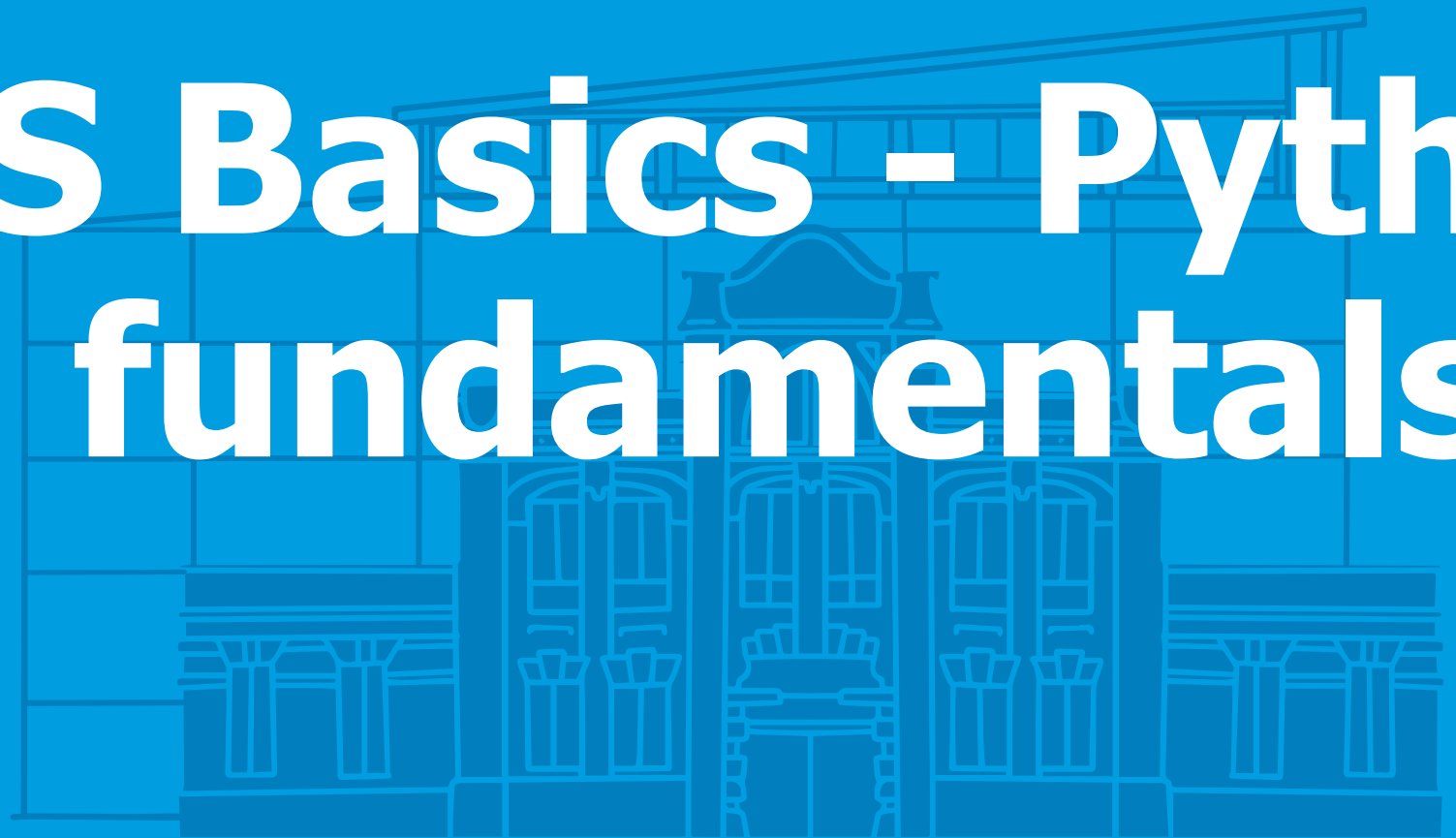
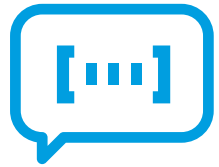
netmind
a bts company

MI1762

OS Basics - Python fundamentals



netmind
a bts company



A. Introduction

01 Standard Library

02 Data Types

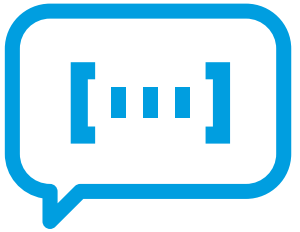
03 Flow control structures, loop

04 Lists

05 Files

06 Regular Expressions

netmind
a bts company



Standard Library

01

Standard Library

- It is a set of modules, some made in C and others in Python that are incorporated into the interpreter.
- It includes a series of common functionalities that do not need to be imported
  - https://docs.python.org/3.6/libraryiindex.html

Built-in Functions

https://docs.python.org/3.10/library/functions.html

A
abs()
aiter()
all()
any()
anext()
ascii()

B
bin()
bool()
breakpoint()
bytearray()
bytes()

C
callable()
chr()
classmethod()
compile()
complex()

D
delattr()
dict()
dir()
divmod()

E
enumerate()
eval()
exec()

F
filter()
float()
format()
frozenset()

G
getattr()
globals()

H
hasattr()
hash()
help()
hex()

I
id()
input()
int()
isinstance()
issubclass()
iter()

L
len()
list()
locals()

M
map()
max()
memoryview()
min()

N
next()

O
object()
oct()
open()
ord()

P
pow()
print()
property()

R
range()
repr()
reversed()
round()

S
set()
setattr()
slice()
sorted()
staticmethod()
str()
sum()
super()

T
tuple()
type()

V
vars()

Z
zip()

_
__import__()

Predefined constants

https://docs.python.org/3.10/1ibrary/constants.html

- `False`
- `True`
- `None`
- `Notimplemented`
- `Note`
- `Ellipsis`
- `debug`

Creating the first Python program

Write the code

- Create a new project (folder)
- Create a new file main.py
- Add the this code:

```
print("Hello World!")
```

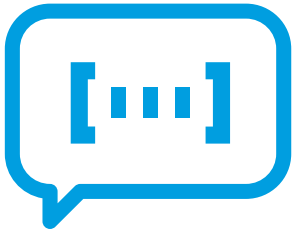
Execute the code

- Open a terminal console in the directory and execute the code

```
python main code. py
```

netmind
a bts company

02



Type of data

Basic Data Types

String

- "This is a string"
- 'This is a string'

Boolean (False, True, None)

- Boolean Operations:
- and, or, not

Comparisons

- <
- <=
- >
- >=
- ==
- !=
- is # identity
- is not # not identical

Numeric Types

- int
- float
- complex

Arithmetic Operators

- x + y
- x - y
- x*y
- x / y
- x // y # integer division
- x % y

Arithmetic Functions

- abs(x)
- int(x)
- float(x)
- complex(re, im)
- c.conjugate() # conjugate a complex number c
- divmod(x, y) #(x // y, x % y)
- pow(x, y) # x to the power of y
- x **y #x to the power of y

Bitwise functions

- x | y # bitwise or of x and y
- x ^ y # bitwise exclusive or of x and y
- X & y # and bitwise and of x and y
- x << x # shifted left by n bits (1) (2)
- X >> x # shifted right by n bits (1)(3)
- ~x # the bits of x inverted

Formatting Strings

- It is achieved using the **strformatO** method:

  https://docs.python.orginibraryistring.html#format-string-syntax

- You can insert the text to replace in the string **between {}**:

  'Coordinates: {latitude}, {longitude }'.format(latitude='137.24N1', longitude='1-115.81W1')

- In version 2, the operator %

  '%d'%(42,)

  '{ :d }'format(42)

- More information at https://pyformatinfoi

Formatting Strings

There is a set of conversion flags

- '!s' calls str() on the value
- '!r' calls reprO
- '!a' calls ascii()

Examples

- "Harold's a clever {0!s} " # Calls strO on the argument first
- "Bring out the holy {name!r}" # Calls reprO on the argument first
- "More {!a}" # Calls ascii() on the argument first

Formatting Strings

Lists

They are another type of sequences in Python, similar to an array

https://diveintopython3.net/native-datatypes.html#lists

They are handled in a similar way to an array

Declaration:

- list1 = ['physics', 'chemistry', 1997, 2000]
- list2 = list(range(4,10,2))

Access:

- list1 [2]=27
- list1 [2] # Search from beginning (0)
- list1[-2] # Search from final
- list1[1:] # Allow range
- list1 [1:3] # Allow range

Delete:

- del list[2]

Actions on lists:

- len([1, 2, 3]) #Length
- [1, 2, 3] + [4, 5, 6] # Concatenation
- ['Hi!'] * 4 # Repeat → ['Hi! ', 'Hi!', 'Hi!', 'Hi!']
- 3 in [1, 2, 3] # Search → True
- for x in [1, 2, 3]: print x # iteration → 1 2 3

Dictionaries

They are lists of key-value sets (a hash)

```
dict = {'Narre': 'Zara', 'Age': 7, 'Class': Tirst'}
```

Access and Update

- dict['Name’]

Delete :

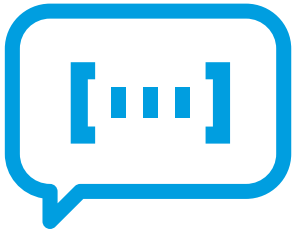
- del dict['Name']

Length:

- len(dict)

Playing with lists, tuples and dictionaries

netmind
a bts company



Flow control structures, loop

02

Control Structures

There are the following expressions:

- If
- if else
- if elif else
- While
- For
- There is no switch

Control Structures: if

```
var1 = 100
if varl :
   print ("1 - Got a true expression value")
```

Control Structures: if else

```
if varl :
   print ("1 - Got a true expression value")
   print (varl)
else:
   print ("1 - Got a false expression value")
   print (varl)
```

Control Structures: if elif else

```
if var == 200:
   print ("1 - Got a true expression value")
   print (var)
elif var == 150:
   print ("2 - Got a true expression value")
   print (var )
elif var == 100:
   print ("3 - Got a true expression value")
   print (var)
else:
   print ("4 - Got a false expression value")
   print (var)
```

Control Structures: while

```
count = 1
while (count < 9):
   print ('The count is:', count)
   count = count + 1
```

```
count = 0
while count < 5:
   print (count, " is less than 5")
   count = count + 1
else:
   print (count, " is not less than 5")
```

Control Structures: for

```
for letter in 'Pythoni: # First Example
  print ('Current Letter :', letter)


fruits = ['banana', 'apple', 'mango']
for fruit in fruits: # Second Example
  print (' Current fruit :', fruit)



fruits = ['banana', 'apple', 'mango']
  for index in range(len(fruits)):
  print ('Current fruit :', fruits[index])
```

Control Structures: for

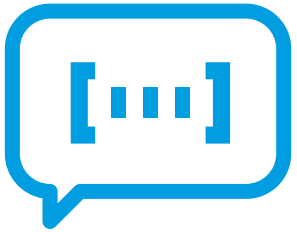
```
for num in range(10,20): #to iterate between 10 to 20
  for i in range(2,num): #to iterate on the factors of the number
    if num%i == 0: # to determine the first factor
      j=num/i #to calculate the second factor
      print ('%d equals %d * %d' % (num,i,j))
      break #to move to the next number, the #first FOR
else: # else part of the loop
  print (num, lis to prime number')
```

Something more complex

Create a script that goes through a list with the months of the year and displays them on the screen:

- In ascending order
- In descending order
- From 3 to 3
- As long as its equivalent number is a prime number

netmind
a bts company



Lists

04

List Comprehension

Let's see an example of building a list with this system

```
Celsius = [39.2, 36.5, 37.3, 37.8]
Fahrenheit = [ ((float(9)/5)*x + 32) for x in Celsius 1

list = [(x,y,z) for x in range(1,30) for y in range(x,30) for z in range(y,30) if
x**2 + y**2 == z**2]
```

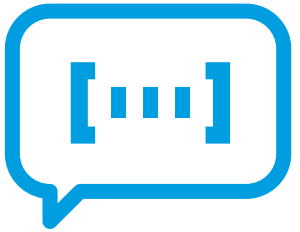
Generating lists

Generates a list with the first 10 sum numbers:

- sum(1) = 1
- ...
- sum(9) = 1+2+3+4+5+6+7+8+9 = 45
- sum(10) = 1+2+3+4+5+6+7+8+9+10= 55

netmind
a bts company

05



Files

Files

Python allows to work with files in a very simple way

https://docs.python.org/3/tutorial/inputoutput.html

To handle the files we will need the path and the opening mode

- **open**(filename, mode)
- **mode:**
  - 'r': read-only
  - 'w' write-only (an existing file with the same name will be deleted)
  - `a` opens the file to add; data written to the file is automatically appended to the end.
  - `r+' opens the file for both reading and writing.
  - `b' added to mode opens the file in binary mode: data is read and written as object bytes.

Files

Writing:

```
string="my text"
fobj_out = open("ad_lesbiam.txt","w")
fobj_out.write(string)
fobj_out.close()
```

Reading:

```
fobj_in = open("ejemplo.txt")
for line in fobj_in:
  print line.rstrip()
fobj_in.close()
```

As list:

```
lines= open("sample.txt").readlines()
```

As string:

```
text = open("sample.txt").readO
print text(16:34)
```

Files

Deleting:
Use the **remove or unlink** method of the **os library**

- os.remove(path, *, dir_fd=None)

```
import os
os.remove(“/path/dile_name>.<txt>")
```

First check that it is a file

- os.path.isfile(“/path/to/file")

To delete (empty) directories

- os.rmdir(path,*, dir_fd=None)

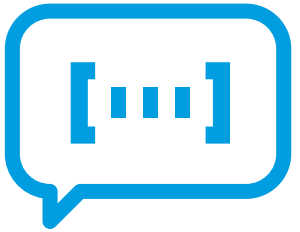
Reading and writing files

Write a script that copies the content of a file "sample.txt" into another "new.txt"

- Then display the content of new.txt line by line
- Finally delete new.txt

Do the same for an image

netmind
a bts company



Regular Expressions

06

Regular Expressions

Regular expressions will allow us to perform matches between different elements to know if they comply with a certain pattern

Python has a specific library for it: **re**

https://docs.python.org/3/library/re.html

Therefore it must be imported with the import command before using import re:

```
import re

if re.search("cat","A cat and a rat can't be friends."):
    print ("Some kind of cat has been found :-)")
else:
    print ("No cat has been found :-(")
```

Methods

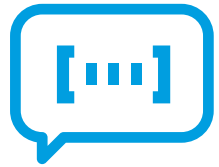
```
re.search(pattern, string, flags=0)
re.match(pattern, string, flags=0)
re.fullmatch(pattern, string, flags=0)
re.split(pattern, string, maxsplit=0, flags=0)
re.findall(pattern, string, flags=0)
re.finditer(pattern, string, flags=0)
re.sub(pattern, repl, string, count=0 , flags=0)
re.escape(pattern)
re.purge0
```

More information at
https://python-course.eu/advanced-python/regular-expressions.php

Working with regular expressions

Write a script that reads a "sample.txt" file and identifies the number of lines where a given text appears (stored in a variable)

netmind
a bts company



B. Functional Programming

07 Work with functions

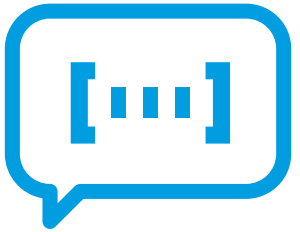
08 Positional and nominated arguments

09 Default arguments

10 Recursion

11 More about defining functions

12 Input and Call Parameters



Work with functions

07

Defining a function

To define a function we will use the reserved word **def**:

```
def sum(x, y):
        """Returns x + y IT II TI
        return x + y
```

You have to take into account the indentation so that it works!

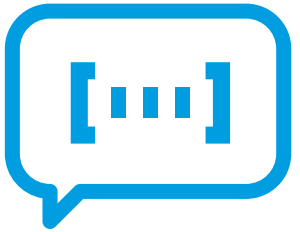
Calling a function

To call the function we simply have to name it and pass the necessary parameters

```
res=suma(2,3)
print(res)
```

*The order of the factors does not affect the product as long as the arguments are named

netmind
a bts company



Positional and nominated arguments

08

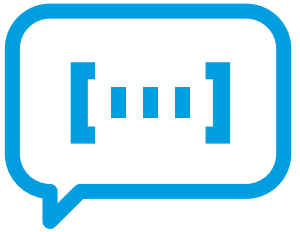
Positional and named arguments

```
res=sum (2,y=3)
print(res)
```

```
res=sum (x=2,y=3)
print(res)
```

```
res=sum (y=3,x=2)
print(res)
```

netmind
a bts company



Default arguments

09

Default arguments

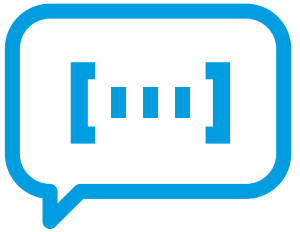
In order to have a variable number of arguments passed to a function without failing, it will be necessary to define default values.

```
def subtraction(x, y=5):
        """optional parameter"""
        return x – y

res=subtraction(5)
print(res)

res=subtraction(5,2)
print(res)
```

```
def multi(x=2,y=3):
        return x*y

res=multi()
print(res)

res=multi(2)
print(res)


res=multi(2,3)
print(res)
```

Playing with functions

Create a function that determines whether a phrase (stored in a function) is a palindrome or not.



Recursion

10

Recursion

Recursive functions in Python are done quite elegantly to quite elegantly

```
def factorial(n):
  if n == 1:
    return 1
  else:
    return n * factorial(n-1)

print(factorial(5))
```

Recursion

Let's see how calls happen inside the recursive function

```
def factorial(n):
  print("factorial has been called with n = " + str(n))
  if n == 1:
    return 1
  else:
    res = n * factorial (n-1)
    print("intermediate result for ", n, "*factorial(" ,n-1, "): ",res)
    return res

print(factorial(5))
```

Recursion

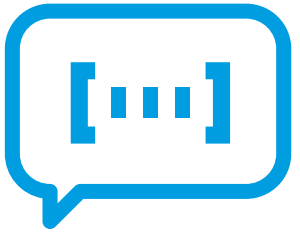
Let's see how calls happen inside the recursive function

```
factorial has been called with n = 5
factorial has been called with n = 4
factorial has been called with n = 3
factorial has been called with n = 2
factorial has been called with n = 1
intermediate result for 2 * factorial( 1 ): 2
intermediate result for 3 * factorial( 2 ): 6
intermediate result for 4 * factorial( 3 ): 24
intermediate result for 5 *factorial( 4 ): 120
```

Remove Duplicates

Create a function that, given a list of strings, removes the duplicated strings.

netmind
a bts company



More about defining functions

11

Keywords Arguments

- When using keyword arguments in a function call, the arguments are identified by the name of the parameter.
- This allows you to skip arguments or put them out of order because the Python interpreter is able to use the supplied keywords to match values with parameters.

```
def printme(str):
  "This prints a passed string finto this function"
  print(str)


# Now you can call printme function # Now you can call printme function
printme (str="My string")
```

Keywords Arguments

When using keyword arguments in a function call, the arguments are identified by the name of the parameter. This allows you to skip arguments or put them out of order because the Python interpreter is able to use the supplied keywords to match values with parameters.

```
def printme(str):
  "This prints a passed string into this function"
  print(str)

# Now you can call printme function
printme (str="My string")


def printinfo(name, age):
  "This prints a passed info into this function"
  print("{!s}",format(name))
  print("{!s}",format(age))
  return

# Now you can call printomfo function
printinfo (age=50,name="miki")
```

Arbitrary Argument Lists

- As in other high-level languages, it is possible for a function to expect to receive an arbitrary -unknown- number of arguments.
- These arguments will arrive at the function in the form of a tuple.
- To define arbitrary arguments in a function, precede the parameter with an asterisk (*):

```
def iterate_arbitrary_parameters(fixed_parameter, *arbitrary):
    print(fixed_parameter)

    # Arbitrary parameters are run as tupees
    for argument in arbitrary:
        print(argument)


iterate_arbitrary_parameters(' Fixed', 'arbitrary 1', 'arbitrary 2', 'arbitrary 3')
```

Unpack of argument lists

- An inverse situation to the previous one can also occur. That is, the function expects a fixed list of parameters, but these, instead of being available separately, are contained in a function expects a fixed list of parameters, but these, instead of being available separately, separately, are contained in a be available separately, are contained in a list or tuple.
- In this case, the asterisk sign (*) must precede the name of the list or tuple that is passed as a parameter during the function call.

```
def calculate(amount, discount):
  return amount - (amount *discount / 100)

data = [1500, 10]
print(calculate(*data))
```

Unpack of argument lists

- The same case can occur when the values to be passed as parameters to a function are available in a dictionary. Here, they should be passed to the function, preceded by two asterisks (**).

```
def calculate_emp(amount, discount):
  return amount - (amount * discount / 100)

data = {"discount": 10, "amount": 1500}
print (calculate emp_(**data))
```

Lambda Expressions

- When writing functional-style programs, you often need small functions that act as **predicates** or combine elements in some way.
- One way to write small functions is to use the **lambda expression**.
- Lambda takes a series of parameters and an expression that combines these parameters, and creates a small function that returns the value of the expression

```
# normal
def f(x):
  return x**2

print(f(8))

#lambda
g = lambda x: x**2

print(g(8))
```

```
# Other Examples
lowercase = lambda x: x.lower0

print_assign = lambda name, value: name +
'=' + str(value)
adder = lambda x, y: x+y
```

Lambda Expressions

- The **def option is preferred over lambda**
- One of the reasons is that lambda is quite limited in the functions it can define.
- The result has to be computable as a single expression, which means it cannot have if... elif ... else or try ... except statements.
- If you try to do too much in a lambda statement, you'll end up with an overly complicated expression that's hard to read.

```
#lambda
total = reduce lambda a, b: (0, a[1] + b[1]), items)[1]

#equivalent
def combine (a, b):
  return 0, a[1] + b[i]

total = reduce(combine, items)[1]
```

Documentation Strings

- A docstring is a string literal that occurs as the first statement in a module, function, class, or method definition. Such a docstring becomes the **__doc__** special attribute of that object.
- All modules should normally have docstrings, and all functions and classes exported by a module should also have docstrings.
- Public methods (including the **__init__** constructor) should also have docstrings.
- For consistency, you should always use **"""triple single quotes"""** in a docstrings, as it allows multi-lining.

  https://www.python.org/dev/peps/pep-0257

```
def kos_root():
  """Return the pathname of the KOS root directory."""
  global _kos_root
  if _kos_root: return _kos_root
  ...
```

Function Annotations

- Are expressions that invoke decorators that modify the behavior of a function

```
@helloGalaxy
@helloSolarSystem
def hello():
  print ("Helio, world!")
```

Function Annotations

- When it finds annotations the interpreter is as follows follow the below process
  - Add helloGalaxy on the annotation stack.
  - Add helloSolarSystem to the annotation stack.
- Then it does the standard processing for a function definition...
  - Compiles the code for helium into a function object (we call it functionObject1)
  - Attaches the name "hello" to functionObjecti.
- Then...
  - Pop helloSolarSystem off the annotation's stack,
  - pass functionObjectl to helloSolarSystem
  - helloSolarSystem returns a new function object (we call it functionObject2), and
  - ...
  - the interpreter binds the original name "hello" to functionObject2

Function Annotations

- Finally...
  - pops helloGalaxy off the annotation's stack,
  - passes functionObject2 to helloGalaxy
  - helloGalaxy returns a new function object (we call it functionObject3), and ... the interpreter binds the original name "hello" to functionObject3
  - the interpreter binds the original name "hello" to functionObject3

Defining a Decorator

- Decorators are defined in the same way as any other function, but they return functions instead of values

```
def helloSolarSystem(original_function):
  def new_function():
    original_function()
    # the () after "original_function" causes original_function to be called
    print("Hello, solar system!")
  return new_function


def helloGalaxy(original_function):
  def new_function():
    original_function()
    # the () after "original_function" causes original_function to be called
    print("Hello, galaxy!")
  return new_function
```

Defining a Decorator

```
# Now we can call helium using decorators
@helloGalaxy
@helloSolarSystem
def hello():
  print("Hello, world!")


# Here is where we actually *do* something!
hello()
```

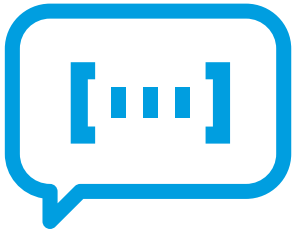
Passing arguments

For helio function with parameters

```
def hello(targetName=None):
  if targetName:
    print("Hello, " + targetName +"!")
  else:
    print("Hello, world!")
```

Define annotations that use those parameters

netmind
a bts company



Input and Call Parameters

12

The input function

- It allows capturing user input via the terminal

```
input([prompt])
```

- This will generate a prompt that will wait until the user enters a value and hits enter:

```
num = input('Enter a number: i)
print( num)
```

Call parameters

- When we call a python script it is desirable to capture the parameters of the invocation.
- This is achieved through the argument variable: **argv**
- To use it it is necessary to **import it from the sys** library

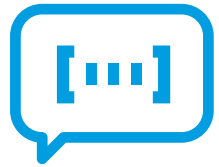
```
from sys import argv
script, first, second , third = argv


print("The script is called:", script)
print("Your first variable is:", first)
print("Your second variable is:", second)
print("Your third variable is:", third)
```

- The **first parameter will always be the name** of the script!

Creating a guessing game

- Define a script that receives two parameters: max and min.
- Randomly generate a number to guess.
- Then ask the user to guess the number, indicating that the guess is "smaller‟ or "bigger‟ than the secret number.
- If the user guesses it, the program must ask if the user wants to continue playing and repeat the process

netmind
a bts company



C. Object-oriented Programming

13 Defining a class

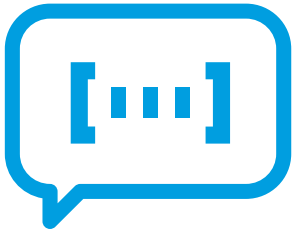
14 Methods and instance attributes

15 Methods and class attributes

16 Inheritance

17 Modules

netmind
a bts company



Defining a class

13

Defining a class

- In order to carry out object-oriented programming we will need to manage classes, methods, properties and objects> To define a class
- To define a class

```
class ClassName:
    'Optional class documentation string'
    #class_suite
```

- Creating an object

```
obj= ClassName()
```

Defining a class

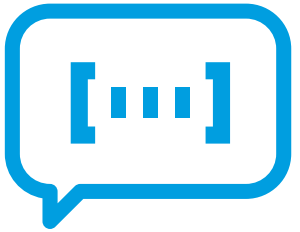
```
class Antenna0:
    color = ""
    length = ""

class Hair():
    color = ""
    texture = ""

class Eye():
    shape = ""
    color = ""
    size = ""
```

Model a user

- Create a class that models a user

netmind
a bts company

14



Methods and instance attributes

Instance methods and attributes

- The attributes of a class define the properties of the class
- Methods define the behavior that a class
- Instance methods and attributes are those that are defined in the class but can change from instance to instance.

```
class Employee:
 'Common base class for all employees'
 #this variable is static and is shared between instances of Employee
 empCount = 0

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

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

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

Instance methods and attributes

```
"This would create first object of Employee class"
empl = Employee("Zara", 2000)

"This would create second object of Employee class"
emp2 = Employee("Manni", 5000)
empl.displayEmployee0
emp2. displayEmployee0
print ("Total Employee %d" % Employee.empCount)


empl.age = 7 # Add an 'age' attribute.
empl.age = 8 # Modify age' attribute.
del empl.salary # Delete lsalary' attribute.
```

Instance methods and attributes

- In this case they are all the methods that are described in the class and that are not annotated with @classmethod.
- This would make it a method of the class and not of the instance
- Otherwise, methods are normal functions, except that they can use the self keyword (like this) to access inner methods or attributes inner methods or attributes

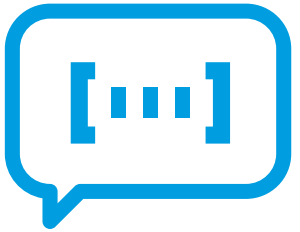
Instance methods and attributes

- The constructor will be called __init or __init__ and can be given predefined parameters

```
class Song(object):
  def __init__(self, lyrics):
    self.lyrics = lyrics

def sing_me_a_song(self):
  for line in self.lyrics:
    print(line)
```

```
Happy_bday = Song(["Happy birthday to you",
                   "I don't want to get sued",
                   "So stop right there"])


bulls_on_parade = Song(["They rally around the family",
                        "With pockets full of shells"])

happy_bday.sing_me_a_song()

bulls_on_parade.sing_me_a_song()
```

netmind
a bts company



Methods and class attributes

15

Class methods and attributes

- In the case of class methods they should be headed with it @classmethod annotation to @classmethod annotation

```
@classmethod
def introduce(cls):
  print("Hello, 1 am %s!" % cls)
```

- In the case of class attributes, it would be worth placing the attribute outside the definition of any

```
class method Employee:
  """This variable is static and is shared between instances of Employee"""
  empCount = 0
```

- These class attributes are maintained between executions of objects of the class

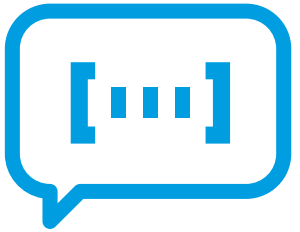
Playing with objects

We want to define a program that calculates the fastest car. For this we will use objects that model cars and engines:

- Car
  - brand: String
  - engine: Engine
  - diameter_wheels: int
  - position=0: int
  - move(time) : float # depending on the type of motor advances x meters/minute
    - Simple calculation: circ_tires*pi*rpm
- Engine
  - type: String
  - rpm: int

- Generates two car instances
- Simulates the fastest car to travel 10min

netmind
a bts company



Inheritance

16

Class methods and attributes

- To Inherit from a parent class, simply indicate the class as an attribute of the class being defined

  ```
  class Child(Parent):
  ```

- In Python we have the possibility of having multiple inheritance

  But it is not recommended to use multiple inheritance

- From the child class we can access the parent class with the `super()` method

Inheritance

```
class Parent: # define parent class
  parentAttr = 100
  def init (self):
    print ("Calling parent constructor")
  def parentMethod(self):
    print CCalling parent methon
  def setAttr(self, attr):
    Parent.parentAttr = attr
  def getAttr (self):
    print ("Parent attribute :", Parent.parentAttr)
```

Inheritance

```
class Child(Parent): # define child class
  def init (self):
    print ("Calling child constructor")
  def childMethod(self):
    print ('Calling child method')

c = Child() # instance of child
c.childMethod() # child calls its method
c.parentMethod() # calls parent's method
c.setAttr(200) # again call parent's method
c.getAttr() # again call parent's method
```

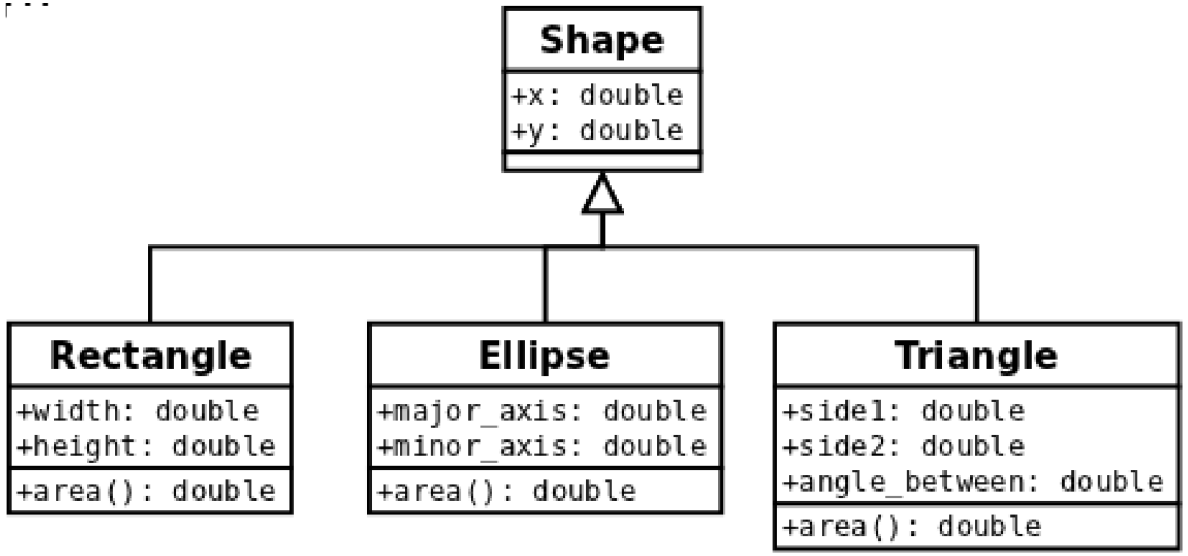
Multiple Inheritance

- Defined by indicating the parent classes as parameters to the definition

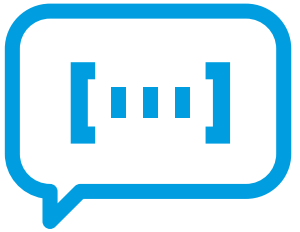
```
class A: # define your class A
 def __init__(self):
   self


class B: # define your class B
 def __init__(self):
   self


class C(A , B): # subclass of A and B
 def __init__(self):
   self
```

Playing with inheritance

- We want to implement a program that calculates areas, following the following UML diagram:



- Calculate the area of a rectangle, an ellipse and a triangle
- It would work for a square?



Modules

17

Modules

- Modules allow you to split your application code so that you can reuse application code.
- In Python, each of our **.py** files is called a module.
- Modules can be loaded using the import module name

```
import module_name
```

- Creating a module is as simple as creating a file with **`module_name.py`**

  https://docs.python.org/3/tutorial/modules.html

Packaged modules

- For a folder to be considered a package, it must contain an init file called **__init__.py**
- Packages can also contain other sub-packages
- Modules do not necessarily have to belong to a package

```
.
├── modulo1.py
└── paquete
    ├── __init__.py
    ├── modulo1.py
    └── subpaquete
        ├── __init__.py
        ├── modulo1.py
        └── modulo2.py
```

Modules

- You can import entire modules or functions, or specific values defined in a module

```
import module # import a module that does not belong to a package
import package.module1 # import a module that is inside a package
import package.subpackage.modulol
```

Namespaces

- To access (from the module where the import was made), to any element of the imported module, it is done through the namespace, followed by a dot **(.)** and the name of the element that you want to obtain.
- In Python, a namespace is the name that has been indicated after the word import, that is, the path **(namespace)** of the module:

```
print(module.CONSTANT_1)
print(package.module1.CONSTANT_1)
print(package.subpackage.module1.CONSTANT_1)
```

Aliases

- It is also possible to abbreviate the namespaces by means of an alias.
- To do this, during the import, the keyword is assigned as followed by the alias with which we will refer to that imported namespace in the future

```
import module as m
import package.modulol as pm
import package.subpackage.modulol as psm

print(m.CONSTANT_1)
print(pm.CONSTANT_1)
print(psm.CONSTANT_1)
```

Import modules without using namespaces

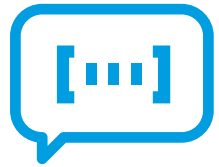
- In Python, it is also possible to import only the elements that you want to use from a module. To do this, use the from statement followed by the namespace, plus the import statement followed by the element to be imported

```
from package.modulol import CONSTANT_1
print(CONSTANT_1)


from package.modulol import CONSTANT_1 as Cl, CONSTANT_2 as C2
from package.subpackage. modulol import CONSTANT_1 as CS1, CONSTANT_2 as C52


print(C1)
print(C2)
print(CS1)
print(C52)
```

Creating modules

- Create a module that allows calculating any type of flat shape area.
- Then generate a script that asks for the type of shape to calculate and returns the answer

netmind
a bts company



D. Advanced Programming

18 Error handling, exception handling

19 Multitasking programs

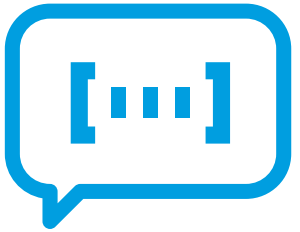
20 Standard Library Modules

21 Some useful libraries

22 Project structure

23 Module testing

netmind
a bts company



Error handling, exception handling

18

Error Control

- One of the main parts of any language is the possibility of managing errors
- For this it has a **Try-Except** structure
- The **try** will be the piece of code that will always be executed and only if it fails will it execute the **except**

```
while True:
  try:
    n = input("Please enter an integer: ")
    n = int(n)
    break
  except ValueError:
    print("No valid integer! Please try again ...")

  print("Great, you successfully entered an integer!")
```

Error Control

- There is the possibility of handling a finally

```
try:
  x= float(input("Your number: "))
  print(x)
  inverse = 1.0 ix
except ValueError:
  print("You should have given either an int or a float ")
except ZeroDivisionError:
  print("Infinity")
finally:
  print("There may or may not have been an exception.")
```

Error Control

- Python has a lot of exceptions that forces a program to raise an error when something goes wrong.

  https://www.programiz.com/python-programming/exceptions

- If you want to define a custom exception, simply inherit an Exception class and inherit the Exception

```
class Error(Exception):
    """Base class for other exceptions"""
    pass
```

Error Control

- Exceptions can also be raised with the command

```
raise exception(args)
```

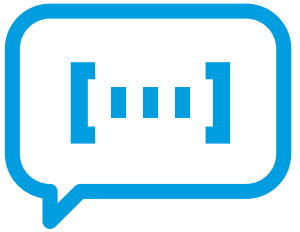
- **raise** without parameters raises the last exception

```
try:
  raise ValueError("I have raised an Exception")
except ValueError as exp:
  print("Error",exp)

try:
  raise ValueError
except ValueError as exp:
  print("Error",exp)
```

Handling exceptions

- Create a script that receives a file name as a parameter, check if it exists and count the number of lines it has
- If the file does not exist, it should show a prompt asking for a new name

netmind
a bts company



Multitasking programs

19

Multitasking Programs

- In Python you can manage different parallel threads of execution
- There are two modules to generate threads **Thread** and **Threading**
- The Threading module facilitates the task of creating and launching threads
  - https://docs.python.org/3/library/threading.html
- Constructor is

```
class threading.Thread(group=None, target=None, name=None, args=0, kwargs={}, *, daemon=None)
```

  - **group**: must be None; reserved for future extension when a ThreadGroup class is implemented.
  - **target**: is the object to be invoked by the runü method. The default value is None, which means that nothing is called.
  - **name**: is the name of the thread. By default, a unique name of the form "Thread-N" is constructed, where N is a decimal number.
  - **args**: is the argument tuple for the invocation of the target. Default is O.
  - **kwargs**: is a dictionary of keyword arguments to the target invocation. The default value is {}.

Multitasking Programs

```
import threading

def worker(num):
    """thread worker function"""
    printeWorker {}i.format(num))
    return

threads = []
for i in range(5):
    t = threading.Thread(target=worker, args=03)
    threads.append(t)
    t.start()
```

Threading Methods

start{)	Starts the thread activity.
run()	A method that represents thread activity.
join(timeout=None)	Wait until the thread ends.
name	String used for identification purposes only.
ident	The thread identifier of this thread, or None if the thread has not been started. It is a non-zero integer.
i s_a live()	Returns whether the thread is alive.
daemon	A boolean value indicating whether this thread is a daemon thread (True) or not (False). This must be set before start O is called, otherwise RuntimeError is raised.

Example

```
# -*- coding: utf-8 -*-

import threading
import time

exitFlag : O

class myThread (threading.Thread)z
   def init (self, threadlD, name, counter):
      threading.Thread.__init__(self)
      self.threadlD : threadlD
      self.name : name
      self.counter : counter

   def run(self):
      print("5tarting " + self.name)
      prínt_tíme(self.name, self.counter, 5)
      print("Exiting " + self.name)

   def print_time(threadName, delay, counter):
      while counter:
         if exitFlag:
            threadName.exit0
         time.sleep(delay)
         print("%s: alas" % (threadName,

   time.ctime(time.time0)))
         counter -= 1

# Create new threads
threadl : myThread(l, "Thread-1", 1)
thread2 : myThread(2, "Thread—2", 2)

# Start new Threads
thread1.start()
thread2.start()
threadl.join()
thread2.join()
print("Exiting Main Thread")
```

Signals between threads

- There are times when it is important to be able to synchronize operations in two or more threads.
- An easy way to communicate between threads is to use **Event objects**.
- An event handles an internal flag that calling processes can **set()** or **clear()** on.
- Other threads can **wait()** for the **set()** flag to be set, blocking progress until allowed to continue.
- More details at:

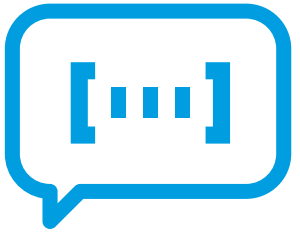
  https://pymotw.com/3/threading/

A multi-threaded script

Create a script that downloads the following images using 4 workers:

- https://farm5.staticflickr.com/4117/4787042405_37e548cf3a_o_d.jpg
- https://farm3.staticflickr.com/2375/2457990042_e6d6982cb2_o_d.jpg
- https://farm4.staticflickr.com/3149/3104818507_06cf582ba3_o_d.jpg
- https://farm3.staticflickr.com/2801/4084837185_4c12f32b1f_o_d_jpg

netmind
a bts company



Standard Library Modules

20

System modules

- The module allows us to access functionalities dependent on the Operating System. Above all, those that refer to the Operating System. Above all, those that refer us to information about its environment and allow us to manipulate the directory structure.

  https://docs.python.org/3/library/os.html

System modules

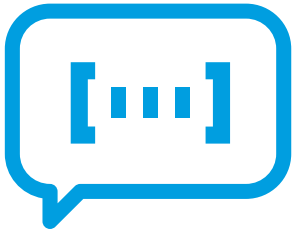
Know if a file or directory can be accessed	**os.access(path, access_mode)**
Know the current directory	**os.getcwdO**
Change working directory	**os.chdir(new_path)**
Change to root working directory	**os.chroot( )**
Change the permissions of a file or directory	**os.chmod(path, permissions)**
Change the owner of a file or directory	**os.chown(path, permissions)**
Create a directory	**os.mkdir(path{, mode])**
Create directories recursively	**os.mkdirs(path[, mode])**
Remove a file	**os.remove(path)**
Remove a directory	**os.rmdir(path)**
Recursively remove directories	**os.removedirs(path)**
Rename a file	**os.rename(current, new)**
Create a link symbolic	**os.symlink(path, destination_name)**

System modules

- See the complete list in the document: > PythonStandardModules.pdf
  https://docs.python.org/3/library/index.html

- Useful modules, packages and libraries
  https://wiki.python.org/moin/UsefulModules

netmind
a bts company

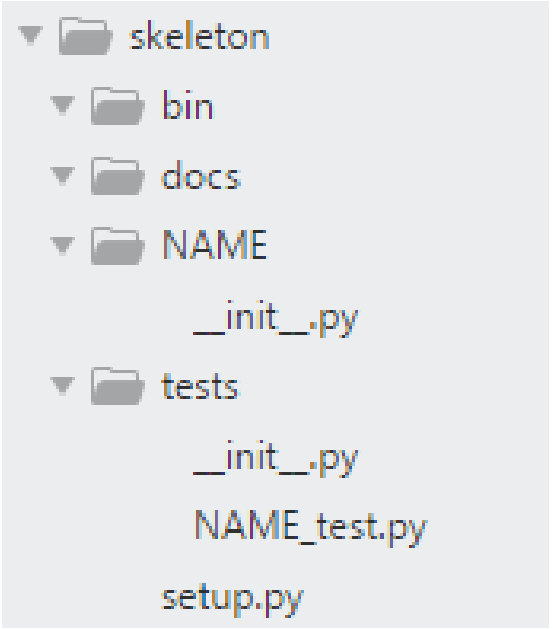


Project structure

22

Project skeleton

- This skeleton directory skeleton will contain all the basic elements needed to get a project up and running: layout, tests, modules and installation scripts.
- When creating a new project, simply copy this directory to a new name and edit the files to get started.



Project skeleton: setup.py

- The setup script is the hub of all activity from building, distributing, and installing modules using the Distutils

  https://docs.python.org/3/distutils/setupscript.html

- Contains the definition of what is needed for the module.
- To see the options:

```
python setup.py --help
```

- To install the modules

```
python setup.py install
```

Project skeleton: setup.py

```
try:
 from setuptools import setup
except ImportError:
 from distutils.core import setup

config = {
 'description': 'My Project',
 'author': 'Ricardo A',
 'url': 'URL to get it at.',
 'download_url': 'Where to download it.',
 'author_email': Tricardo@enmotionvalue.corni,
 'version': '0.11,
 'install_requires': triase'',
 'packages': ['NAME'',
 'scripts': [],
 'name': 'projectname'
}
setup(**config)
```

PIP

- The PyPA-recommended tool for installing Python packages.
  - PYPA: https://packaging.python.org/guides/tool-recommendations
  - PIP: https://pip.pypa.io/en/stable/

- To install any published module or package we will use

  ```
  pip install <module>
  ```

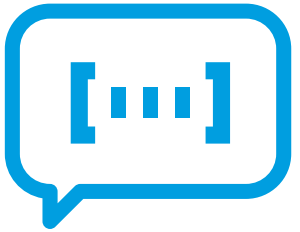
  https://packaging.python.org/en/latest/tutorials/installing-packages/

- Within a project we can use "**pip install .**" to install the package (dependenies)
  - If **requirements.txt** exists, this command can be used:

  ```
  pip install -r requirements.txt
  ```

Creating a project

- Create a skeleton described previously.
- Make a copy for a project named "calculator_areas"
- Install the dependencies



Module testing

23

Automated tests

- Python allows you to define unit tests to test the functionalities of a module
- The objective of the tests is to reduce the time needed to test the software, guarantee its regressivity and help identify possible errors
- Python has two main modules for testing :
  - unittest

    https://docs.python.org/3/library/unittest.html
  - nose: inherits and complements unittest

    http://nose.readthedocs.io/en/latest

    Needs to be installed using pip:

```
pip install nose
```

Writing a TestCase

- To generate a test case we will **import the nose tools** and the module to test.
- Then we will **generate as many functions as tests** we want to perform
- The functions must **start with test**
- Inside each function we will **make as many assertions as necessary** to perform the necessary checks
- List of assert methods:

  https://docs.python.org/3/library/unittest.html#assert-methods

- We will save the test in the project's **tests directory**
- Finally we will launch (in root) the nose command to search for and execute the tests of a project:

```
nosetests
```

Writing a TestCase

```
# class Room
class Room(object):
        def __init__(self, name, description):
                self.name : name
                self.description : description
                self.paths : {}

        def go(self, direction):
                return self.paths.get(direction, None)

        def add_paths(self, paths):
                self.paths.update(paths)
```

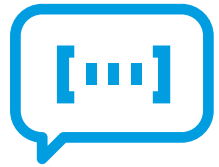
Writing a TestCase

```
# test para Room
from nose.tools import *
from ex47.game import Room


def test_room():
        gold : Room("GoldRoom","""" This room has gold in it you can grab. There's a X
door to the north.""")
        assert_equal(gold.name, "GoldRoom")
        assert_equal(gold.paths, £))
```

Generating a unit test

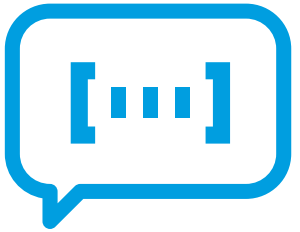
- Generate the unit tests for your area calculator project
- Run the tests

netmind
a bts company



Networking

netmind
a bts company



Sockets

24

Sockets

- Python provides two levels of access to network services.
  - At a low level, you can access basic socket support in the underlying operating system, allowing you to implement clients and servers for both connection-oriented and connectionless protocols.
  - It also has libraries that provide higher-level access to specific application-level network protocols, such as FTP, HTTP, etc.
- Sockets are the end points of a bidirectional communications channel.
- Sockets can communicate within a process, between processes on the same machine, or between processes on different continents.
- They can be implemented on several different channel types: Unix domain sockets, TCP, UDP, etc.
- The socket library provides specific classes for handling the common transports, as well as a generic interface for handling the rest.

The socket module

https://docs.python.org/3/library/socket.html

- To create a socket, you must use the socket.socket() function available in the socket module, which has the syntax:

```
s = socket.socket (socket_family, socket_type, protocol = 0)
```

  - **socket_family**: This is AF_UNIX or AF_INET, as explained previously.
  - **socket_type**: This is SOCK_STREAM or SOCK_DGRAM.
  - **protocol**: Normally omitted, defaulting to 0.

socket Methods

Method	Description
s.bind()	This method binds the address (hostname, port number pair) to the socket.
s.listen()	This method sets and starts the TCP listener.
s.accept()	This passively accepts the connection from the TCP client, waiting until the connection arrives (blocking).
s.connect()	Start the connection on the client side
s.recv()	Receive the TCP message
s.send()	Transmit the TCP message
s.recvfrom()	Receive the UDP message
s.sendto()	Transmit the UDP message

Sockets – Server

```
import socket


# create a socket object
serversocket = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)


# get local machine name
host = socket.gethostname()


port = 9999


# bind to the port
serversocket.bind(host, port)


# queue up to 5 requests
serversocketlisten(5)


while True:
   # establish a connection
   clientsocket, addr = serversocket.accept()


   print("Got a connection from Tos" Wo
str(addr))


   msg = 'Thank you for connectingT + "\r\n"
   clientsocket.send(msg.encode('asciii))
   clientsocket.close0
```

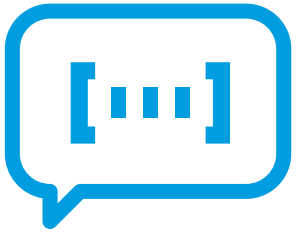
Sockets - client

```
import socket


# create a socket object
s = socket.socket(socket.AF_INET,
socket.SOCK_STREAM)


# get local machine narre
host = socket.gethostname()


port = 9999


# connection to hostname on the port.
s.connect((host, port))


# Receive no more than 1024 bytes
msg = s.recv(1024)
s.close()
print (msg.decode('ascii'))
```

Playing with sockets

- Create a list or a numerical type and send all the elements that are multiples of two from the client to the server, and make the latter stores them in a list.
- Encrypts party-to-party communication using a simple encryption/decryption

netmind
a bts company



Reading a URL

25

http module

- http is a package that compiles several modules to work with HyperText Transfer Protocol:
  - **http.client** is a low-level HTTP protocol client; for opening top-level URLs urllib.request to top-level URLs **urllib.request**
  - **http.server** contains basic socketserver-based HTTP server classes.
  - **http.cookies** has utilities for managing the state of cookies
  - **http.cookiejar** provides cookie persistence

Implementing a server

```
from http.server import BaseHTTPRequestHandler, HTTPServer

# HTTPRequestHandler class
class testHTTPServer_RequestHandler(BaseHTTPRequestHandler):

  # GET
  def do_GET(self):
    # Send response status code
    selfsend_response(200)

    # Send headers
    self.send_header('Content-type', textihtm1')
    seltend_headers0
```

Implementing a server

```
    # Send message back to client
    message = "Helio world!"
    # Write content as utf-8 data
    seltwfile.write(bytes(message, "utf8"))
    return
def run():
  print('starting server…')

  # Server settings
  # Choose port 8080, for port 80, which is normally used for a http server, you need root access
  server_address = ('127.0.0.1 ', 8081)
  httpd = HWPSewer(sewer_address, testHlTP5erver_RequestHandler)
  print('running server…')
  httpd.serve_forever()

  run()
```

Implementing a server

```
# HTTPRequestHandler class
class testHTTPServer_RequestHandler(BaseHTTPRequestHandler):
 # GET
 def do_GET(self):
  # Send response status code
  self.send_response(200)

  # Send headers
  selfsend header('Content-type', ‘text/html’)
  self.end_headers()

  # Send message back to client
  message = "Helio world"
  # VVrite content as utf-8 data
  self.wfile.write(bytes(message, "utf8"))
  return
```

Using http.client

```
import http.client

conn = http.client.HTTPConnection("localhost", 8081)
conn.request("GET", "/")

r1 = conn.getresponse()
print(r1.status,r1.reason)
datal = r1.read()
print(data1)
```

Using urilib

- urIlib.request is a Python module for accessing URLs (Uniform Resource Locators).
- It offers a very simple interface, in the urlopen function, capable of obtaining URLs using a variety of different protocols.
- It also offers a slightly more complex interface to handle common situations - like basic authentication, cookies, proxies and so on. These are provided by handlers and openers objects.

  https://docs.python.org/3/howto/urllib2.html

Using urilib

- Basic reading

```
import urllib.request
with urllib.request.urlopen('http://python.org/') as response:
html = response.read()
```

- Storing in a file

```
import urllib.request
local_filename, headers = urllib.
request.urlretrieve('http://python.org/')
html = open(local_filename)
```

Using urilib - Using headers and data (POST)

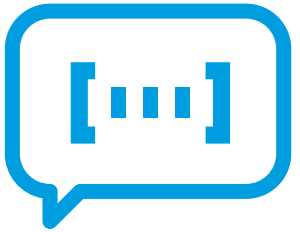
```
import urilib.parse
import urllib.request

url = 'http://www.someserver.com/cgi-bin/register.cgi'
user_agent = 'Mozilla/5.0 (Windows NT 6.1; Win64; x64)'
values = {'name':'Michael Foord',
 'location': 'Northampton',
 'language': 'Python' }
headers = {'User-Agent': user_agent}

data = urilib.parse.urlencode(values)
data = data.encode('ascii')
req = urllib.request.Request(url, data, headers)
with urilib.requesturlopen(req) as response:
    the page = response.read()
```

Read the headlines

- Use python for reading the New York Times headlines

netmind
a bts company

26



Accessing a Web Service

Querying a WebService

- We can use the requests library
  - https://realpython.com/python-requests/

- requests facilitates access to REST-type web services, including authentication or authentication
- If the module is not available, it can be installed using pip:

```
pip install requests
```

Querying a WebService

```
import requests


r = requests.get('https://api.github.com/user', auth=(username,
password))
r.status_code
r.headers['content-type']
r.encoding
r.text
print(r.json())
```

JSON in python

- The python json library allows to encode and decode json
  - https://docs.python.org/3/library/json.html
  - json.dump(obj, fp, *, skipkeys=False, ensure_ascii=True, check_circular=True, allow_nan=True, cls=None, indent=None, separators=None, default=None, sort_keys=False, **kw)
    - Serialize an object as JSON
  - json.dumps — serialize an object to a string Formatted JSON
  - json.load(fp,*, cls=None, object_hook=None, parse_float=None, parse_int=None, parse_constant=None, object_pairs_hook=None,**kw)
    - Deserialize an fp to a Python object

- All of these methods use the conversion table defined in:

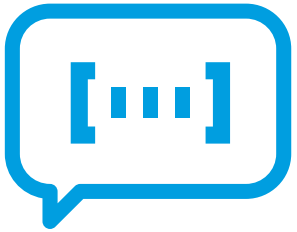
  https://docs.python.org/3/library/json.html#json-to-py-table

Consuming a Rest API

- Access and consume the API https://timeapi.io/swagger/index.html

27

Mail



Mailing

- Python provides the smtplib module, which defines an SMTP client session object that can be used to send mail to any machine on the Internet with an SMTP or ESMTP listening daemon.
  - https://docs.python.org/3/library/smtplib.html

- The syntax to use is as follows

```
import smtplib
smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]]
```

  - **host**: the SMTP host (IP or domain.) This is optional.
  - **port**: if host is provided, you need to specify the port where the SMTP server is listening. Usually port 25.
  - **local_hostname**: If the SMTP server is running on the local machine, specify localhost.

Mailing

- An SMTP object has a method called sendmail, which is normally used to send a message.
- Three parameters are needed:
  - **The sender** - A string with the sender's address.
  - **Recipients** - A list of strings, one for each recipient.
  - **The message**: a message as a string formatted as specified in the various RFCs.
- By default the message will be sent as text, but if they find html tags, it will be sent as html

Mailing

```
import smtplib


sender = Trom@fromdomain.comi
receivers = rtotodomain.coml
message = """From: From Person <fromefromdomain.com>
To: To Person <to@todomain.com>
Subject: SMTP e-mail test
This is a test e-mail message.
"""
try:
 smtpObj = smtplib.SMTPClocalhostT)
 smtpObj.sendmail(sender, receivers, message)
 print "Successfully sent email"
except SMTPException:
 print "Error: unable to send email"
```

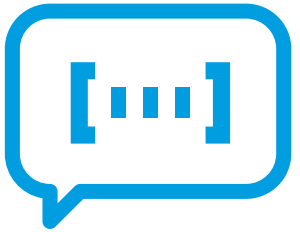
Using GMail

- Send an email using your gmail account
- You must activate the less secure option of your account at https://www.google.com/settings/security/lesssecureapps

- Use port 587
- Activate the ehlo option and starttls
- Use the login method to indicate your username/password

28

FTP



FTP

- The **ftplib** library implements the FTP protocol.
  - https://docs.python.org/3/library/ftplib.html

- Using FTP we can create and access remote files through function calls using the syntax:

```
from ftplib import FTP

ftplib.FTP(host='', user='', passwd='', acct='', timeout=None,
source_address=None)
```

- There is also the possibility to use TLS with the FTP_TLS subclass:

```
ftplib.FTP_TLS(host='', user='', passwd='', acct='', keyfile=None, certfile
=None, context=None, timeout=None, source_address=None)
```

List a directory

```
import ftplib

ftp = ftplib.FTP("ftp.nluug.n1")
ftp.login("anonymous", "ftplib-example-1")

data = []

ftp.dir(data.append)

ftp.quit()

for line in data:
 print("-", line)
```

Change and list a directory: ftp.cwd('/')

```
import ftplib

ftp = ftplib.FTP("ftp.nluug.n1")
ftp.login("anonymous", "ftplib-example-1")

data = []

ftp.cwd('/pub') # change directory to /pub/
ftp.dir(data.append)

ftp.quit()

for line in data:
 print("-", line)
```

Download a file

```
import ftplib

def getFile(ftp, filename):
 try:
   ftp.retrbinary("RETR " + filename, open(filename, 'wb').write)
 except:
   print(" Error")

ftp = ftplib.FTP("ftp.nluug.n1")
ftp.login("anonymous", "ftplib-example-1")

ftp.cwd('/pub') # change directory to /pub/
getFile(ftp, 'README.nluug')

ftp.quit()
```

Upload a file

```
import ftplib
import os


def upload(ftp, file):
 ext = os.path.splitext(file)[1]
 if ext in (".txt", ".htm", ".html"):
   ftp.storlines("STOR " + file, open(file))
 else:
   ftp.storbinary("STOR " + file, open(file, "rb"), 1024)


ftp = ftplib.FTP("127.0.0.1")
ftp.login("username", "password")

upload(ftp, "README.nluug")
```

SFTP

- To deal with sftp there is a module that solves most of the casuistry: pysftp
  https://pypi.org/project/pysftp/
- To use it it is necessary to install it using pip

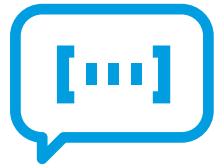
```
pip install pysftp
```

```
import pysftp

with pysftp.Connection('hostname', username=imel, password=isecreti) as sftp:
  with sftp.cd('public'):  # temporarily chdir to public
       sftp.put(‘my/local/filename') # upload file to public/ on remote
       sftp.get('remote_file') # get a remote file
```

Playing with ftp

- Generate a script that connects to an ftp server, shows the root list and from there allows you to change directories, download a file (or several) or upload one (or several)
- The files can be text or binary

netmind
a bts company

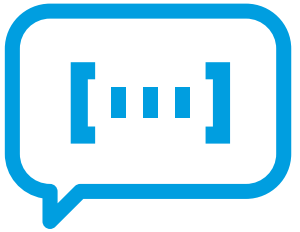


Web Development

29 Flask

netmind
a bts company

29



Flask

Flask

- Flask is a **microframework** developed by Armin Ronacher that allows you to create web applications in the blink of an eye, all with an absurdly small number of lines of code.
  - http://flask.pocoo.org/

- Flask, unlike Django and Pyramid, does not come with hundreds of modules to tackle common development tasks in web development. Rather it focuses on providing the bare minimum so that you can get a basic application up and running in a matter of minutes.
- It is perfect, for example, for rapid prototyping of projects.

EXERCISE

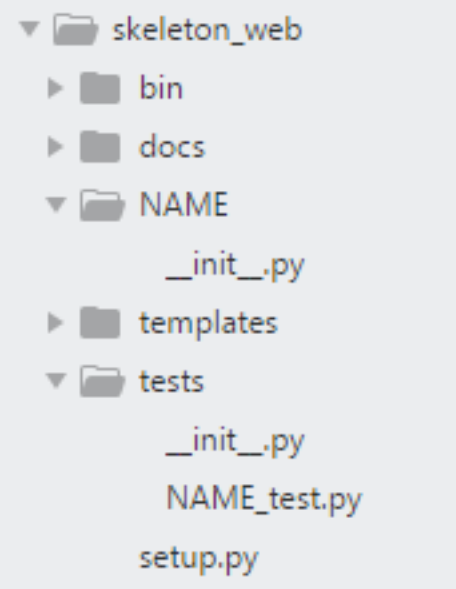
My First Flask App

Install Flask

```
pip install flask
```

Create a web project structure

- Create an skeleton like this one →



Create a server script:

```
from flask import Flask
app = Flask(__name__)


@app.route("/")
def hello():
  return "Helio World!"
```

Launch the server:

```
SET FLASK_APP=app.py
flask run
```

- Or just: py app.py

Access to the app in your browser:
http://localhost:5000/

Finish the app
- Press *CTRL+C*

- We have imported the Flask class
- Next we have created an app instance with the __main__ argument necessary for Flask to search for files, templates and other files.
- Next we have indicated that the route / (using the app.route annotation) is going to contain a function called hello that will return the message we have written.
- Finally we are going to execute the application in the host that we have assigned, creating a web server instantly.

Flask

- app.route accepts two parameters: **route and methods**

```
app.route('/hello', methods=["POST"])
```

- Likewise except path params:

```
@app.route('/<username>')
def show_user(username):
  pass


 @app.route('/post/<int:post_id>')
 def show_post(post_id):
  pass
```

Flask - templates

- We will create the templates in the templates directory

  templates/index.html

```
<html>
<head>
 <title>Gothons Of Planet Percal #25</title>
</head>
<body>
{% if greeting %}
        I just wanted to say <em style="color:
green; font-size: 2em;">{{ greeting }}</em>.
{% else %}
        <em>Hello</em>, world!
{% endif %}
</body>
</html>
```

- You can insert python code using the syntax **{%<python>%}**
- And insert values using mustache code **{{<value>}}**

Flask - templates

- We will relate the template from the code with **render_template**

```
from flask import Flask
from flask import render_template

app = Flask(__name__)

@app.route('/')
def index():
    greeting = 'Helio World'
    return render_template("index.html", greeting=greeting)


if __name__ == "__main__"
    app.run()
```

Flask - inputs

- We can collect the query params when we have a GET request with:

```
request.args.get('<param_name>,'<default>'')
```

- We can also use forms to send GET or POST requests and collect the parameters with:

```
request.form('<field_form>')
```

Flask - input templates

- **templates/hello_form.html:**

```
<hl >Fill Out This Form</h1>

<form method="POST" action="/hello">
<div class="pure-form">
 <fieldset>
 <Iegend> A Greeting App</legend>
 <input type="text" placeholder="greeting word" name="greet" />
 <input type="text" placeholder="your name" name="name" />
 <input class ="pure-button pure-button-primary" type="submit"f>
 </fieldset>
</div>
</form>
```

Flask Layouts

- You can create template skeletons for common elements and reuse them in your other templates
  - **templates/layout.html:**

```
<html>
<head>
 <title>Gothons From Planet Percal #25</title>
</head>
<body>
  {% block content %}
  {% endblock %}
</body>
</html>
```

Flask Layouts

- In the secondary template, the layout is incorporated using {% extends "<base template>"%}
  - **templates/index. html:**

```
{% extends "layout.html" %}

{% block content %}
    {% if greeting %}
        I just wanted to say
        <em style="color: green; font-size: 2em;">{{ greeting }}</em>.
    {% else %}
        <em>Hello</em>, world!
    {% endif %}

{% endblock %}
```

Flask - Automated tests

- You can use nose to implement automatic tests that simulate requests and form submissions (and much more)
- To do this, you must import the application and **set the application's testing property to True**.

Flask - Automated tests

- **tests/app test.py**

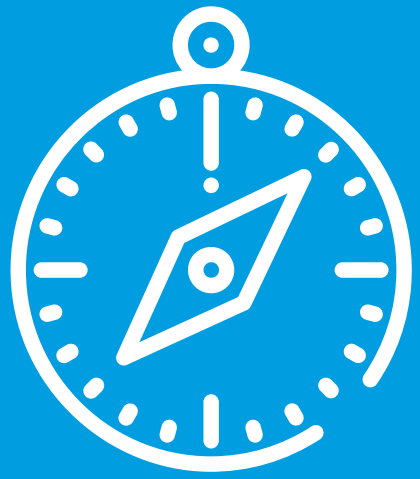
```
from nose.tools import*
from app_forms_ly import app

app.testing = True
web = app.test_client()


def test_index():
 result = web.get('/', follow_redirects=True)
 assert_equal(result.status_code, 200)
 assert_in(b"FiII Out This Farm", result.data)


def test_hello():
 result = web.get('/hello', follow_redirects=True)
 assert_equal(result.status_code, 200)
 assert_in(b"Hello, Nobody", result.data)

data = {'name': 'Ricardo', 'greet': 'Hey'}
result = web.post('/hello',
follow_redirects=True,data=data)
assert_in(b"Ricardo", result.data)
assert_in(b"Hey!", result.data)
```

Implementing a REST API

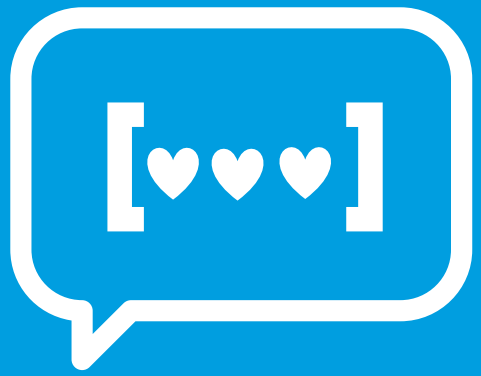
- Create a REST api that returns a user's data based on the path_param uid for the path:
  /users/<uid>

- You will need to use flask jsonify

```
from flask import Flask, jsonify
...
return jsonify({'data':data})
```



Next steps



We would like to know your opinion!

Please, let us know what you think about the content.
From Netmind we want to say thank you, we appreciate time and effort you have taking in answering all of that is important in order to improve our training plans so that you will always be satisfied with having chosen us
quality@netmind.es

netmind
a bts company

Thanks!

Follow us:

netmind
a bts company

EMPOWERING DIGITAL TEAMS

netmind.net

Barcelona

C. dels Almogàvers 123
08018 Barcelona
Tel. +34 933 041 720
Fax. +34 933 041 722

Madrid

C. Bambú, 8
28036 Madrid
Tel. +34 914 427 703
Fax +34 914 427 707

Atlanta

3372 Peachtree Road NE
Suite 115
T. +1 (678) 366 1393
Atlanta, GA 30326