

# Introduction to Python

## **Object-Oriented Programming**

# Topics

- 1) Classes
- 2) Class vs Object
- 3) `__init__`(dunder init)
- 4) Functions vs Methods
- 5) `self`
- 6) Importing modules

# An Example

So far, our data has been one piece of information: an int, a float, a string.

Suppose we are writing a program that manages a database of students. We need a data type that contains information about a student.

A student has more than just one piece of information: name, age, id, address, list of courses, grades etc... These collectively are the **data** of a student.

A student might have some **functionality**: ability to print personal information, change their address, update school information, add a course, update a grade.

We like a data type that can bundle **data** and **functionality** into one variable.

A **class** bundles together *data* (instance variables or attributes) and *functionality* (functions).

# An Example of a class

We'll discuss "self" later in the slides.

```
class Student:
    def __init__(self, name, id):
        self.name = name
        self.id = id
    def print_info(self):
        print(self.name, self.id)
```

Class definition.

This class Student can then be used to create multiple Student objects.

```
s1 = Student("Mike Smith", 34323)
s2 = Student("Sarah Jones", 67432)
print(s1.name)      # Mike Smith
print(s2.name)      # Sarah Jones
print(s1.id)        # 34323
s1.print_info()     # Mike Smith 34323
s2.print_info()     # Sarah Jones 67432
```

Each Student object has data(name, id) and functionality(print\_info).

To access data/functionality, the dot notation is used.

# Class vs Objects

A **class** bundles together *data* (instance variables or attributes) and *functionality* (methods).

Thus, in the previous example, Student is a **class** and s1 and s2 are two of its **objects**.

# OOP/OOD

**Object-Oriented Programming**(OOP) is a programming paradigm based on the concepts of **objects** which bundles together **data**(in the form of instance variables) and **functionality or behavior**(in the form of methods).

Many popular languages are object-oriented(C++, Java, Javascript, Python).

In OOP, programs are made up of many objects and a program run is the interaction of these objects.

# Custom Classes

A **class** bundles together **data** (instance variables or attributes) and **functionality** (methods).

A list has data(the elements of the list). It also has methods that manipulate those data(append, insert, pop, remove, etc...).

The classes int, bool, str, list, tuple, etc... are built-in classes.

Python provides the ability for programmers to design their own classes(**custom classes**).

# Class

We like to be able to build our own classes to represent objects relevant to our game or application.

A game might have a Character class, from which we may create several Character **instances** or **objects**.

This **reusability** feature is important especially when we need to create many objects(for example enemies) with similar data and behaviors.



# Examples

Suppose you are writing an arcade game. What are some useful classes and their corresponding objects?

Example:

The **Character** Class represents characters in the game.

**Variables/Attributes/Data:** name, position, speed.

**Behavior/Methods:** shoot(), runLeft(), runRight(), jump().

Objects: From the same blueprint, the Character class, we can create multiple Character objects.

# Examples

Your game might have more than one classes. Each class can have many objects of that class or type.

**Classes:** Character, Boss, Tile, Bullet.

Objects:

- 1) You may have one player object from the Character class.
- 2) Several Boss objects, one for each level.
- 3) A set of Tile objects for the the platforms on which the game objects walk.
- 4) Many Bullet objects are created as Character or Boss objects shoot.

# Class Declaration

A class is declared with the keyword `class` followed by the class name.

```
class ClassName:
```

To create and initialize our instance variables, we need to define a **special method** called `__init__` (double underscore init or "dunder init"). This method is sometimes called the **constructor**.

# The Character Class

An example of a class.

```
class Character:
    def __init__(self, i_name, i_x, i_speed):
        self.name = i_name
        self.x = i_x
        self.speed = i_speed
```

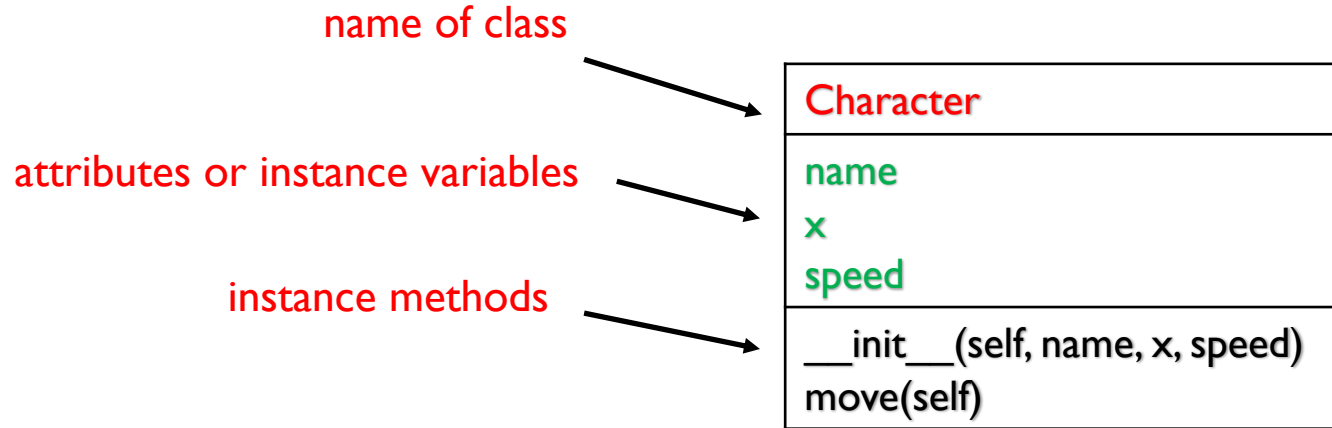
Constructor: **init** is a **special method** that creates and initializes the instance variables(or attributes) (pronounced "dunder init" (double underscore init))

instance variables or instance attributes (use **self** with dot notation)

The **self** parameter is automatically set to reference the newly created object. It can use another name but "**self**" is the convention.

# A Class Diagram

Here's a class diagram that can help you visualize a class.



# Class

2) The self parameter is now pointing to the newly created Character object or instance

```
class Character:
```

```
    def __init__(self, i_name, i_x, i_speed):
```

```
        self.name = i_name
```

```
        self.x = i_x
```

```
        self.speed = i_speed
```

```
p = Character("John", 10, 4)
```

1) An object is first created in memory. Then `__init__` is called and the address of this object is sent to `self`.

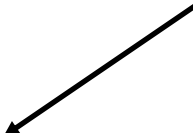
3) The self reference is then used to create and initialize the other attributes or variables of the object.

In this case: name, x and speed are the three attributes of the Character object.

# Class

```
class Character:
    def __init__(self, i_name, i_x, i_speed):
        self.name = i_name
        self.x = i_x
        self.speed = i_speed
```

4) After all instance variables are initialized, the address or reference of the object is returned(to p).



```
p = Character("John", 10, 4)
print(p.x, p.speed) # accessing attributes: 10 4
p.speed = 15 # modifying an instance attribute
print(p.speed) # 15
```

# game.py

```
class Character:
    def __init__(self, i_name, i_x, i_speed):
        self.name = i_name
        self.x = i_x
        self.speed = i_speed
```

- 1) Character is a **class** or **type**.
- 2) p is an **instance** of the Character class.
- 3) p is an **object** of the Character class.
- 4) name, x and speed are **attributes** of the object.

```
p = Character("John", 10, 4)
print(p.x, p.speed) # accessing attributes, 10 4
p.speed = 15 # modifying an instance attribute
print(p.speed) # 15
```



# Function vs Methods

A function defined inside of a class is called a **method(instance method)**.

We saw that `__init__` is one example of a method.

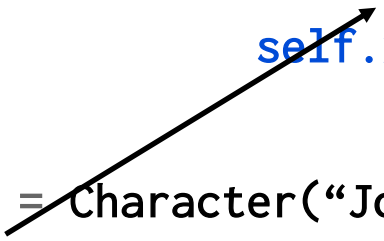
The first parameter of an instance method refers to the instance or object being manipulated. By convention, we use "self" for this first parameter.

Note: In addition to instance methods, Python supports **class methods** and **static methods**. We won't discuss these in this class.

# game.py

```
class Character:
    def __init__(self, i_name, i_x, i_speed):
        ...
    def move(self):
        self.x += self.speed

p = Character("John", 10, 4)
p.move()
print(p.x) # 14
e = Character("Sarah", 100, -5)
e.move()
print(e.x) # 95
main()
```



`move()` is an **instance method**. The first parameter of a method (`self`) refers to the instance being manipulated.

In this case, `p` is being moved.

We have seen this notation before. For example:

```
a = [1, 2, 3]
a.pop()
```

# game.py

```
class Character:
    def __init__(self, i_name, i_x, i_speed):
        ...
    def move(self):
        self.x += self.speed
```

```
p = Character("John", 10, 4)
p.move()
print(p.x) # 14
e = Character("Sarah", 100, -5)
e.move()
print(e.x) # 95
```

In this case, e is being moved.

# game2.py

```
class Character:
    def __init__(self, i_name, i_x, i_speed):
        self.name = i_name
        self.x = i_x
        self.speed = i_speed
    def move(self):
        self.x += self.speed
```

```
p1 = Character("Jack", 10, 4)
p2 = Character("Jill", 20, -3)
p1.move() # p1.x = 14
p2.move() # p2.x = 17
```

The utility of writing a class is that we can create many objects or instances of that class. This code for this example creates 2 Character objects.

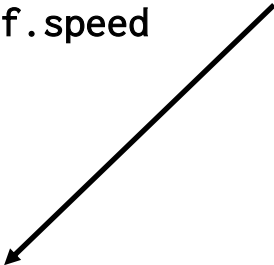
# game3.py

```
import random

class Character:
    def __init__(self, i_name, i_x, i_speed):
        self.name = i_name
        self.x = i_x
        self.speed = i_speed
    def move(self):
        self.x += self.speed
```

```
enemies = []
for i in range(10):
    x = random.randrange(0, 800)
    enemies.append(Character("Goomba", x, 5))
```

randrange(a, b) generates a random integer from a(included) to b(not included).



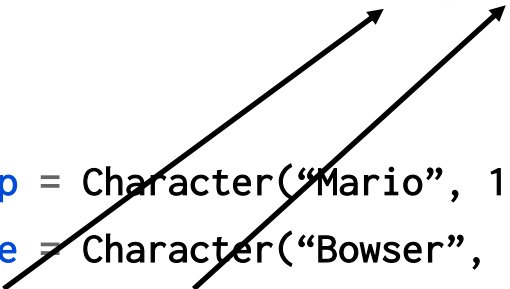
We can even create any number of randomized objects.



# game3.py

```
class Character:
    def __init__(self, i_name, i_x, i_speed): ...
    def move(self): ...
    def shoot(self, target): ...
```

```
p = Character("Mario", 10, 4)
e = Character("Bowser", 20, -3)
e.shoot(p) # p1.x = 14
```



The object `e` is shooting the object `p` in this example.

Thus, `e`'s address is sent to `self` and `p`'s address is sent to `target`.

# Python Program Template

`main.py`

`# declare and initialize global variables with file scope`

`...`

`# function definitions`

`def func1(...):`

`...`

`def func2(...):`

`...`

`# class definitions`

`class MyClass1:`

`...`

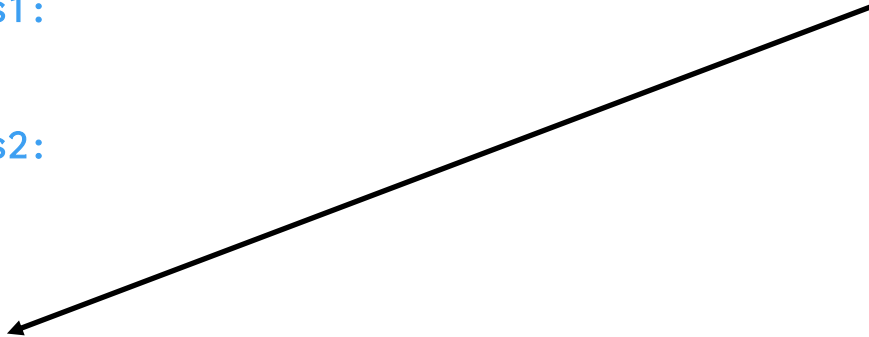
`class MyClass2:`

`...`

`func1()`

`a = MyClass1()`

If our program has a small number of functions and classes, we can define all of them in the same file and use them right below their definitions.



# A Program with Multiple Modules

A more complex program may require many functions, classes. We may wish to organize them into different **modules**.

A **module** is a .py file that contains code, including variable, function and class definitions.

**Importing** a module will execute all of its statements. The objects defined in the imported module is now available in the current module. Let's see how this is done.



# A Program with Multiple Modules

The statement **import** can be used to import the entire module. All of the code from `helper.py` is executed. The dot(".") notation is needed to access imported code.

## main.py

```
import helper

print(helper.a)
helper.lst.append("hello")
print(helper.lst)
print(helper.add(3, 5))
```

## helper.py

```
print("in helper.py!")

a = 5

lst = [1, "hi"]

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

Output:

in helper.py!

5

[1, "hi", "hello"]

8

You can see the code for this example on repl.it at:

<https://repl.it/@LongNguyen18/ImportModulesPython>

# A Program with Multiple Modules

You can specify an **alias** for the imported module.

## main.py

```
import helper as hp

print(helper.a)
print(helper.lst)
print(helper.add(3, 5))
```

## helper.py

```
print("in helper.py!")

a = 5

lst = [1, "hi"]

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

Output:

in helper.py!

5

[1,"hi"]

8

# A Program with Multiple Modules

You can import all objects by using \*. No dot notation is needed to access the imported code in this case.

**main.py**

```
from helper import *  
  
print(a)  
print(lst)  
print(add(3, 5))
```

**helper.py**

```
print("in helper.py!")  
  
a = 5  
  
lst = [1, "hi"]  
  
def add(x, y):  
    return x + y
```

Output:  
in helper.py!  
5  
[1,"hi"]  
8

# Lab I

Write the Student class which has two instance variables: name(str) and gpa(float).

Write the average\_gpa function which accepts a list of Student objects and returns the average gpa.

Write the main method and:

- 1) Create a Student object and store it in a variable. Print out name and gpa of the Student object using the dot notation.
- 2) Create a list of three Student objects. Use a for loop to print out the names.
- 3) Call average\_gpa and make sure it works by printing out the average gpa.

# Lab I

Modify the previous lab by putting Student class and the average gpa function in a different module(.py).

Make the necessary import statement to make your code runs correctly.

# References

- I) Halterman, Richard. Fundamentals of Python Programming.  
Southern Adventist University.