a type.

Example 1 : Operator + Can be applied for numbers and Point object as illustrated below

Case 1: + and * applied for number type

```
def multadd(x, y, z):
    return (x*y + z)
multadd(3,2.0,10)
```

```
16.0
```

Case 2 : * and + operators applied for Point objects

```
class Point:
    def __init__(self,x,y):
        self.x = x
        self.y = y
    def __add__(self,other):
           return Point(self.x + other.x, self.y + other.y)
    def __mul__(self,other):
        return Point(self.x * other.x, self.y * other.y)
    def __rmul__(self, other):
        return Point(other * self.x,  other * self.y)
```

```
P1 = Point(3,4)
P2 = Point(5,7)
P3 = P1 + P2
```

```
print(P3.x,P3.y)
```

```
8 11
```

```
P4= P1*P2
```

```
P=multadd(2,P1,P2)
print(P.x,P.y)
```

```
11 15
```

Example3 : Function histogram () works with strings , lists , tuples and even dictionaries as illustrated below :

```
def histogram(s):
    d = dict()
    for c in s:
        if c not in d:
            d[c] = 1
        else:
            d[c] = d[c]+1
    return d
```

```
histogram("Word")  # For Strings
```

```
{'W': 1, 'o': 1, 'r': 1, 'd': 1}
```

```
t = ['spam', 'egg', 'spam', 'spam', 'bacon', 'spam']
histogram(t) # For Tuples
```

```
{'spam': 4, 'egg': 1, 'bacon': 1}
```

Python Date Time

Python has a module named datetime to work with dates and times. Let's create a few simple programs related to date and time

```
# To print time
import datetime
time_now = datetime.datetime.now()
print(time_now)
```

```
2020-11-16 00:19:13.056952
```

```
# To print date Time
import datetime
date_now = datetime.date.today()
print(date_now)
```

```
2020-11-16
```

```
# To print Year , Month and Day
from datetime import date
td = date.today()
print("Current Year:",td.year)
print("Current Month:",td.month)
print("Current Day:",td.day)
```

```
Current Year: 2020
Current Month: 11
Current Day: 16
```

Handling Time Zone

```
# To print Year , Month and Day
from datetime import date
td = date.today()
print("Current Year:",td.year)
print("Current Month:",td.month)
print("Current Day:",td.day)
```

```
Current Year: 2020
Current Month: 11
Current Day: 16
```

```python
# Handlin Time Zone
from datetime import datetime
import pytz
local = datetime.now()
print("LOCAL:",local)
```

```
LOCAL: 2020-11-16 00:21:21.266288
```

```python
tz_NY = pytz.timezone('America/New_York')
datetime_NY = datetime.now(tz_NY)
print("NY:",datetime_NY.strftime("%m/%d/%Y,%H:%M:%S"))
```

```
NY: 11/15/2020,13:51:39
```

calendar (Year ,w,l,c)

This function shows the year, width of characters, no. of lines per week and column separations.

```python
import calendar
print("The calendar of 2020 is : ")
print(calendar.calendar(2020,1,1,4))
```

```
The calendar of 2020 is :
                                2020

      January                   February                    March
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
       1  2  3  4  5                      1  2                         1
 6  7  8  9 10 11 12       3  4  5  6  7  8  9       2  3  4  5  6  7  8
13 14 15 16 17 18 19      10 11 12 13 14 15 16       9 10 11 12 13 14 15
20 21 22 23 24 25 26      17 18 19 20 21 22 23      16 17 18 19 20 21 22
27 28 29 30 31            24 25 26 27 28 29         23 24 25 26 27 28 29
                                                    30 31

       April                      May                       June
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
       1  2  3  4  5                   1  2  3       1  2  3  4  5  6  7
 6  7  8  9 10 11 12       4  5  6  7  8  9 10       8  9 10 11 12 13 14
13 14 15 16 17 18 19      11 12 13 14 15 16 17      15 16 17 18 19 20 21
20 21 22 23 24 25 26      18 19 20 21 22 23 24      22 23 24 25 26 27 28
27 28 29 30               25 26 27 28 29 30 31      29 30

        July                     August                   September
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
       1  2  3  4  5                      1  2          1  2  3  4  5  6
 6  7  8  9 10 11 12       3  4  5  6  7  8  9       7  8  9 10 11 12 13
13 14 15 16 17 18 19      10 11 12 13 14 15 16      14 15 16 17 18 19 20
20 21 22 23 24 25 26      17 18 19 20 21 22 23      21 22 23 24 25 26 27
27 28 29 30 31            24 25 26 27 28 29 30      28 29 30
                          31

      October                   November                  December
Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su      Mo Tu We Th Fr Sa Su
          1  2  3  4                         1          1  2  3  4  5  6
 5  6  7  8  9 10 11       2  3  4  5  6  7  8       7  8  9 10 11 12 13
12 13 14 15 16 17 18       9 10 11 12 13 14 15      14 15 16 17 18 19 20
```

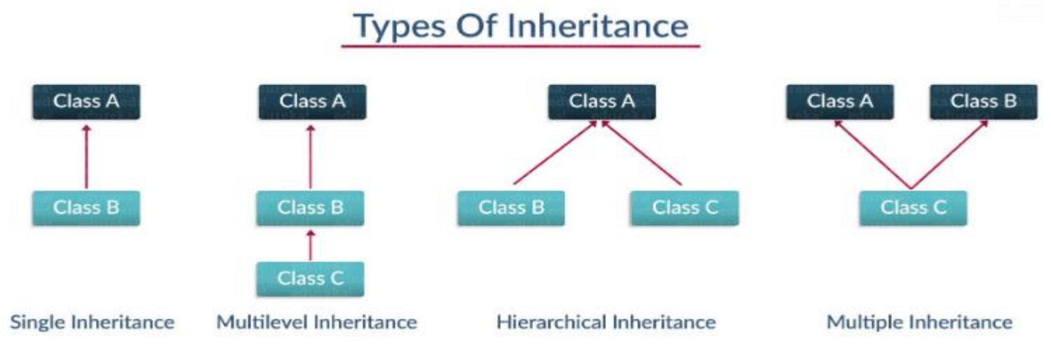
```
19 20 21 22 23 24 25    16 17 18 19 20 21 22    21 22 23 24 25 26 27
26 27 28 29 30 31       23 24 25 26 27 28 29    28 29 30 31
                        30
```

Object-Oriented Programming methodologies:

Object-Oriented Programming methodologies deal with the following four pillars

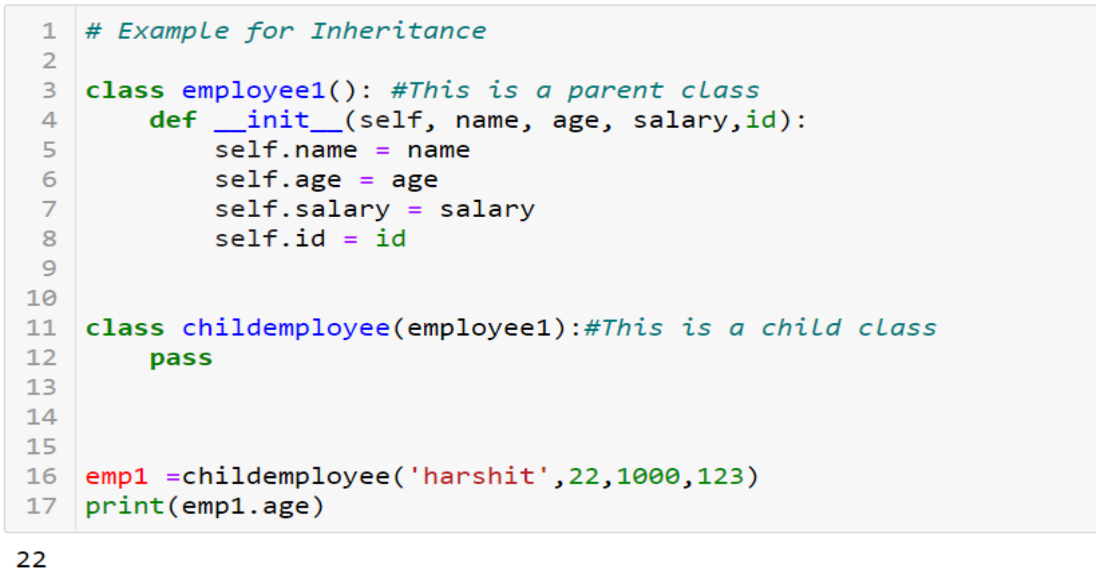
1. Inheritance
2. Polymorphism
3. Encapsulation
4. Abstraction

1. **Inheritance:** Inheritance generally means *"inheriting or transfer of characteristics from parent to child class without any modification"*. The new class is called the **derived/child** class and the one from which it is derived is called a **parent/base** class.



Single Inheritance: Single level inheritance enables a derived class to inherit characteristics from a single parent class.

Example:



```
# Example for Inheritance

class employee1(): #This is a parent class
    def __init__(self, name, age, salary,id):
        self.name = name
        self.age = age
        self.salary = salary
        self.id = id


class childemployee(employee1):#This is a child class
    pass



emp1 =childemployee('harshit',22,1000,123)
print(emp1.age)
```

```
22
```

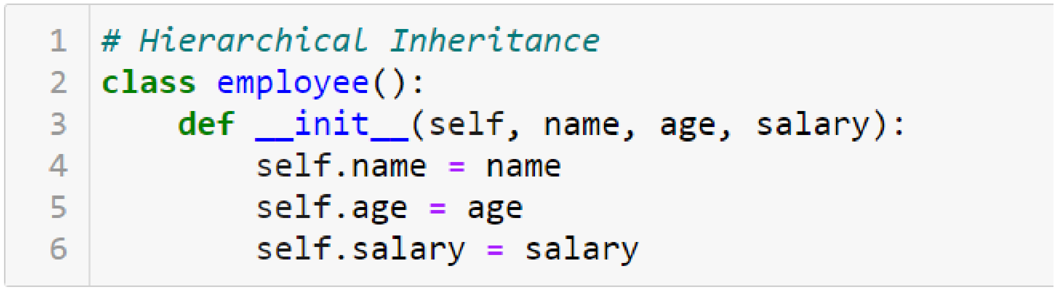
Multilevel Inheritance: Multi-level inheritance enables a derived class to inherit properties from an immediate parent class which in turn inherits properties from his parent class.

Example:

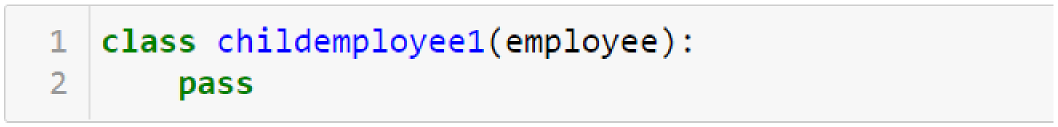
```python
# Example for Multi level Inheritance
class employee():# Super class
    def __init__(self,name,age,salary):
        self.name = name
        self.age = age
        self.salary = salary

# First child class
class childemployee1(employee):
    pass


# Second childclass
class childemployee2(childemployee1):
    pass


emp1 = childemployee1('harshit',22,1000)
emp2 = childemployee2('arjun',23,2000)

print(emp1.age)
print(emp2.age)
```

```
22
23
```

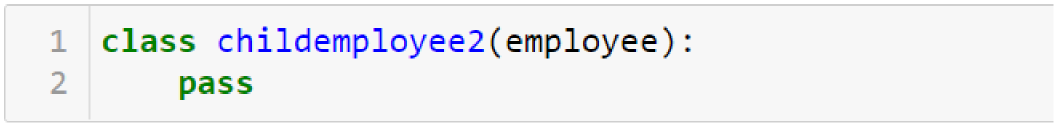
Hierarchical Inheritance: Hierarchical level inheritance enables more than one derived class to inherit properties from a parent class.



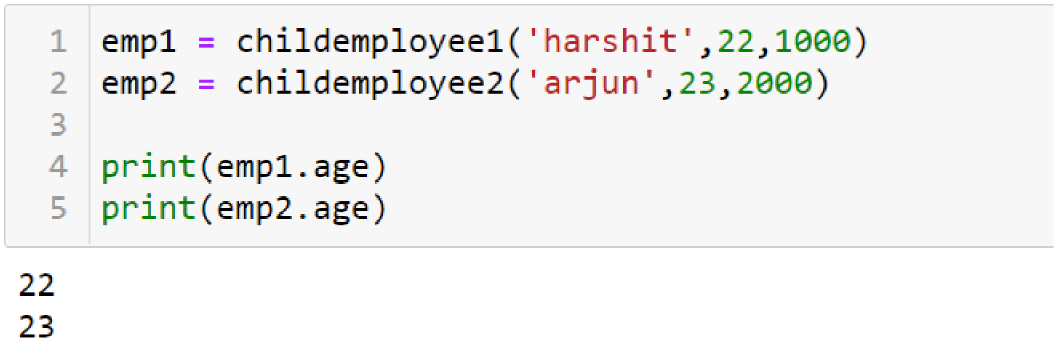
```
# Hierarchical Inheritance
class employee():
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary
```



```
class childemployee1(employee):
    pass
```



```
class childemployee2(employee):
    pass
```



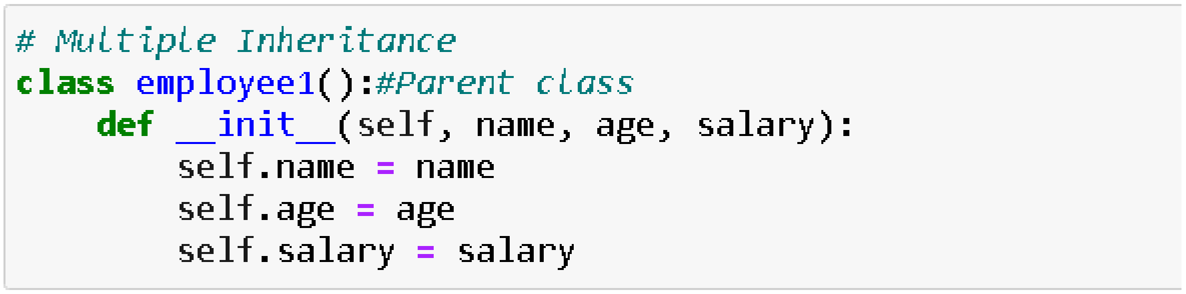
```
emp1 = childemployee1('harshit',22,1000)
emp2 = childemployee2('arjun',23,2000)

print(emp1.age)
print(emp2.age)
```

```
22
23
```

Multiple Inheritance:

Multiple level inheritance enables one derived class to inherit properties from more than one base class.

Example:



```
# Multiple Inheritance
class employee1():#Parent class
    def __init__(self, name, age, salary):
        self.name = name
        self.age = age
        self.salary = salary
```

```python
class employee2():#Parent class
    def __init__(self,name,age,salary,id):
        self.name = name
        self.age = age
        self.salary = salary
        self.id = id
```

```python
class childemployee(employee1,employee2):
    def __init__(self, name, age, salary,id):
        self.name = name
        self.age = age
        self.salary = salary
        self.id = id
```

```python
emp1 = employee1('harshit',22,1000)
emp2 = employee2('arjun',23,2000,1234)
```

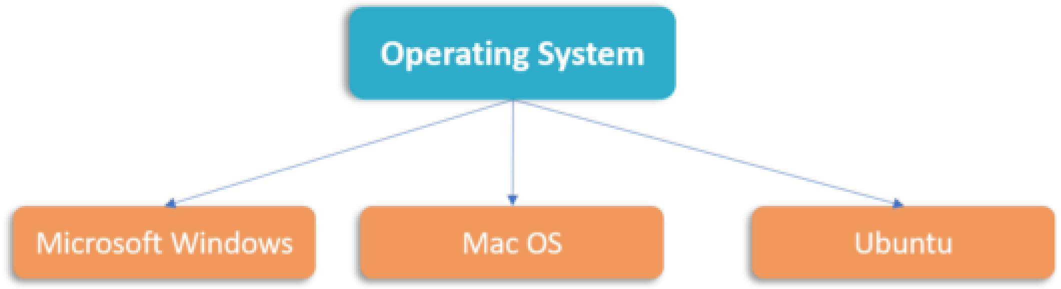
```python
print(emp1.age)
print(emp2.id)
```

```
22
1234
```

https://www.edureka.co/blog/object-oriented-programming-python/

Polymorphism:

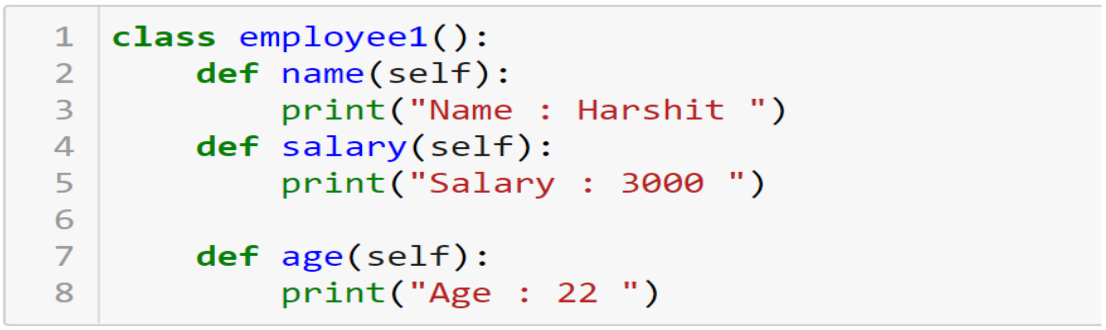
Polymorphism is a OOP methodology where one task can be performed in several different ways. I*t is a property of an object which allows it to take multiple forms*.



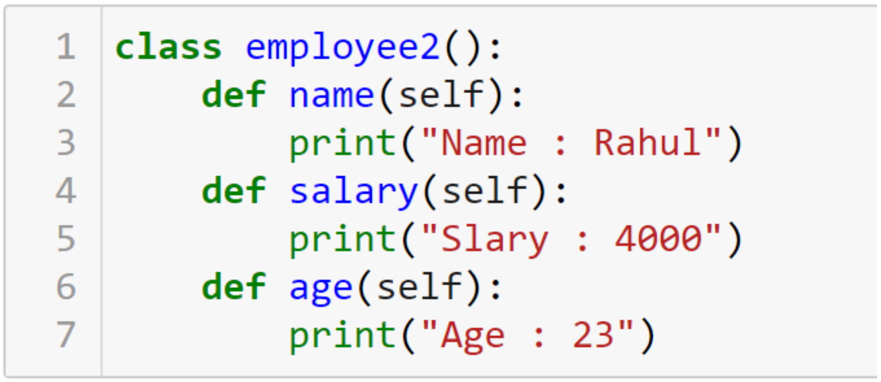
Polymorphism is of two types:

- *Compile-time Polymorphism*
- *Run-time Polymorphism*

Compile-time Polymorphism:
A compile-time polymorphism also called as static polymorphism which gets resolved during the compilation time of the program. One common example is "method overloading". Let me show you a quick example of the same.



```python
class employee1():
    def name(self):
        print("Name : Harshit ")
    def salary(self):
        print("Salary : 3000 ")

    def age(self):
        print("Age : 22 ")
```



```python
class employee2():
    def name(self):
        print("Name : Rahul")
    def salary(self):
        print("Slary : 4000")
    def age(self):
        print("Age : 23")
```

```python
def func(obj):# Method Overloading
          obj.name()
          obj.salary()
          obj.age()
```

```python
func(obj_emp1)
func(obj_emp2)
```

```
Name : Harshit
Salary : 3000
Age : 22
Name : Rahul
Slary : 4000
Age : 23
```

Explanation:

- In the above Program, I have created two classes 'employee1' and 'employee2' and created functions for both 'name', 'salary' and 'age' and printed the value of the same without taking it from the user.
- Now, welcome to the main part where I have created a function with 'obj' as the parameter and calling all the three functions i.e. 'name', 'age' and 'salary'.
- Later, instantiated objects emp_1 and emp_2 against the two classes and simply called the function. Such type is called method overloading which allows a class to have more than one method under the same name.

Run-time Polymorphism:

A run-time Polymorphism is also, called as dynamic polymorphism where it gets resolved into the run time. One common example of Run-time polymorphism is "method overriding". Let me show you through an example for a better understanding.

```python
class employee():
    def __init__(self,name,age,id,salary):
        self.name = name
        self.age = age
        self.salary = salary
        self.id = i
    def earn(self):
        pass
```

```python
class childemployee1(employee):
    def earn(self):#Run-time polymorphism
        print("no money")
```

```python
class childemployee2(employee):
    def earn(self):
        print("has money")
```

```python
c = childemployee1
c.earn(employee)
d = childemployee2
d.earn(employee)
```

```
no money
has money
```

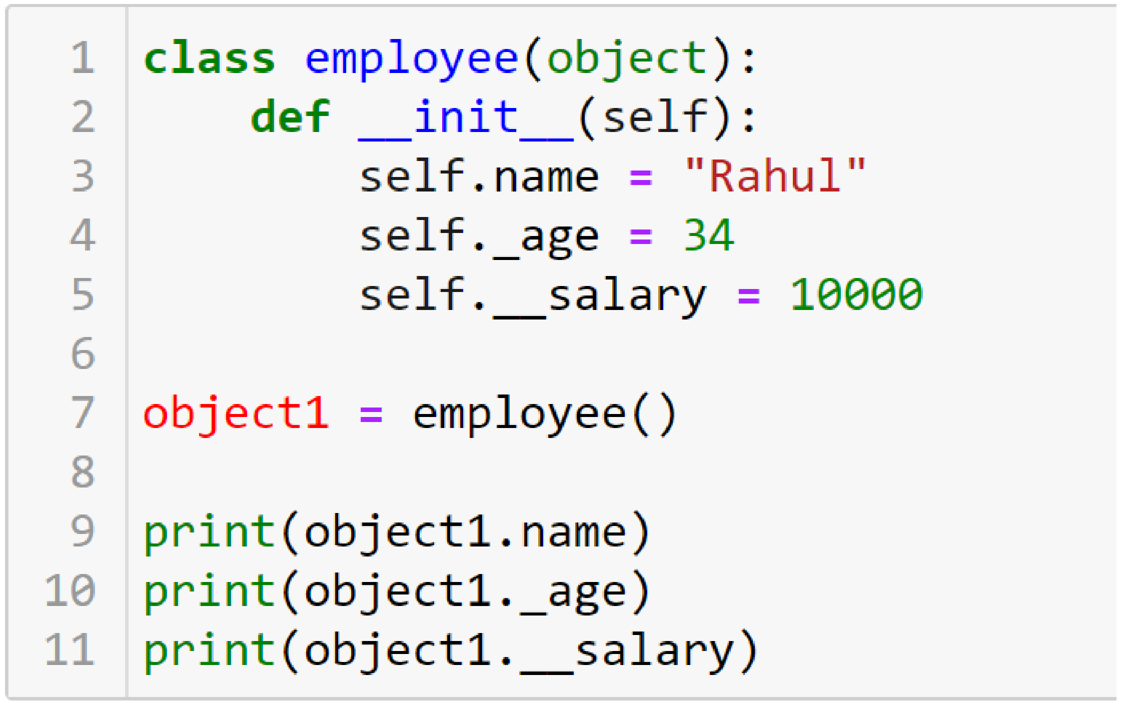
Explanation: In the above example, I have created two classes ‘childemployee1’ and ‘childemployee2’ which are derived from the same base class ‘employee’.Here’s the catch one did not receive money whereas the other one gets. Now the real question is how did this happen? Well, here if you look closely I created an empty function and used *Pass* (a statement which is used when you do not want to execute any command or code). Now, Under the two derived classes, I used the same empty function and made use of the print statement as ‘no money’ and ‘has money’.Lastly, created two objects and called the function.

Moving on to the next Object-Oriented Programming Python methodology, I’ll talk about encapsulation.

Encapsulation:
In a raw form, encapsulation basically means binding up of data in a single class. Python does not have any private keyword, unlike Java. A class shouldn’t be directly accessed but be prefixed in an underscore.

Let me show you an example for a better understanding.

Example:



```python
class employee(object):
    def __init__(self):
        self.name = "Rahul"
        self._age = 34
        self.__salary = 10000

object1 = employee()

print(object1.name)
print(object1._age)
print(object1.__salary)
```

```
Rahul
34

---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-18-a327bfb3045e> in <module>
      9 print(object1.name)
     10 print(object1._age)
---> 11 print(object1.__salary)

AttributeError: 'employee' object has no attribute '__salary'
```

Explanation: You will get this question what is the underscore and error? Well, python class treats the private variables as(__salary) which can not be accessed directly.

So, I have made use of the setter method which provides indirect access to them in my next example.

Example:

```python
class employee():
    def __init__(self):
        self.__maxearn = 1000000

    def earn(self):
        print("earning is:{}".format(self.__maxearn))

    # setter method used for accesing private class
    def setmaxearn(self,earn):#
        self.__maxearn = earn

emp1 = employee()
emp1.earn()

emp1.__maxearn = 10000
emp1.earn()

emp1.setmaxearn(10000)
emp1.earn()
```

```
earning is:1000000
earning is:1000000
earning is:10000
```

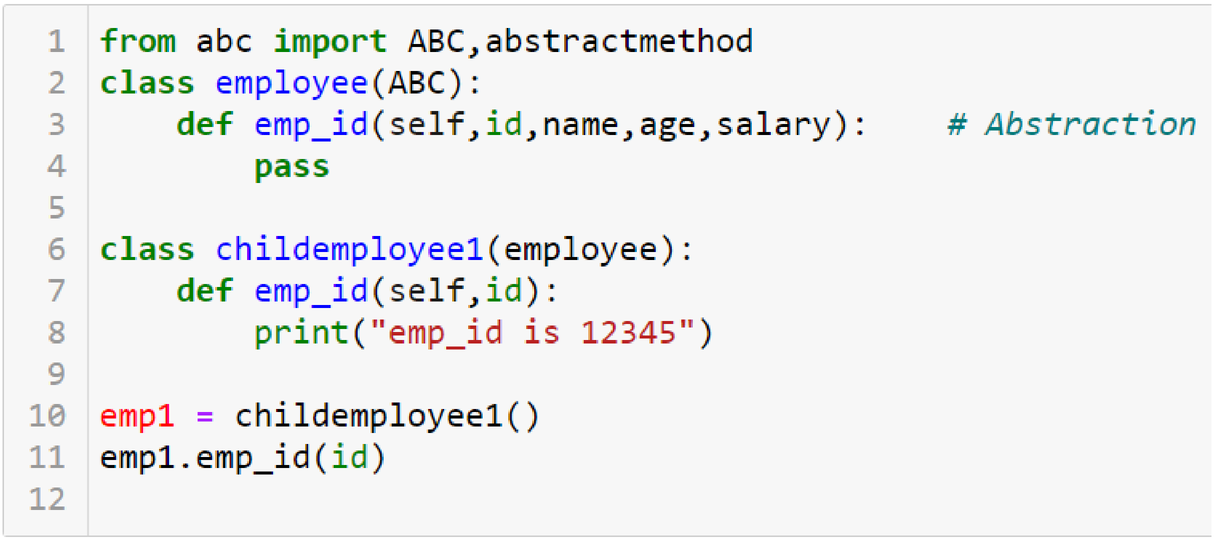
Explanation: Making Use of the **setter method** provides *indirect access to the private class method*. Here I have defined a class employee and used a (__maxearn) which is the setter method used here to store the maximum earning of the employee, and a setter function setmaxearn() which is taking price as the parameter.

This is a clear example of encapsulation where we are restricting the access to private class method and then use the setter method to grant access.

Next up in object-oriented programming python methodology talks about one of the key concepts called abstraction.

Abstraction: Suppose you booked a movie ticket from bookmyshow using net banking or any other process. You don't know the procedure of how the pin is generated or how the verification is done. This is called 'abstraction' from the programming aspect, it basically means you only show the implementation details of a particular process and hide the details from the user. It is used to simplify complex problems by modeling classes appropriate to the problem.

An abstract class cannot be instantiated which simply means you cannot create objects for this type of class. It can only be used for inheriting the functionalities.



```
from abc import ABC,abstractmethod
class employee(ABC):
    def emp_id(self,id,name,age,salary):    # Abstraction
        pass

class childemployee1(employee):
    def emp_id(self,id):
        print("emp_id is 12345")

emp1 = childemployee1()
emp1.emp_id(id)

```

```
emp_id is 12345
```

Explanation: As you can see in the above example, we have imported an abstract method and the rest of the program has a parent and a derived class. An object is instantiated for the 'childemployee' base class and functionality of abstract is being used.

Sets of Objects :

Having seen this pattern, and having learned about lists and objects, you should not be surprised to learn that you can create lists of objects. You can also create objects that contain lists (as attributes); you can create lists that contain lists; you can create objects that contain objects; and so on.

Card Objects :

If you are not familiar with common playing cards, now would be a good time to get a deck, or else this chapter might not make much sense. There are fifty-two cards in a deck, each of which belongs to one of four suits and one of thirteen ranks. The suits are Spades, Hearts, Diamonds, and Clubs (in descending order in bridge). The ranks are Ace, 2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, and King. Depending on the game that you are playing, the rank of Ace may be higher than King or lower than 2.

If we want to define a new object to represent a playing card, it is obvious what the attributes should be: `rank` and `suit`. It is not as obvious what type the attributes should be. One possibility is to use strings containing words like `"Spade"` for suits and `"Queen"` for ranks. One problem with this implementation is that it would not be easy to compare cards to see which had a higher rank or suit.

An alternative is to use integers to **encode** the ranks and suits. By encode, we do not mean what some people think, which is to encrypt or translate into a secret code. What a computer scientist means by encode is to define a mapping between a sequence of numbers and the items I want to represent. For example:

```
Spades   -->  3
Hearts   -->  2
Diamonds -->  1
Clubs    -->  0
```

An obvious feature of this mapping is that the suits map to integers in order, so we can compare suits by comparing integers. The mapping for ranks is fairly obvious; each of the numerical ranks maps to the corresponding integer, and for face cards:

```
Jack   -->  11
Queen  -->  12
King   -->  13
```

The reason we are using mathematical notation for these mappings is that they are not part of the Python program. They are part of the program design, but they never appear explicitly in the code. The class definition for the `Card` type looks like this:

```
class Card:
    def __init__(self, suit=0, rank=0):
        self.suit = suit
        self.rank = rank
```

As usual, we provide an initialization method that takes an optional parameter for each attribute.

To create an object that represents the 3 of Clubs, use this command:

```
three_of_clubs = Card(0, 3)
```

The first argument, `0`, represents the suit Clubs.

Class attributes and the `__str__` method

In order to print `Card` objects in a way that people can easily read, we want to map the integer codes onto words. A natural way to do that is with lists of strings. We assign these lists to class attributes at the top of the class definition:

```
class Card:
    suits = ["Clubs", "Diamonds", "Hearts", "Spades"]
    ranks = ["narf", "Ace", "2", "3", "4", "5", "6", "7",
             "8", "9", "10", "Jack", "Queen", "King"]

    def __init__(self, suit=0, rank=0):
        self.suit = suit
        self.rank = rank

    def __str__(self):
        return (self.ranks[self.rank] + " of " + self.suits[self.suit])
```

A class attribute is defined outside of any method, and it can be accessed from any of the methods in the class.

Inside `__str__`, we can use `suits` and `ranks` to map the numerical values of `suit` and `rank` to strings. For example, the expression `self.suits[self.suit]` means use the attribute `suit` from the object `self` as an index into the class attribute named `suits`, and select the appropriate string.

The reason for the `"narf"` in the first element in `ranks` is to act as a place keeper for the zero-eth element of the list, which will never be used. The only valid ranks are 1 to 13. This wasted item is not entirely necessary. We could have started at 0, as usual, but it is less confusing to encode 2 as 2, 3 as 3, and so on.

With the methods we have so far, we can create and print cards:

```
>>> card1 = Card(1, 11)
>>> print card1
Jack of Diamonds
```

Class attributes like `suits` are shared by all `Card` objects. The advantage of this is that we can use any `Card` object to access the class attributes:

```
>>> card2 = Card(1, 3)
>>> print card2
3 of Diamonds
>>> print card2.suits[1]
Diamonds
```

The disadvantage is that if we modify a class attribute, it affects every instance of the class. For example, if we decide that Jack of Diamonds should really be called Jack of Swirly Whales, we could do this:

```
>>> card1.suits[1] = "Swirly Whales"
>>> print card1
Jack of Swirly Whales
```

The problem is that all of the Diamonds just became Swirly Whales:

```
>>> print card2
3 of Swirly Whales
```

It is usually not a good idea to modify class attributes.

Comparing cards

For primitive types, there are conditional operators (`<`, `>`, `==`, etc.) that compare values and determine when one is greater than, less than, or equal to another. For user-defined types, we can override the behavior of the built-in operators by providing a method named `__cmp__`. By convention, `__cmp__` takes two parameters, `self` and `other`, and returns 1 if the first object is greater, -1 if the second object is greater, and 0 if they are equal to each other.

Some types are completely ordered, which means that you can compare any two elements and tell which is bigger. For example, the integers and the floating-point numbers are completely ordered. Some sets are unordered, which means that there is no meaningful way to say that one element is bigger than another. For example, the fruits are unordered, which is why you cannot compare apples and oranges.

The set of playing cards is partially ordered, which means that sometimes you can compare cards and sometimes not. For example, you know that the 3 of Clubs is higher than the 2 of Clubs, and the 3 of Diamonds is higher than the 3 of Clubs. But which is better, the 3 of Clubs or the 2 of Diamonds? One has a higher rank, but the other has a higher suit.

In order to make cards comparable, you have to decide which is more important, rank or suit. To be honest, the choice is arbitrary. For the sake of choosing, we will say that suit is more important, because a new deck of cards comes sorted with all the Clubs together, followed by all the Diamonds, and so on.

With that decided, we can write `__cmp__`:

```python
def __cmp__(self, other):
    # check the suits
    if self.suit > other.suit: return 1
    if self.suit < other.suit: return -1
    # suits are the same... check ranks
    if self.rank > other.rank: return 1
    if self.rank < other.rank: return -1
    # ranks are the same... it's a tie
    return 0
```

In this ordering, Aces appear lower than Deuces (2s).

Decks

Now that we have objects to represent `Card`s, the next logical step is to define a class to represent a `Deck`. Of course, a deck is made up of cards, so each `Deck` object will contain a list of cards as an attribute.

The following is a class definition for the `Deck` class. The initialization method creates the attribute `cards` and generates the standard set of fifty-two cards:

```
class Deck:
    def __init__(self):
        self.cards = []
        for suit in range(4):
            for rank in range(1, 14):
                self.cards.append(Card(suit, rank))
```

The easiest way to populate the deck is with a nested loop. The outer loop enumerates the suits from 0 to 3. The inner loop enumerates the ranks from 1 to 13. Since the outer loop iterates four times, and the inner loop iterates thirteen times, the total number of times the body is executed is fifty-two (thirteen times four). Each iteration creates a new instance of `Card` with the current suit and rank, and appends that card to the `cards` list.

The `append` method works on lists but not, of course, tuples.

Printing the deck

As usual, when we define a new type of object we want a method that prints the contents of an object. To print a `Deck`, we traverse the list and print each `Card`:

```
class Deck:

    ...

    def print_deck(self):

        for card in self.cards:

            print card
```

Here, and from now on, the ellipsis (`...`) indicates that we have omitted the other methods in the class.

As an alternative to `print_deck`, we could write a `__str__` method for the `Deck` class. The advantage of `__str__` is that it is more flexible. Rather than just printing the contents of the object, it generates a string representation that other parts of the program can manipulate before printing, or store for later use.

Here is a version of `__str__` that returns a string representation of a `Deck`. To add a bit of pizzazz, it arranges the cards in a cascade where each card is indented one space more than the previous card:

```
class Deck:

    ...

    def __str__(self):

        s = ""

        for i in range(len(self.cards)):

            s = s + " " * i + str(self.cards[i]) +
"\n"

        return s
```

This example demonstrates several features. First, instead of traversing `self.cards` and assigning each card to a variable, we are using `i` as a loop variable and an index into the list of cards.

Second, we are using the string multiplication operator to indent each card by one more space than the last. The expression `" " * i` yields a number of spaces equal to the current value of `i`.

Third, instead of using the `print` command to print the cards, we use the `str` function. Passing an object as an argument to `str` is equivalent to invoking the `__str__` method on the object.

Finally, we are using the variable `s` as an accumulator. Initially, `s` is the empty string. Each time through the loop, a new string is generated and concatenated with the old value of `s` to get the new value. When the loop ends, `s` contains the complete string representation of the `Deck`, which looks like this:

```
>>> deck = Deck()
>>> print deck
Ace of Clubs
 2 of Clubs
  3 of Clubs
   4 of Clubs
     5 of Clubs
       6 of Clubs
        7 of Clubs
         8 of Clubs
          9 of Clubs
           10 of Clubs
            Jack of Clubs
             Queen of Clubs
              King of Clubs
               Ace of Diamonds
```

And so on. Even though the result appears on 52 lines, it is one long string that contains newlines.

Shuffling the deck

If a deck is perfectly shuffled, then any card is equally likely to appear anywhere in the deck, and any location in the deck is equally likely to contain any card.

To shuffle the deck, we will use the `randrange` function from the `random` module. With two integer arguments, `a` and `b`, `randrange` chooses a random integer in the range `a <= x < b`. Since the upper bound is strictly less than `b`, we can use the length of a list as the second parameter, and we are guaranteed to get a legal index. For example, this expression chooses the index of a random card in a deck:

```
random.randrange(0, len(self.cards))
```

An easy way to shuffle the deck is by traversing the cards and swapping each card with a randomly chosen one. It is possible that the card will be swapped with itself, but that is fine. In fact, if we precluded that possibility, the order of the cards would be less than entirely random:

```
class Deck:

    ...

    def shuffle(self):

        import random

        num_cards = len(self.cards)

        for i in range(num_cards):

            j = random.randrange(i, num_cards)

            self.cards[i], self.cards[j] =
self.cards[j], self.cards[i]
```

Rather than assume that there are fifty-two cards in the deck, we get the actual length of the list and store it in `num_cards`.

For each card in the deck, we choose a random card from among the cards that haven't been shuffled yet. Then we swap the current card (`i`) with the selected card (`j`). To swap the cards we use a tuple assignment:

```
self.cards[i], self.cards[j] = self.cards[j],
self.cards[i]
```

Removing and dealing cards

Another method that would be useful for the `Deck` class is `remove`, which takes a card as a parameter, removes it, and returns `True` if the card was in the deck and `False` otherwise:

```
class Deck:
    ...
    def remove(self, card):
        if card in self.cards:
            self.cards.remove(card)
            return True
        else:
            return False
```

The `in` operator returns `True` if the first operand is in the second, which must be a list or a tuple. If the first operand is an object, Python uses the object's `__cmp__` method to determine equality with items in the list. Since the `__cmp__` in the `Card` class checks for deep equality, the `remove` method checks for deep equality.

To deal cards, we want to remove and return the top card. The list method `pop` provides a convenient way to do that:

```
class Deck:
    ...
    def pop(self):
        return self.cards.pop()
```

Actually, `pop` removes the *last* card in the list, so we are in effect dealing from the bottom of the deck.

One more operation that we are likely to want is the boolean function `is_empty`, which returns true if the deck contains no cards:

```
class Deck:
    ...
    def is_empty(self):
        return (len(self.cards) == 0)
```

17. Inheritance

17.1. Inheritance

The language feature most often associated with object-oriented programming is **inheritance**. Inheritance is the ability to define a new class that is a modified version of an existing class.

The primary advantage of this feature is that you can add new methods to a class without modifying the existing class. It is called inheritance because the new class inherits all of the methods of the existing class. Extending this metaphor, the existing class is sometimes called the **parent** class. The new class may be called the **child** class or sometimes subclass.

Inheritance is a powerful feature. Some programs that would be complicated without inheritance can be written concisely and simply with it. Also, inheritance can facilitate code reuse, since you can customize the behavior of parent classes without having to modify them. In some cases, the inheritance structure reflects the natural structure of the problem, which makes the program easier to understand.

On the other hand, inheritance can make programs difficult to read. When a method is invoked, it is sometimes not clear where to find its definition. The relevant code may be scattered among several modules. Also, many of the things that can be done using inheritance can be done as elegantly (or more so) without it. If the natural structure of the problem does not lend itself to inheritance, this style of programming can do more harm than good.

In this chapter we will demonstrate the use of inheritance as part of a program that plays the card game Old Maid. One of our goals is to write code that could be reused to implement other card games.

17.2. A hand of cards

For almost any card game, we need to represent a hand of cards. A hand is similar to a deck, of course. Both are made up of a set of cards, and both require operations like adding and removing cards. Also, we might like the ability to shuffle both decks and hands.

A hand is also different from a deck. Depending on the game being played, we might want to perform some operations on hands that don't make sense for a deck. For example, in poker we might classify a hand (straight, flush, etc.) or compare it with another hand. In bridge, we might want to compute a score for a hand in order to make a bid.

This situation suggests the use of inheritance. If Hand is a subclass of Deck, it will have all the methods of Deck, and new methods can be added.

In the class definition, the name of the parent class appears in parentheses:

```
class Hand(Deck):
    pass
```

This statement indicates that the new Hand class inherits from the existing Deck class.

The Hand constructor initializes the attributes for the hand, which are name and cards. The string name identifies this hand, probably by the name of the player that holds it. The name is an optional parameter with the empty string as a default value. cards is the list of cards in the hand, initialized to the empty list:

```
class Hand(Deck):
    def __init__(self, name=""):
       self.cards = []
       self.name = name
```

For just about any card game, it is necessary to add and remove cards from the deck. Removing cards is already taken care of, since Hand inherits remove from Deck. But we have to write add:

```
class Hand(Deck):
    ...
    def add(self,card):
        self.cards.append(card)
```

Again, the ellipsis indicates that we have omitted other methods. The list append method adds the new card to the end of the list of cards.

17.3. Dealing cards

Now that we have a Hand class, we want to deal cards from the Deck into hands. It is not immediately obvious whether this method should go in the Hand class or in the Deck class, but since it operates on a single deck and (possibly) several hands, it is more natural to put it in Deck.

deal should be fairly general, since different games will have different requirements. We may want to deal out the entire deck at once or add one card to each hand.

deal takes two parameters, a list (or tuple) of hands and the total number of cards to deal. If there are not enough cards in the deck, the method deals out all of the cards and stops:

```
class Deck :
    ...
    def deal(self, hands, num_cards=999):
        num_hands = len(hands)
        for i in range(num_cards):
            if self.is_empty(): break   # break if out of cards
            card = self.pop()           # take the top card
            hand = hands[i % num_hands] # whose turn is next?
            hand.add(card)              # add the card to the hand
```

The second parameter, `num_cards`, is optional; the default is a large number, which effectively means that all of the cards in the deck will get dealt.

The loop variable `i` goes from 0 to `nCards-1`. Each time through the loop, a card is removed from the deck using the list method `pop`, which removes and returns the last item in the list.

The modulus operator (`%`) allows us to deal cards in a round robin (one card at a time to each hand). When `i` is equal to the number of hands in the list, the expression `i % nHands` wraps around to the beginning of the list (index 0).

17.4. Printing a Hand

To print the contents of a hand, we can take advantage of the `printDeck` and `__str__` methods inherited from `Deck`. For example:

```
>>> deck = Deck()
>>> deck.shuffle()
>>> hand = Hand("frank")
>>> deck.deal([hand], 5)
>>> print hand
Hand frank contains
2 of Spades
 3 of Spades
  4 of Spades
   Ace of Hearts
    9 of Clubs
```

It's not a great hand, but it has the makings of a straight flush.

Although it is convenient to inherit the existing methods, there is additional information in a `Hand` object we might want to include when we print one. To do that, we can provide a `__str__` method in the `Hand` class that overrides the one in the `Deck` class:

```
class Hand(Deck)
    ...
    def __str__(self):
        s = "Hand " + self.name
        if self.is_empty():
            s = s + " is empty\n"
        else:
            s = s + " contains\n"
        return s + Deck.__str__(self)
```

Initially, `s` is a string that identifies the hand. If the hand is empty, the program appends the words `is empty` and returns `s`.

Otherwise, the program appends the word `contains` and the string representation of the `Deck`, computed by invoking the `__str__` method in the `Deck` class on `self`.

It may seem odd to send `self`, which refers to the current `Hand`, to a `Deck` method, until you remember that a `Hand` is a kind of `Deck`. `Hand` objects can do everything `Deck` objects can, so it is legal to send a `Hand` to a `Deck` method.

In general, it is always legal to use an instance of a subclass in place of an instance of a parent class.

17.5. The `CardGame` class

The `CardGame` class takes care of some basic chores common to all games, such as creating the deck and shuffling it:

```
class CardGame:
    def __init__(self):
        self.deck = Deck()
        self.deck.shuffle()
```

This is the first case we have seen where the initialization method performs a significant computation, beyond initializing attributes.

To implement specific games, we can inherit from `CardGame` and add features for the new game. As an example, we'll write a simulation of Old Maid.

The object of Old Maid is to get rid of cards in your hand. You do this by matching cards by rank and color. For example, the 4 of Clubs matches the 4 of Spades since both suits are black. The Jack of Hearts matches the Jack of Diamonds since both are red.

To begin the game, the Queen of Clubs is removed from the deck so that the Queen of Spades has no match. The fifty-one remaining cards are dealt to the players in a round robin. After the deal, all players match and discard as many cards as possible.

When no more matches can be made, play begins. In turn, each player picks a card (without looking) from the closest neighbor to the left who still has cards. If the chosen card matches a card in the player's hand, the pair is removed. Otherwise, the card is added to the player's hand. Eventually all possible matches are made, leaving only the Queen of Spades in the loser's hand.

In our computer simulation of the game, the computer plays all hands. Unfortunately, some nuances of the real game are lost. In a real game, the player with the Old Maid goes to some effort to get their neighbor to pick that card, by displaying it a little more prominently, or perhaps failing to display it more prominently, or even failing to fail to display that card more prominently. The computer simply picks a neighbor's card at random.

17.6. `OldMaidHand` class

A hand for playing Old Maid requires some abilities beyond the general abilities of a `Hand`. We will define a new class, `OldMaidHand`, that inherits from `Hand` and provides an additional method called `remove_matches`:

```
class OldMaidHand(Hand):
    def remove_matches(self):
        count = 0
        original_cards = self.cards[:]
        for card in original_cards:
            match = Card(3 - card.suit, card.rank)
            if match in self.cards:
                self.cards.remove(card)
                self.cards.remove(match)
                print "Hand %s: %s matches %s" % (self.name, card, match)
                count = count + 1
        return count
```

We start by making a copy of the list of cards, so that we can traverse the copy while removing cards from the original. Since `self.cards` is modified in the loop, we don't want to use it to control the traversal. Python can get quite confused if it is traversing a list that is changing!

For each card in the hand, we figure out what the matching card is and go looking for it. The match card has the same rank and the other suit of the same color. The expression `3 - card.suit` turns a Club (suit 0) into a Spade (suit 3) and a Diamond (suit 1) into a Heart (suit 2). You should satisfy yourself that the opposite operations also work. If the match card is also in the hand, both cards are removed.

The following example demonstrates how to use `remove_matches`:

```
>>> game = CardGame()
>>> hand = OldMaidHand("frank")
>>> game.deck.deal([hand], 13)
>>> print hand
```

```
Hand frank contains
Ace of Spades
 2 of Diamonds
  7 of Spades
   8 of Clubs
    6 of Hearts
     8 of Spades
      7 of Clubs
       Queen of Clubs
        7 of Diamonds
         5 of Clubs
          Jack of Diamonds
           10 of Diamonds
            10 of Hearts
>>> hand.remove_matches()
Hand frank: 7 of Spades matches 7 of Clubs
Hand frank: 8 of Spades matches 8 of Clubs
Hand frank: 10 of Diamonds matches 10 of Hearts
>>> print hand
Hand frank contains
Ace of Spades
 2 of Diamonds
  6 of Hearts
   Queen of Clubs
    7 of Diamonds
     5 of Clubs
      Jack of Diamonds
```

Notice that there is no `__init__` method for the `OldMaidHand` class. We inherit it from `Hand`.

17.7. `OldMaidGame` class

Now we can turn our attention to the game itself. `OldMaidGame` is a subclass of `CardGame` with a new method called `play` that takes a list of players as a parameter.

Since `__init__` is inherited from `CardGame`, a new `OldMaidGame` object contains a new shuffled deck:

```
class OldMaidGame(CardGame):
    def play(self, names):
        # remove Queen of Clubs
        self.deck.remove(Card(0,12))

        # make a hand for each player
        self.hands = []
        for name in names:
            self.hands.append(OldMaidHand(name))

        # deal the cards
        self.deck.deal(self.hands)
        print "---------- Cards have been dealt"
        self.printHands()

        # remove initial matches
        matches = self.removeAllMatches()
        print "---------- Matches discarded, play begins"
        self.printHands()

        # play until all 50 cards are matched
        turn = 0
        numHands = len(self.hands)
        while matches < 25:
            matches = matches + self.playOneTurn(turn)
            turn = (turn + 1) % numHands

        print "---------- Game is Over"
        self.printHands()
```

The writing of `printHands()` is left as an exercise.

Some of the steps of the game have been separated into methods. `remove_all_matches` traverses the list of hands and invokes `remove_matches` on each:

```
class OldMaidGame(CardGame):
    ...
    def remove_all_matches(self):
        count = 0
        for hand in self.hands:
            count = count + hand.remove_matches()
        return count
```

`count` is an accumulator that adds up the number of matches in each hand and returns the total.

When the total number of matches reaches twenty-five, fifty cards have been removed from the hands, which means that only one card is left and the game is over.

The variable `turn` keeps track of which player's turn it is. It starts at 0 and increases by one each time; when it reaches `numHands`, the modulus operator wraps it back around to 0.

The method `playOneTurn` takes a parameter that indicates whose turn it is. The return value is the number of matches made during this turn:

```
class OldMaidGame(CardGame):
    ...
    def play_one_turn(self, i):
        if self.hands[i].is_empty():
            return 0
        neighbor = self.find_neighbor(i)
        pickedCard = self.hands[neighbor].popCard()
        self.hands[i].add(pickedCard)
        print "Hand", self.hands[i].name, "picked", pickedCard
        count = self.hands[i].remove_matches()
        self.hands[i].shuffle()
        return count
```

If a player's hand is empty, that player is out of the game, so he or she does nothing and returns 0.

Otherwise, a turn consists of finding the first player on the left that has cards, taking one card from the neighbor, and checking for matches. Before returning, the cards in the hand are shuffled so that the next player's choice is random.

The method `find_neighbor` starts with the player to the immediate left and continues around the circle until it finds a player that still has cards:

```
class OldMaidGame(CardGame):
    ...
    def find_neighbor(self, i):
        numHands = len(self.hands)
        for next in range(1,numHands):
            neighbor = (i + next) % numHands
            if not self.hands[neighbor].is_empty():
                return neighbor
```

If `find_neighbor` ever went all the way around the circle without finding cards, it would return `None` and cause an error elsewhere in the program. Fortunately, we can prove that that will never happen (as long as the end of the game is detected correctly).

We have omitted the `print_hands` method. You can write that one yourself.

The following output is from a truncated form of the game where only the top fifteen cards (tens and higher) were dealt to three players. With this small deck, play stops after seven matches instead of twenty-five.

```
>>> import cards
>>> game = cards.OldMaidGame()
>>> game.play(["Allen","Jeff","Chris"])
---------- Cards have been dealt
Hand Allen contains
King of Hearts
 Jack of Clubs
  Queen of Spades
   King of Spades
    10 of Diamonds

Hand Jeff contains
Queen of Hearts
```

```
 Jack of Spades
  Jack of Hearts
   King of Diamonds
    Queen of Diamonds

Hand Chris contains
Jack of Diamonds
 King of Clubs
  10 of Spades
   10 of Hearts
    10 of Clubs

Hand Jeff: Queen of Hearts matches Queen of Diamonds
Hand Chris: 10 of Spades matches 10 of Clubs
---------- Matches discarded, play begins
Hand Allen contains
King of Hearts
 Jack of Clubs
  Queen of Spades
   King of Spades
    10 of Diamonds

Hand Jeff contains
Jack of Spades
 Jack of Hearts
  King of Diamonds

Hand Chris contains
Jack of Diamonds
 King of Clubs
  10 of Hearts

Hand Allen picked King of Diamonds
Hand Allen: King of Hearts matches King of Diamonds
Hand Jeff picked 10 of Hearts
Hand Chris picked Jack of Clubs
Hand Allen picked Jack of Hearts
Hand Jeff picked Jack of Diamonds
Hand Chris picked Queen of Spades
Hand Allen picked Jack of Diamonds
Hand Allen: Jack of Hearts matches Jack of Diamonds
Hand Jeff picked King of Clubs
Hand Chris picked King of Spades
Hand Allen picked 10 of Hearts
Hand Allen: 10 of Diamonds matches 10 of Hearts
Hand Jeff picked Queen of Spades
Hand Chris picked Jack of Spades
Hand Chris: Jack of Clubs matches Jack of Spades
Hand Jeff picked King of Spades
Hand Jeff: King of Clubs matches King of Spades
---------- Game is Over
Hand Allen is empty

Hand Jeff contains
```

```
Queen of Spades

Hand Chris is empty
```

So Jeff loses.

Acknowledgement

This is to bring to the notice to all concerned readers , the document is prepared for learning materials to students and is for purely academic purpose. The above notes have been prepared using the following resources :

1. "*How to think like a computer scientist* ", by Jeffrey Elkner, Allen B. Downey, and Chris Meyers
2. "*Automating the Boring Stuff using Python*", by Al Sweigart.