Lesson 3

# TUPLES

# Immutable groups of object. Parenthesis instead of square brackets.

names = ("Roger", "Syd")

names[0]  # Roger

names.index("Roger")  # 0

len(names)  # 2

print("Roger" in names)  # True

names[0:2]  # Roger Syd

sorted(names)  # It will create a new tuple (like it did for lists)

newTuple = names + ("Tina",) # You can create a new one, but not modify the original tuple

# DICTIONARIES

# Allows to create key-value pairs. The key and value can be anything. Curly braces are used.

dog = {"name":"Roger", "age":8, "color":"green"}

print(dog["name"]) # Roger

dog["name"] = "Syd" # It rewrites the value of name

print(dog.get("name")) # Syd

print(dog.get("color", "brown")) #If it cannot find the color in the dictionary, it is going to return the default value added (brown in this case)

print(dog.pop("name"))  #It prints it and then remove it

print(dog)

print(dog.popitem()) #It is going to retrieve, print and remove the last item key-value added

print(dog)

print("color" in dog)  # To find out if a key is contained in a dictionary

print(dog.keys())  #To show all the keys --> print(dog)

print(list(dog.keys()))  #To list only the keys, without title --> ['age']

print(dog.values())

print(list(dog.values()))

print(dog.items())

print(list(dog.items()))

print(len(dog))

dog["favourite food"] = "meat" # To add items

print(dog)

del dog["age"]  # To delete items

print(dog)

dogCopy = dog.copy()

# SETS

# Python data structures. Similar to tuples, but they are mutable and not ordered. Similar to dictionaries, but they do not have keys. Uses curly braces without keys.

set1 = {"Roger", "Syd"}

set2 = {"Roger", "Luna"}

set3 = {"Roger"}

intersect = set1 & set2

union = set1 | set2

difference = set1 - set2

subset = set1 > set3  # True

print (intersect, union, difference, subset, len(set1))

# A set cannot have two same elements.

# FUNCTIONS

# Set of instructions that we can run when needed.

def hello(name="my friend", age=15):

  print("Hello " + name + ", you are " + str(age) + " years old!")

hello("Beau", 34)   # To call the function

hello("Elettra", 28)

hello()

# A function can accept more than one parameter, indicated inside the parenthesis. I specify them as arguments when I call the function. Default arguments can be specified (my friend in this case)

# name = parameters

# "Beau" or "Elettra" = arguments

def change(value):

  value = 2

val = 3

change (val)

print (val)  # It is going to be 3 because these kind of objects are immutable. Even if you change it inside a function, it is not gonna change outside.

def change(value):

  value["name"] = "Elettra"

val1 = {"name":"Beau"}

change (val1)

print (val1)  # Dictionaries are mutable.

# return statement: the function ends. If I write only return, it will end and return nothing.

def hello1(name):

  if not name:

    return

  print("Hello " + name + "!")

hello1(False) # The function will just stop and do nothing

hello1("Beau") # The function will print

def hello2(name):

  print("Hello "+ name + "!")

  return name, "Beau", 8

print(hello2("Syd"))  # The function will run and it will print everything returned

# VARIABLE SCOPE

# When you declare a variable, the visibility of the variable depends on where you declared it

# If i declare a variable before the function, it is global and I can use it both inside the function, both outside

thing = 8

def test5():

    print(thing)

print(thing)

test5()

# If I declare the variable inside the function, it is a local function and I can use it only inside the function.

# NESTED FUNCTIONS

# Functions defined inside another function. They are visible only inside the function.

def talk(phrase):

  def say(word):

    print(word)

  words = phrase.split(" ")  #It split the text at each space, creating a list called words

  for word in words:

    say(word)

talk("I am going to buy the milk")

def count():

  count = 0

  def increment():

    nonlocal count  # Since count is not local, I need to declare it to access it inside the subfunction

    count = count +1

    print(count)

  increment()

count()

# CLOSURES

def counter():

    count = 0

    def increment ():

      nonlocal count

      count = count + 1

      return count

    return increment

increment = counter()  #Counter is returning the function increment and I am assigning it to a variable

print(increment())  # In this way I can call the inner function and it is not going to reset to zero. It still has access to the count variable even if the counter function is not active anymore.

print(increment())  # 2

print(increment())  # 3

print(increment())  # 4

# OBJECTS

# Everything in Python is an object

kick = 8  # The variable kick has access to methods available for integers

print(kick.real)

print(kick.imag)

print(kick.bit\_length())

list = [1,2] # The variable list has access to methods available for lists

list.append(3)

list.pop()

print(id(list)) # Location of the object

# Some objects are mutable and some not.

# age = 8

# age = age + 8

# In this case it is going to create a new variable. Instead for a dictionary, it is going to change the dictionary itself.

# LOOPS

# While and For

condition = True

while condition == True:

  print("The condition is True") # This

  # example is an infinite loop

  condition = False # It is going to run only

  # once

count\_x= 0

while count\_x <= 10:

  print("Ciao " + str(count\_x))

  count\_x = count\_x + 1

items\_d = [1, 2, 3, 4]

for items\_d in items\_d:

  print(items\_d)   #Clever way to iterate things

for bbb in range(15):

  print(bbb)

balls = ["basket", "football", "tennis", "volley"]

for index, item in enumerate(balls):

  print (index, item)   #To print both index and item

# CONTINUE AND BREAK

# While and for can be interrupted and continued

items = [1,2,3,4]

for item in items:

    if item == 2:

      continue # It does not print 2 because it does not run any code after continue and goes on with another iteration.

    print(item)

for item in items:

    if item == 2:

      break # It prints only 1 because it stops the cycle completely.

    print(item)

# CLASSES

# A class is a type of an object. It is possible to create them and define methods.

class Dog:

    def \_\_init\_\_(self, name, age):

      self.name = name

      self.age = age

    def bark(self):

      print("woof!")

roger = Dog("Roger", 8)  # Assign the new class to the object

print(type(roger))  # <class '\_\_main\_\_.Dog'>

print(roger.name)

print(roger.age)

roger.bark()  # Calling bark already prints woof! so I do not need to add print.

# CLASS INHERITANCE

class Animal:

  def walk(self):

    print("Walking...")

class Dog(Animal):

    def \_\_init\_\_(self, name, age):

      self.name = name

      self.age = age

    def bark(self):

      print("woof!")

Lessie = Dog("Lessie", 4)

Lessie.walk()   # It inherits the method from Animal class

True

Roger

Syd

green

Syd

{'age': 8, 'color': 'green'}

('color', 'green')

{'age': 8}

False

dict\_keys(['age'])

['age']

dict\_values([8])

[8]

dict\_items([('age', 8)])

[('age', 8)]

1

{'age': 8, 'favourite food': 'meat'}

{'favourite food': 'meat'}

{'Roger'} {'Roger', 'Luna', 'Syd'} {'Syd'} True 2

Hello Beau, you are 34 years old!

Hello Elettra, you are 28 years old!

Hello my friend, you are 15 years old!

3

{'name': 'Elettra'}

Hello Beau!

Hello Syd!

('Syd', 'Beau', 8)

8

8

I

am

going

to

buy

the

milk

1

1

2

3

4

8

0

4

2565876129280

The condition is True

Ciao 0

Ciao 1

Ciao 2

Ciao 3

Ciao 4

Ciao 5

Ciao 6

Ciao 7

Ciao 8

Ciao 9

Ciao 10

1

2

3

4

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

0 basket

1 football

2 tennis

3 volley

1

3

4

1

<class '\_\_main\_\_.Dog'>

Roger

8

woof!

Walking...