Module 1: Introduction to everything

Lab 2: Python function and OOPs concepts

Objevtive

- **Function**
  - Introduction to Function
  - How to define function?
  - parameters and arguments
  - return statement
  - lambda
- **Object Oriented Programming**
  - Introduction
  - Class and Object
  - Inheritance
  - Encapsulation
  - Polymorphism

Functions and scope

We have seen that Python has several built-in functions (e.g. `print()` or `max()`). But you can also create a function. A function is a reusable block of code that performs a specific task. Once you have defined a function, you can use it at any place in your Python script. You can even import a function from an external module (as we will see in the next chapter). Therefore, they are beneficial for tasks that you will perform more often. Plus, functions are a convenient way to order your code and make it more readable!

Now let's get started!

Writing a function

A **function** is an isolated chunk of code that has a name, gets zero or more parameters, and returns a value. In general, a function will do something for you based on the input parameters you pass it, and it will typically return a result. You are not limited to using

functions available in the standard library or the ones provided by external parties. You can also write your own functions!

Whenever you are writing a function, you need to think of the following things:

- What is the purpose of the function?
- How should I name the function?
- What input does the function need?
- What output should the function generate?

Why use a function?

There are several good reasons why functions are a vital component of any non-ridiculous programmer:

- encapsulation: wrapping a piece of useful code into a function so that it can be used without knowledge of the specifics
- generalization: making a piece of code useful in varied circumstances through parameters
- manageability: Dividing a complex program up into easy-to-manage chunks
- maintainability: using meaningful names to make the program better readable and understandable
- reusability: a good function may be useful in multiple programs
- recursion!

How to define a function

Let's say we want to sing a birthday song to Julee Then we print the following lines:

```
print("Happy Birthday to you!")
print("Happy Birthday to you!")
print("Happy Birthday, dear kartik.")
print("Happy Birthday to you!")
```

```
Happy Birthday to you!
Happy Birthday to you!
Happy Birthday, dear kartik.
Happy Birthday to you!
```

This could be the purpose of a function: to print the lines of a birthday song for Emily. Now, we define a function to do this. Here is how you define a function:

- write `def`;
- the name you would like to call your function;
- a set of parentheses containing the parameter(s) of your function;
- a colon;

- a docstring describing what your function does;
- the function definition;
- ending with a return statement

Statements must be indented so that Python knows what belongs in the function and what not. Functions are only executed when you call them. It is good practice to define your functions at the top of your program or in another Python module.

We give the function a clear name, `happy_birthday_to_kartik`, and we define the function as shown below. Note that we specify what it does in the docstring at the beginning of the function:

```
def happy_birthday_to_kartik(): # Function definition
    """
    Print a birthday song to Julee.
    """
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print("Happy Birthday, dear kartik.")
    print("Happy Birthday to you!")
```

If we execute the code above, we don't get any output. That's because we only told Python: "Here's a function to do this, please remember it." If we actually want Python to execute everything inside this function, we have to call it:

How to call a function

It is important to distinguish between a function **definition** and a function **call**. We illustrate this in 1.3.1. You can also call functions from within other functions. This will become useful when you split up your code into small chunks that can be combined to solve a larger problem. This is illustrated in 1.3.2.

A simple function call

A function is **defined** once. After the definition, Python has remembered what this function does in its memory. A function is **executed/called** as many times as we like. When calling a function, you should always use parenthesis.

```
# function definition:

def happy_birthday_to_kartik(): # Function definition
    """
    Print a birthday song to kartik.
    """
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print("Happy Birthday, dear kartik.")
    print("Happy Birthday to you!")
```

```
# function call:

print('Function call 1')

happy_birthday_to_kartik()

print()
# We can call the function as many times as we want (but we define it
only once)
print('Function call 2')

happy_birthday_to_kartik()

print()

print('Function call 3')

happy_birthday_to_kartik()

print()
# This will not call the function

print('This is not a function call')
happy_birthday_to_kartik
```

```
Function call 1
Happy Birthday to you!
Happy Birthday to you!
Happy Birthday, dear kartik.
Happy Birthday to you!

Function call 2
Happy Birthday to you!
Happy Birthday to you!
Happy Birthday, dear kartik.
Happy Birthday to you!

Function call 3
Happy Birthday to you!
Happy Birthday to you!
Happy Birthday, dear kartik.
Happy Birthday to you!

This is not a function call

<function __main__.happy_birthday_to_kartik()>
```

Calling a function from within another function

We can also define functions that call other functions, which is very helpful if we want to split our task into smaller, more manageable subtasks:

```
def new_line():
    """Print a new line."""
    print()

def two_new_lines():
    """Print two new lines."""
    new_line()
    new_line()

print("Printing a single line...")
new_line()
print("Printing two lines...")
two_new_lines()
print("Printed two lines")
```

```
Printing a single line...

Printing two lines...


Printed two lines
```

```
help(happy_birthday_to_kartik)
```

```
Help on function happy_birthday_to_kartik in module __main__:

happy_birthday_to_kartik()
    Print a birthday song to kartik.
```

```
type(happy_birthday_to_kartik)
```

```
function
```

Working with function input

Parameters and arguments

We use parameters and arguments to make a function execute a task depending on the input we provide. For instance, we can change the function above to input the name of a person and print a birthday song using this name. This results in a more generic function.

To understand how we use **parameters** and **arguments**, keep in mind the distinction between function *definition* and function *call*.

Parameter: The variable `name` in the **function definition** below is a **parameter**. Variables used in **function definitions** are called **parameters**.

Argument: The variable `my_name` in the function call below is a value for the parameter `name` at the time when the function is called. We refer to such variables as **arguments**. We use arguments so we can direct the function to do different kinds of work when we call it at different times.

```
# function definition with using the parameter `name'
def happy_birthday(name):
    """
    Print a birthday song with the "name" of the person inserted.
    """
    print("Happy Birthday to you!")
    print("Happy Birthday to you!")
    print(f"Happy Birthday, dear {name}.")
    print("Happy Birthday to you!")

# function call using specifying the value of the argument
happy_birthday("kartik")
```

```
Happy Birthday to you!
Happy Birthday to you!
Happy Birthday, dear kartik.
Happy Birthday to you!
```

```
my_name="kartik"
happy_birthday(my_name)
```

```
Happy Birthday to you!
Happy Birthday to you!
Happy Birthday, dear kartik.
Happy Birthday to you!
```

```
happy_birthday()
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call
last)
<ipython-input-10-e6218e6c7f60> in <module>
----> 1 happy_birthday()

TypeError: happy_birthday() missing 1 required positional argument:
'name'
```

Functions can have multiple parameters. We can for example multiply two numbers in a function (using the two parameters x and y) and then call the function by giving it two arguments:

```
def multiply(x, y):
    """Multiply two numeric values."""
```

```
    result = x * y
    print(result)

multiply(2020,5278238)
multiply(2,3)
```

```
10662040760
6
```

Positional vs keyword parameters and arguments

The function definition tells Python which parameters are positional and which are keyword. As you might remember, positional means that you have to give an argument for that parameter; keyword means that you can give an argument value, but this is not necessary because there is a default value.

So, to summarize these two notes, we distinguish between:

1) **positional parameters**: (we indicate these when defining a function, and they are compulsory when calling the function)

2) **keyword parameters**: (we indicate these when defining a function, but they have a default value - and are optional when calling the function)

```
def multiply(x, y, third_number=1): # x and y are positional
parameters, third_number is a keyword parameter
    """Multiply two or three numbers and print the result."""
    result=x*y*third_number
    print(result)

multiply(2,3) # We only specify values for the positional parameters
multiply(2,3,third_number=4) # We specify values for both the
positional parameters, and the keyword parameter
```

```
6
24
```

Output: the `return` statement

Functions can have a **return** statement. The `return` statement returns a value back to the caller and **always** ends the execution of the function. This also allows us to use the result of a function outside of that function by assigning it to a variable:

```
def multiply(x, y):
    """Multiply two numbers and return the result."""
    multiplied = x * y
    return multiplied

#here we assign the returned value to variable z
```

```
result = multiply(2, 5)

print(result)
```

```
10
```

```
multiply(30,20)
```

```
600
```

Returning multiple values

```
def calculate(x,y):
    """Calculate product and sum of two numbers."""
    product = x * y
    summed = x + y

    #we return a tuple of values
    return product, summed

# the function returned a tuple and we unpack it to var1 and var2
var1, var2 = calculate(10,5)

print("product:",var1,"sum:",var2)
```

```
product: 50 sum: 15
```

Lambda - anonymous function

In Python, an **anonymous function** is a function that is defined without a name.

While normal functions are defined using the **def** keyword in Python, anonymous functions are defined using the lambda keyword.

In opposite to a normal function, a Python **lambda** function is a single expression. But, in a lambda body, we can expand with expressions over multiple lines using parentheses () or a multiline string """ """.

For example: lambda n:n+n

The reason behind the using anonymous function is for instant use, that is, one-time usage and the code is very concise so that there is more readability in the code.

Hence, anonymous functions are also called lambda functions.

Lambda forms can take any number of arguments but return just one value in the form of an expression. They cannot contain commands or multiple expressions.

An anonymous function cannot be a direct call to print because lambda requires an expression.

Syntax:

lambda **argument_list**: **expression**

```
#  Program to show the use of lambda functions

double = lambda x: x * 2

print(double(6))
```

```
12
```

Explanation:

In the above program, lambda **x: x * 2** is the lambda function. Here x is the argument and **x * 2** is the expression that gets evaluated and returned.

This function has no name. It returns a function object which is assigned to the identifier double. We can now call it as a normal function.

Use of lambda Function in python

We use lambda function when we require a nameless function for a short period of time.

In Python, we generally use it as an argument to a higher-order function (a function that takes in other functions as arguments). lambda function are used along with built-in functions like **filter()**, **map()**, **reduce()** etc.

```
# Program to filter out only the even items from a list

my_list = [1, 5, 4, 6, 8, 11, 3, 12]  # total 8 elements

new_list = list(filter(lambda x: (x%2 == 0), my_list)) # returns the
output in form of a list

print("Even numbers are: ", new_list)
```

```
Even numbers are:  [4, 6, 8, 12]
```

lambda function with map()

The **map()** function in Python takes in a function and a list.

The function is called with all the items in the list and a new list is returned which contains items returned by that function for each item.

Here is an example use of **map()** function to double all the items in a list.

```
# Program to double each item in a list using map()

my_list = [1, 5, 4, 6, 8, 11, 3, 12]  # total 8 elements
new_list = list(map(lambda x: x * 2, my_list)) # returns the output in
form of a list
print("Double values are: ", new_list)
```

```
Double values are:  [2, 10, 8, 12, 16, 22, 6, 24]
```

lambda function with reduce()

The **reduce()** function is used to minimize sequence elements into a single value by applying the specified condition.

The **reduce()** function is present in the functools module; hence, we need to import it using the import statement before using it.

```
from functools import reduce
list1 = [20, 13, 4, 8, 9]
add = reduce(lambda x, y: x+y, list1)
print("Addition of all list elements is : ", add)
```

```
Addition of all list elements is :  54
```

Object Oriented Programming

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).

Object-oriented programming (OOP) is a programming paradigm based on the concept of **"objects"**. The object contains both data and code: Data in the form of properties (often known as attributes), and code, in the form of methods (actions object can perform).

An object has two characteristics:

- attributes
- behavior

For example:

A parrot is can be an object,as it has the following properties:

- name, age, color as attributes

- singing, dancing as behavior

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

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

Class

In Python, everything is an object. A class is a blueprint for the object. To create an object we require a model or plan or blueprint which is nothing but class.

We create class to create an object. A class is like an object constructor, or a "blueprint" for creating objects. We instantiate a class to create an object. The class defines attributes and the behavior of the object, while the object, on the other hand, represents the class.

Class represents the properties (attribute) and action (behavior) of the object. Properties represent variables, and actions are represented by the methods. Hence class contains both variables and methods.

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

Syntax:

```
class classname:
    '''documentation string'''
    class_suite
```

- **Documentation string:** represent a description of the class. It is optional.
- **class_suite:** class suite contains component statements, variables, methods, functions, attributes.

The example for class of parrot can be :

```
class Parrot:
    pass
```

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

```
class Person:
    pass
print(Person)
```

```
# Creating a class

class Person:
    pass
print(Person)
```

```
<class '__main__.Person'>
```

Object

The physical existence of a class is nothing but an object. In other words, the object is an entity that has a state and behavior.

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

Syntax:

```
reference_variable = classname()
```

The example for object of parrot class can be:

```
obj = Parrot()
```

Here, **`obj`** is an **`object`** of class Parrot.

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

```
p = Person()
print(p)
```

```
# Example 1: We can create an object by calling the class

p = Person()
print(p)
```

```
<__main__.Person object at 0x00000218D8CE95B0>
```

```
# Example 2: Creating Class and Object in Python

class Student:
    """This is student class with data"""
    def learn(self):    # A sample method
        print("Welcome to Dr. Milaan Parmar's class on Python
Programming")

stud = Student()        # creating object
stud.learn()            # Calling method

# Output: Welcome to Dr. Milaan Parmar's class on Python Programming
```

```
Welcome to Dr. Milaan Parmar's class on Python Programming
```

Class Constructor

In the examples above, we have created an object from the **Person** class. However, a class without a constructor is not really useful in real applications. Let us use constructor function to make our class more useful. Like the constructor function in Java or JavaScript, Python has also a built-in **__init__()** constructor function. The **__init__()** constructor function has **self** parameter which is a reference to the current instance of the class.

```
class Person:
      def __init__ (self, name):
        # self allows to attach parameter to the class
          self.name =name

p = Person('Milaan')
print(p.name)
print(p)
```

```
Milaan
<__main__.Person object at 0x00000218D8CE9220>
```

Let us add more parameters to the constructor function.

```
# Example 1: add more parameters to the constructor function.

class Person:
      def __init__(self, firstname, lastname, age, country, city):
            self.firstname = firstname
            self.lastname = lastname
            self.age = age
            self.country = country
            self.city = city

p = Person('Milaan', 'Parmar', 96, 'England', 'London')
print(p.firstname)
print(p.lastname)
print(p.age)
print(p.country)
print(p.city)
```

```
Milaan
Parmar
96
England
London
```

Instance Variables and Methods

If the value of a variable varies from object to object, then such variables are called instance variables. For every object, a separate copy of the instance variable will be created.

When we create classes in Python, instance methods are used regularly. we need to create an object to execute the block of code or action defined in the instance method.

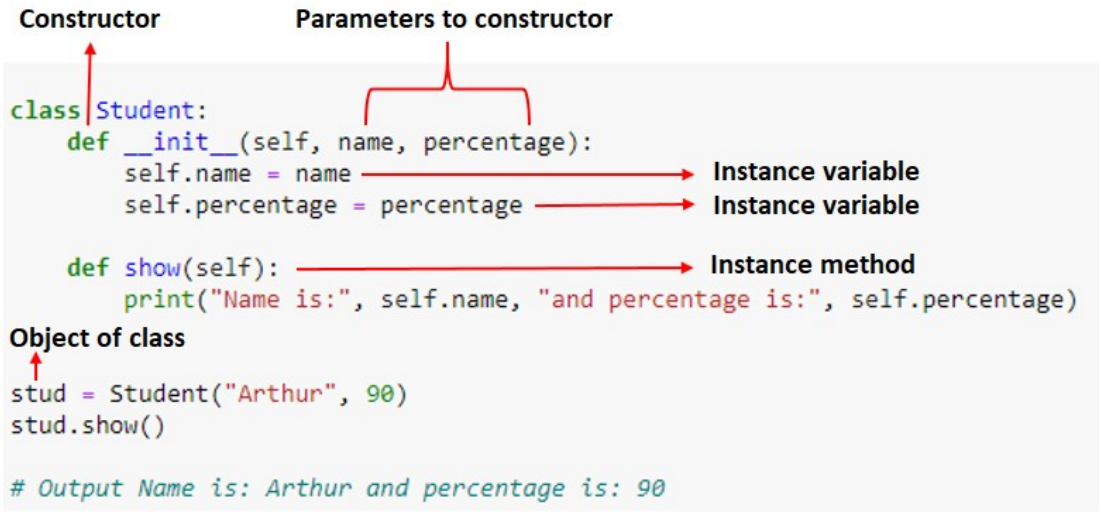
We can access the instance variable and methods using the object. Use dot (**.**) operator to access instance variables and methods.

In Python, working with an instance variable and method, we use the **`self`** keyword. When we use the **`self`** keyword as a parameter to a method or with a variable name is called the instance itself.

Note: Instance variables are used within the instance method

```
# Example 2: Creating Class and Object in Python

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("Arthur", 90)
stud.show()

# Output Name is: Arthur and percentage is: 90
```

```
Name is: Arthur and percentage is: 90
```



```
# Example 3: Creating Class and Object in Python

class Parrot:
    species = "bird"                    # class attribute
    def __init__(self, name, age):      # instance attribute
        self.name = name
        self.age = age

# instantiate the Parrot class
blu = Parrot("Blu", 10)
woo = Parrot("Woo", 15)

# access the class attributes
print("Blu is a {}".format(blu.__class__.species))
print("Woo is also a {}".format(woo.__class__.species))

# access the instance attributes
print("{} is {} years old".format( blu.name, blu.age))
print("{} is {} years old".format( woo.name, woo.age))
```

```
Blu is a bird
Woo is also a bird
Blu is 10 years old
Woo is 15 years old
```

Explanation:

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

These attributes are defined inside the **`__init__`** method of the class. It is the initializer method that is first run as soon as the object is created.

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

We can access the class attribute using **`__class__.species`**. Class attributes are the same for all instances of a class. Similarly, we access the instance attributes using **`blu.name`** and **`blu.age`**. However, instance attributes are different for every instance of a class.

Object Method

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

Objects can have methods. The methods are functions which belong to the object.

```
# Example 1:

class Person:
      def __init__(self, firstname, lastname, age, country, city):
            self.firstname = firstname
            self.lastname = lastname
            self.age = age
            self.country = country
            self.city = city
      def person_info(self):
        return f'{self.firstname} {self.lastname} is {self.age} years
old. He lives in {self.city}, {self.country}'

p = Person('Milaan', 'Parmar', 96, 'England', 'London')
print(p.person_info())
```

```
Milaan Parmar is 96 years old. He lives in London, England
```

```
# Example 2: Creating Object Methods in Python

class Parrot:

    # instance attributes
    def __init__(self, name, age):
        self.name = name
        self.age = age

    # instance method
    def sing(self, song):
        return "{} sings {}".format(self.name, song)

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

# instantiate the object
blu = Parrot("Blu", 10)

# call our instance methods
print(blu.sing("'Happy'"))
print(blu.dance())
```

```
Blu sings 'Happy'
Blu is now dancing
```

Explanation:

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

Object Default Methods

Sometimes, you may want to have a default values for your object methods. If we give default values for the parameters in the constructor, we can avoid errors when we call or instantiate our class without parameters. Let's see how it looks:

```
class Person:
      def __init__(self, firstname='Milaan', lastname='Parmar',
age=96, country='England', city='London'):
            self.firstname = firstname
            self.lastname = lastname
            self.age = age
            self.country = country
            self.city = city

      def person_info(self):
        return f'{self.firstname} {self.lastname} is {self.age} years
old. He lives in {self.city}, {self.country}.'

p1 = Person()
print(p1.person_info())
p2 = Person('Ben', 'Doe', 30, 'Finland', 'Tampere')
print(p2.person_info())
```

```
Milaan Parmar is 96 years old. He lives in London, England.
Ben Doe is 30 years old. He lives in Tampere, Finland.
```

Method to Modify Class Default Values

In the example below, the **Person** class, all the constructor parameters have default values. In addition to that, we have skills parameter, which we can access using a method. Let us create **add_skill** method to add skills to the skills list.

```
class Person:
      def __init__(self, firstname='Milaan', lastname='Parmar',
age=96, country='England', city='London'):
            self.firstname = firstname
            self.lastname = lastname
            self.age = age
            self.country = country
            self.city = city
            self.skills = []

      def person_info(self):
        return f'{self.firstname} {self.lastname} is {self.age} years
old. He lives in {self.city}, {self.country}.'
```

```
    def add_skill(self, skill):
        self.skills.append(skill)

p1 = Person()
print(p1.person_info())
p1.add_skill('Python')
p1.add_skill('MATLAB')
p1.add_skill('R')
p2 = Person('Ben', 'Doe', 30, 'Finland', 'Tampere')
print(p2.person_info())
print(p1.skills)
print(p2.skills)
```

```
Milaan Parmar is 96 years old. He lives in London, England.
Ben Doe is 30 years old. He lives in Tampere, Finland.
['Python', 'MATLAB', 'R']
[]
```

Inheritance

In Python, inheritance is the process of inheriting the properties of the base class (or parent class) into a derived class (or child class).

In an Object-oriented programming language, inheritance is an important aspect. Using inheritance we can reuse parent class code. Inheritance allows us to define a class that inherits all the methods and properties from parent class. The parent class or super or base class is the class which gives all the methods and properties. Child class is the class that inherits from another or parent class.

In inheritance, the child class acquires and access all the data members, properties, and functions from the parent class. Also, a child class can also provide its specific implementation to the functions of the parent class.

Use of inheritance

The main purpose of inheritance is the reusability of code because we can use the existing class to create a new class instead of creating it from scratch.

Syntax:

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

```
# Example 1: Use of Inheritance in Python

class ClassOne:              # Base class
```

```
    def func1(self):
        print('This is Parent class')

class ClassTwo(ClassOne):    # Derived class
    def func2(self):
        print('This is Child class')

obj = ClassTwo()
obj.func1()
obj.func2()
```

```
This is Parent class
This is Child class
```

Let us create a student class by inheriting from **`Person`** class.

```
# Example 2: Use of Inheritance in Python

class Student(Person):
    pass

s1 = Student('Arthur', 'Curry', 33, 'England', 'London')
s2 = Student('Emily', 'Carter', 28, 'England', 'Manchester')
print(s1.person_info())
s1.add_skill('HTML')
s1.add_skill('CSS')
s1.add_skill('JavaScript')
print(s1.skills)

print(s2.person_info())
s2.add_skill('Organizing')
s2.add_skill('Marketing')
s2.add_skill('Digital Marketing')
print(s2.skills)
```

```
Arthur Curry is 33 years old. He lives in London, England.
['HTML', 'CSS', 'JavaScript']
Emily Carter is 28 years old. He lives in Manchester, England.
['Organizing', 'Marketing', 'Digital Marketing']
```

Exaplanation:

We did not call the **`__init__()`** constructor in the child class. If we didn't call it then we can still access all the properties from the parent. But if we do call the constructor we can access the parent properties by calling **`super()`**.
We can add a new method to the child or we can override the parent class methods by creating the same method name in the child class. When we add the **`__init__()`** function, the child class will no longer inherit the parent's **`__init__()`** function.

Overriding parent method

```
# Example 2: Overriding parent method from above example

class Student(Person):
    def __init__ (self, firstname='Milaan', lastname='Parmar',age=96,
country='England', city='London', gender='male'):
        self.gender = gender
        super().__init__(firstname, lastname,age, country, city)

    def person_info(self):
        gender = 'He' if self.gender =='male' else 'She'
        return f'{self.firstname} {self.lastname} is {self.age} years
old. {gender} lives in {self.city}, {self.country}.'

s1 = Student('Arthur', 'Curry', 33, 'England', 'London','male')
s2 = Student('Emily', 'Carter', 28, 'England', 'Manchester','female')
print(s1.person_info())
s1.add_skill('HTML')
s1.add_skill('CSS')
s1.add_skill('JavaScript')
print(s1.skills)

print(s2.person_info())
s2.add_skill('Organizing')
s2.add_skill('Marketing')
s2.add_skill('Digital Marketing')
print(s2.skills)
```

```
Arthur Curry is 33 years old. He lives in London, England.
['HTML', 'CSS', 'JavaScript']
Emily Carter is 28 years old. She lives in Manchester, England.
['Organizing', 'Marketing', 'Digital Marketing']
```

Exaplanation:

We can use **`super()`** built-in function or the parent name Person to automatically inherit the methods and properties from its parent. In the example above we override the parent method. The child method has a different feature, it can identify, if the gender is male or female and assign the proper pronoun(He/She).

```
# Example 3: Use of Inheritance in Python

# parent class
class Bird:
    def __init__(self):
        print("Bird is ready")

    def whoisThis(self):
```

```
        print("Bird")

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

# child class
class Penguin(Bird):
    def __init__(self):
        # call super() function
        super().__init__()
        print("Penguin is ready")

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

    def run(self):
        print("Run faster")

peggy = Penguin()
peggy.whoisThis()
peggy.swim()
peggy.run()
```

```
Bird is ready
Penguin is ready
Penguin
Swim faster
Run faster
```

Exaplanation:

In the above program, we created two classes i.e. **`Bird`** (parent class) and **`Penguin`** (child class). The child class inherits the functions of parent class. We can see this from the **`swim()`** method.

Again, the child class modified the behavior of the parent class. We can see this from the **`whoisThis()`** method. Furthermore, we extend the functions of the parent class, by creating a new **`run()`** method.

Additionally, we use the **`super()`** function inside the **`__init__()`** method. This allows us to run the **`__init__()`** method of the parent class inside the child class.

Encapsulation

In Python, encapsulation is a method of wrapping data and functions into a single entity. For example, A class encapsulates all the data (methods and variables). Encapsulation

means the internal representation of an object is generally hidden from outside of the object's definition.

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

Need of Encapsulation

Encapsulation acts as a protective layer. We can restrict access to methods and variables from outside, and It can prevent the data from being modified by accidental or unauthorized modification. Encapsulation provides security by hiding the data from the outside world.

In Python, we do not have access modifiers directly, such as public, private, and protected. But we can achieve encapsulation by using single prefix underscore and double underscore to control access of variable and method within the Python program.

```
# Example 1: Data Encapsulation in Python

class Computer:

    def __init__(self):
        self.__maxprice = 900

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

    def setMaxPrice(self, price):
        self.__maxprice = price

c = Computer()
c.sell()

# change the price
c.__maxprice = 1000
c.sell()

# using setter function
c.setMaxPrice(1000)
c.sell()
```

```
Selling Price: 900
Selling Price: 900
Selling Price: 1000
```

Explanation:

In the above program, we defined a Computer class.

We used **init**() method to store the maximum selling price of Computer. We tried to modify the price. However, we can't change it because Python treats the __maxprice as private attributes.

As shown, to change the value, we have to use a setter function i.e setMaxPrice() which takes price as a parameter.

Polymorphism

Polymorphism is based on the greek words **Poly** (many) and **morphism** (forms). We will create a structure that can take or use many forms of objects.

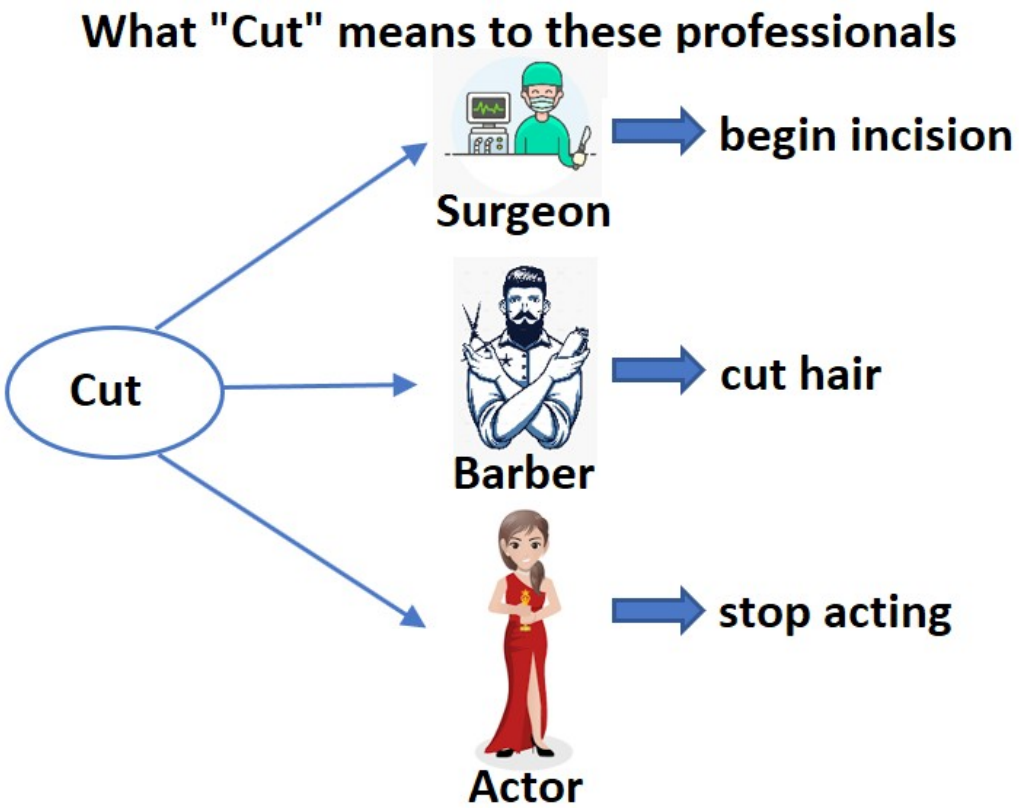
Polymorphism is an ability (in OOP) to use a common interface for multiple forms (data types).

Example 1: The student can act as a student in college, act as a player on the ground, and as a daughter/brother in the home.

Example 2: In the programming language, the **+** operator, acts as a concatenation and arithmetic addition.

Example 3: If we need to color a shape, there are multiple shape options (rectangle, square, circle). However we could use the same method to color any shape.

In Python, polymorphism allows us to define the child class methods with the same name as defined in the parent class.




```
# Example 1: Using Polymorphism in Python

class Parrot:
    def fly(self):
        print("Parrot can fly")

    def swim(self):
        print("Parrot can't swim")

class Penguin:
    def fly(self):
        print("Penguin can't fly")

    def swim(self):
        print("Penguin can swim")

# common interface
def flying_test(bird):
    bird.fly()

#instantiate objects
blu = Parrot()
peggy = Penguin()

# passing the object
```

```
flying_test(blu)
flying_test(peggy)
```

```
Parrot can fly
Penguin can't fly
```

Exaplanation:

In the above program, we defined two classes **`Parrot`** and **`Penguin`**. Each of them have a common **`fly()`** method. However, their functions are different.

To use polymorphism, we created a common interface i.e **`flying_test()`** function that takes any object and calls the object's **`fly()`** method. Thus, when we passed the **`blu`** and **`peggy`** objects in the **`flying_test()`** function, it ran effectively.

```
# Example 2: Using Polymorphism in Python

class Circle:
    pi = 3.14

    def __init__(self, redius):
        self.radius = redius

    def calculate_area(self):
        print("Area of circle:", self.pi * self.radius * self.radius)

class Rectangle:
    def __init__(self, length, width):
        self.length = length
        self.width = width

    def calculate_area(self):
        print("Area of Rectangle:", self.length * self.width)

cir = Circle(9)
rect = Rectangle(9, 6)
cir.calculate_area()   # Output Area of circle: 254.34

rect.calculate_area()  # Output Area od Rectangle: 54
```

```
Area of circle: 254.34
Area of Rectangle: 54
```

Exaplanation:

In the above example, we created two classes called **`Circle`** and **`Rectangle`**. In both classes, we created the same method with the name **`calculate_area`**. This method acts differently in both classes. In the case of the **`Circle`** class, it calculates the area of the circle, whereas, in the case of a **`Rectangle`** class, it calculates the area of a rectangle.

Thank You !!!