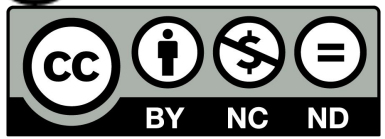
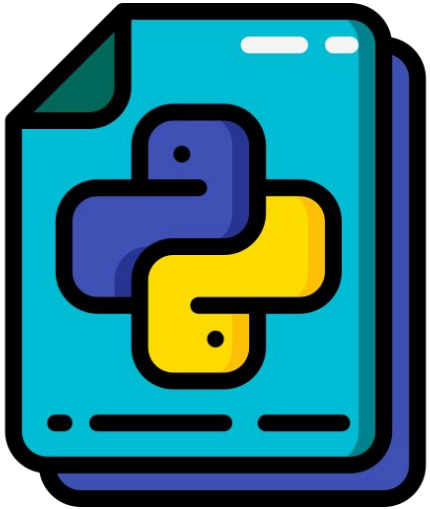
Python Class

An AI Commons initiative



Why Python?

1. Easy to write and understand
2. Fast to learn
3. Compatible with C/C++
4. Vast amount of libraries and community support
5. One of the main tools for modern Data Science and Machine Learning



Previous knowledge

1. Python Syntax
    a. operators, formats, indentation, print
2. Variables
    a. int, float, boolean, strings
3. Data Structures
    a. strings, lists, tuples, dictionaries
4. Flow Control
    a. for, while loops, if and else

Using for loops with lists and dictionaries

1. Let's build a test_list list of the first 5 squares of integers f(x)=x^2

```
test_list = [x**2 for x in range(5)]
test_list
```

```
[0, 1, 4, 9, 16]
```

2. Now let's build a loop_list list that enumerates and appends the integers and their squared values to the loop_list

```
loop_list = []

for i, x in enumerate(test_list):
    loop_list.append((i,x))

loop_list
```

```
[(0, 0), (1, 1), (2, 4), (3, 9), (4, 16)]
```

3. Let's build a dictionary of alphabet letters, with the key being their order value

```
test_dict = {key:value for key, value in enumerate(['a', 'b', 'c'])}
test_dict
```

```
{0: 'a', 1: 'b', 2: 'c'}
```

4. Let's invert the dictionary, with the letters as the keys, and their order index as the value

```
loop_dict = {}

for i in test_dict:
    value = test_dict[i]
    loop_dict[value] = i # Inverting the keys and values

loop_dict
```

```
{'a': 0, 'b': 1, 'c': 2}
```

Getting help

Python comes by default with a **help** function that documents libraries, modules, classes, functions, etc. The **help** function is very useful when you want to know how to use a command, or a function. It uses the following syntax:

```
help(something)
```

```
[9]  1  help(print)

    Help on built-in function print in module builtins:

    print(...)
        print(value, ..., sep=' ', end='\n', file=sys.stdout, flush=False)

        Prints the values to a stream, or to sys.stdout by default.
        Optional keyword arguments:
        file:  a file-like object (stream); defaults to the current sys.stdout.
        sep:   string inserted between values, default a space.
        end:   string appended after the last value, default a newline.
        flush: whether to forcibly flush the stream.
```

You can even build help for your own functions!

```
def square_function(x):
    """ Takes x and returns the square value of x. """
    return x**2

square_function?
```

```
Signature: square_function(x)
Docstring: Takes x and returns the square value of x.
File:      /content/<ipython-input-5-3d677340a47b>
Type:      function
```

Functions

- While programing, it's important to keep the code simple and easy to understand.
- Functions allow this by allowing to run blocks of code in a single line.
- They are used to write code that can be reused.

Example

So, imagine you have a code like this:

```
print("Working")

"""

Some long and/or complicated process

"""

print("Finished!")
```

What happens if we want to run this code repeatedly?

Functions

Instead of repeating the process, lets define this code inside a function!

```
def worker_function():
    print("Working")

    """

    Some long and/or complicated process

    """

    print("Finished!")
```

Let's call it a couple of times

```
print("Work once")
worker_function()

print("Work again")
worker_function()
```

```
Work once
Working
Finished!
Work twice
Working
Finished!
```

Assignment

Functions can also be assigned to a variable

```
w = worker_function

print(type(w))

w()
```

```
<class 'function'>
Working
Finished!
```

Arguments

The parenthesis () allows to pass arguments to the function.

```
my_function(arg_1, arg_2, ..., arg_n)
```

And can return multiple results

```
return output_1, output_2
```

General function syntax

```
def function(arg_1, arg_2):
    process # Whatever your function is doing
    return output_1, output_2
```

```
def function():
    this_is_inside_the_function
and_this_is_not
```

Examples

```
# Returns the square of x

def square(x):

    sqr = x**2

    return sqr

# Returns the cube of x

def cube(x):

    qbe = x**3

    return qbe

# Returns the sum of two functions aplied to x

def func_sum(x, func1, func2):

    return func1(x) + func2(x)

func_sum(2, square, cube)
```

```
12
```

Default values

A function arguments can also be defined with default values

```
# Returns the nth exponential of x

def exponential(x, n=0): # If not declared, n will take de value 0

    return x**n

exponential(2)
```

```
1
```

All default values have to be defined at the end of the arguments list. If you define an argument with a default value before one without a default value you will get an error.

Introducing Scope

What is the output of the following cell?

```
x = 0

def x_10():
    x = 10
    return x

print(x_10())
print(x)
```

```
10
0
```

Scope definition

When calling a function, its variables are stored inside a temporal environment called *namespace.*

The concept of where these variables are defined is called **scope**.

In Python there are three levels of scope:

1. Global: Valid for all the main space
2. Local: Valid only inside the function
3. Built-in: Variables that come with Python

Example of Scope

When calling a variable inside a function, Python will search for it's name in the local variables and then in the global variables.

```
e = 2.71828

def e_exp(x):
    return e**x

e_exp(2)
```

```
7.3890461584
```

To modify a global variable inside a function its necessary to declare it as global.

Let's fix the function x_10

```
x = 0

def x_10():
    global x
    x = 10
    return x

print(x_10())
print(x)
```

```
10
10
```

Arguments of variable length

Sometimes, a function need to be able to receive a variable number of arguments.

In Python this can be done by defining the arguments *args or *kwargs.

```
def mult(*args):
    x = 1
    for arg in args:
        x *= arg
    return x

mult(1,2,3,4)
```

```
24
```

```
def print_info(**kwargs):
    print(kwargs.keys())
    print(kwargs.values())


print_info(a=1, b=2, c=3)
```

```
dict_keys(['a', 'b', 'c'])
dict_values([1, 2, 3])
```

Lambdas

Lambdas are similar to functions, but their syntax is shorter and can have only one expression. They are mostly used for fast data manipulation.

Think of it as writing a function f(x)=x^2 in the following example:

The function

```
def square(x):
    return x**2
```

Can be replaced by the lambda

```
square = lambda x: x**2
```

Their general syntax is

```
lambda_name = lambda arg_1, arg_2 : process
```

Examples

```
def exp_factory(n):
    return lambda x: x**n

square = exp_factory(2)

cube = exp_factory(3)

print(square(4), cube(4))
```

```
16 64
```

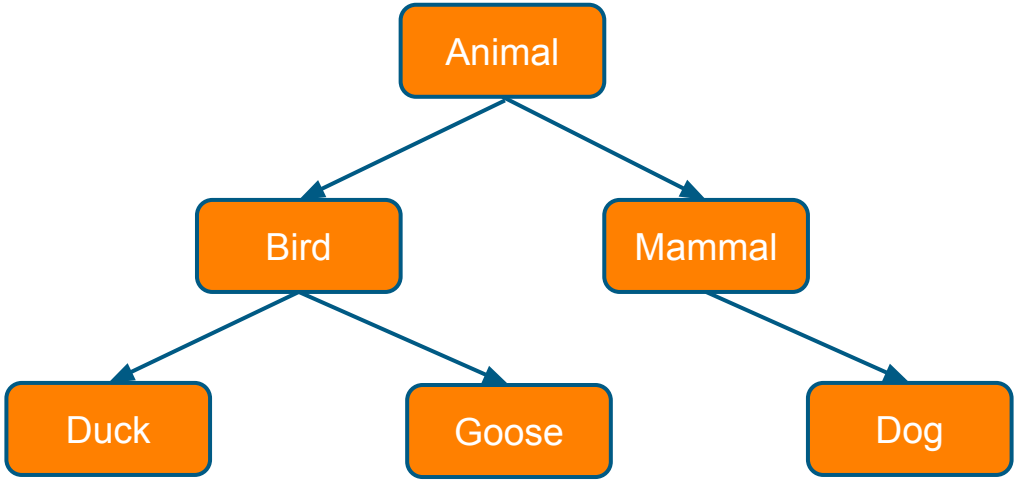
```
m = map(lambda x: x**2, [1, 2, 3, 4, 5])

f = filter(lambda x: x>2, [1, 2, 3, 4, 5])

print(list(m), list(f))
```

```
[1, 4, 9, 16, 25] [3, 4, 5]
```

Additional Material

Object-Oriented Programming (OOP)

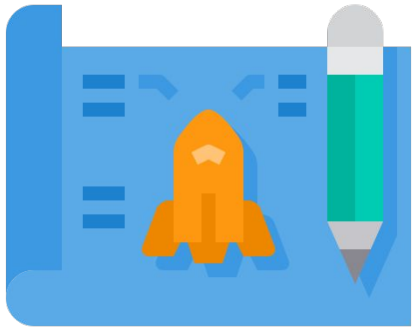
It's one of many programing paradigms to structure programs, where each behavior is associated to an object actions and properties. It helps structure and reduce code size.



Classes

Data structures, like integers and strings, allow us to store, manipulate and represent data. But they might be a bit too simple to represent more complex structures.

A class helps by allowing to organize this data while defining its behavior and interactions.



Classes in python

A class in Python can be defined by the following syntax:

```
class MyClass:

    process
    ...
```

An object can be instantiated (or built) by

```
object = MyClass()
```

Methods

Methods are functions that are associated to a class and can be called by its instances by appending them to the class as shown in the syntax below.

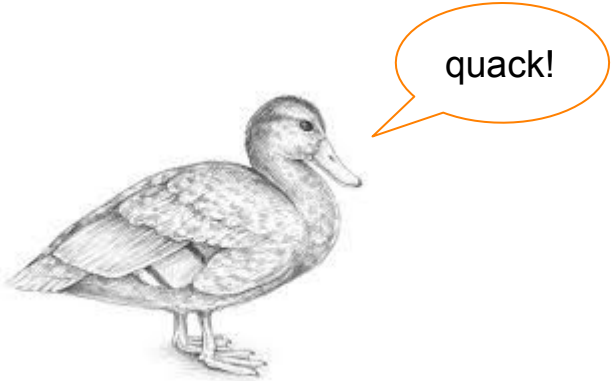
```
TheClass.the_method(args)
```

They are defined inside the class with the syntax **def** just like functions.

```
class TheClass:

    def the_method(args):
        process
```

Example

Let's make a digital duck that quacks




```
class Duck:

    def quack(self, n_quacks=1):
        for _ in range(n_quacks):
            print('quack')

ducky = Duck()

ducky.quack(3)
```

```
quack
quack
quack
```

Self

For a class, **self** represents it's instance, so by using **self** you are indicating Python that you are accessing the methods and variables of that instance.

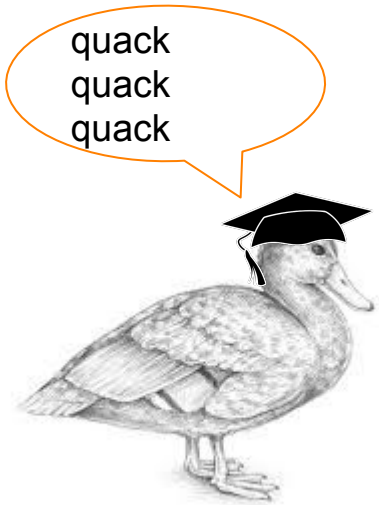
To store a variable inside an instance, we can do the following:

```
MyInstance.class_variable = 'Some values'
```

We can use **self** inside the class definition to indicate Python that we want to save them inside the class instance.

Example

Let's make a smarter duck




```
class SmartDuck:

    def quack(self, n_quacks=1):
        self.n_quacks = n_quacks
        for _ in range(n_quacks):
            print('quack')

    def count_quacks(self):
        print('I last quacked {} times'.format(self.n_quacks) )

smart_ducky = SmartDuck()

smart_ducky.quack(1)
smart_ducky.quack(2)

smart_ducky.count_quacks()
```

```
quack
quack
quack
I last quacked 2 times
```

Special Methods

Special methods are functions with a fixed name (usually with a double underscore) that have some special properties.

Perhaps the most important is the method **__init__**, that initializes an instance, but there are a lot more of them.

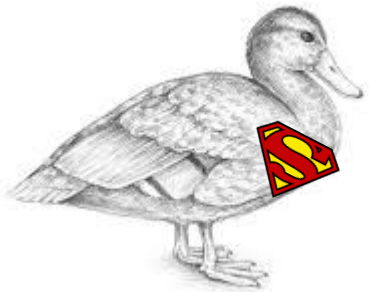
Let's further improve the duck so that each duck has a name, a weight, can remember all the times it has quacked and can add itself to other ducks to make a new duck.

Examples

```python
class SuperDuck:

    def __init__(self, name, weight):
        self.name = name
        self.weight = weight
        self.cuacks = 0

    def cuack(self, n_cuacks=1):
        for _ in range(n_cuacks):
            print('quack')
            self.cuacks += 1

    def count_cuacks(self):
        print('I have quacked {} times'.format(self.cuacks))

    def greet(self):
        print('Hi! my name is {} and I weight {} kg.'.format(self.name, self.weight))

    def __add__(self, other):
        new_name = 'Super {}{}'.format(self.name, other.name)
        new_weight = self.weight + other.weight

        return SuperDuck(new_name, new_weight)
```

```python
tomy = SuperDuck('tomy', 5)
samy = SuperDuck('samy', 4)

tomy.greet()

super_duck = tomy + samy

super_duck.greet()
```

```
Hi! my name is tomy and I weight 5 kg.
Hi! my name is Super tomysamy and I weight 9 kg.
```



Inheritance

Inheritance is a property of classes that allow us to make new classes with the properties and methods of another class.

This creates a hierarchy between the classes, so the class being inherited from is called the **Parent Class** and the class that inherits its the **Child Class**.

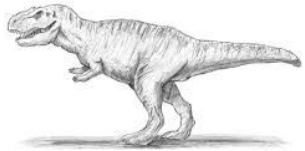
This allows to save a lot of code, and can help in organizing the classes functionalities.

The syntax for inheritance is:

```
class ChildClass(ParentClass):
    do_stuff()
```

Examples

```
class Dinosaur:

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

    def walk(self):
        print('walking')

    def roar(self):
        print('roar')

    def greet(self):
        print('My name is {} and I am a generic dinosaur'.format(self.name))
```



```
class TRex(Dinosaur):

    def attack(self):
        print('bite')
```

```
terry = TRex('terry', 14000)

terry.greet()

terry.attack()
```

```
My name is terry and I am a generic dinosaur
bite
```

Method Override

In the inheritance, a Child can override the Parent methods and properties.

To do this, it's necessary to redefine the method in the Child with the same name. When overriding, the original Parent method can still be accessed with **super()**

Let’s Fix the TRex class with overriding

Overriding example

```
class TRex(Dinosaur):

    def __init__(self, name, weight, theeth):

        super().__init__(name, weight)

        self.theeth = theeth

    def attack(self):
        print('bite')

    def greet(self):
        print('My name is {} and I am a T-Rex'.format(self.name))

    def old_greet(self):
        super().greet()
```

```
terry = TRex('terry', 14000, 100)

terry.greet()

terry.walk()

terry.old_greet()
```

```
My name is terry and I am a T-Rex
walking
My name is terry and I am a generic dinosaur
```

Questions?

Further References

- Geeks for Geeks Python

  https://www.geeksforgeeks.org/python-programming-language/

- Python Practice Book

  https://anandology.com/python-practice-book/