# Robotics project 2024

## Python3 for Robotics

Joaquin Rodriguez

# TABLE OF CONTENTS



# Introduction

- Sources:
  - **TheConstructSim - https://bit.ly/3iPlXzi**
  - **The Python3 documentation: https://bit.ly/3rC1nYd**
- Amount of hours recommended: 11h30
- Topics to master:

**Variables**
**Data manipulation**
**Conditionals and loops**
**Functions**
**Classes**

- Development platform: TheConstructSim - Gazebo

# Why Python ?

- PROS:
  - **From all the available programming languages, Python is the easiest to learn.**
  - **Many libraries availables: OpenAI, OpenCV, ROS, TensorFlow …**
  - **Well documented, internet examples.**
  - **We can treat single values and arrays in the same way.**
  - **No compilation - Cross-platform (depends on interpreter)**

# Why Python ?

- BUT…
  - It is not the fastest way to run your program
  - **Prune to hidden errors**

# Why Python ?

***"It is really important to learn how to program properly, not only in Python, but in any programming language that we choose"***

```python
def get_sign(num):
    return num / abs(num)

def division_by_definition(dividend, divisor):
    result = 0

    sign = get_sign(dividend) * get_sign(divisor)
    abs_divisor = abs(divisor)
    abs_dividend = abs(dividend)
    while (result * abs_divisor) < abs_dividend:
        result = result + 1

    if (result * abs_divisor) > abs_dividend:
        result = result - 1

    return int(result * sign)
```

```python
def division_by_implementation(dividend, divisor):
    return int(dividend / divisor)
```

# Getting started

- The very basic step: ***file with extension .py***
- Any text editor is useful, but programming oriented is better (Spyder, Notepad++, Sublime, VSCode, CLion, Vim).
- How to run ?

***python3 /path/to/file/my_script.py***

***python3 ./my_script.py***

- Recommendation: ***install ipython3*** for an interactive python terminal.

# Step 1: Variables and modules

- A variable is a container - a piece of memory
- It always has a name, a type and a content.
- Python is a soft-typed language → Variables can change their type during execution.
- A variable is created after its first assignment (myVar = 5)

# Step 1: Variables and modules

- Some basics variables types are included by default in Python:
  - **Integers**
  - **Floating point:**
  - **Boolean:**
  - **Strings**
  - **Lists:**
  - **List of Lists:**
  - **Tuples:**
  - **Dictionaries:**

```
a = 2
```

```
b = 2.0
```

```
b = True; c = False
```

```
s = 'This is an string'
```

```
c = [2, 5, 3, 1, 7, 8]; d = [9, 'hello', 'pie', 5.0, 'apple']
```

```
k = [ [3,2,1], [1,1,1], [6,7,8]]
```

```
e = (2, 6, 7); f = (2, 'hello', 10)
```

```
myDict = {}
myDict['apples'] = 10
myDict['oranges'] = 5
```

# Step 1: Variables and modules

- We can visualize the content of a variable with the ***print*** function:

```
print(a)
print('The value of the variable b is {}'.format(b))
print("Using double quotes: {}".format(c))
```

- Some details:
  - **Numeric variables: addition ( + ), subtraction ( - ), division ( / ), multiplication ( * ), power ( ** ). Then we have specific operations: %, AND, OR, bit-wise OR ( | ), bitwise AND ( & ).**
  - **Lists: Read / write. They can store a set of values of different types. We can add elements with the method *append*. We access individual elements with the [ ] and the element index.**

# Step 1: Variables and modules

- We can visualize the content of a variable with the print function:

```
print(a)
print('The value of the variable b is {}'.format(b))
print("Using double quotes: {}".format(c))
```

- Some details:
  - **Tuples: Read-only. They can store a set of values of different types. We can access to individual elements with [ ]. We cannot change their content after initialization.**
  - **Dictionaries: Read / Write. Each entry has the shape Key - Value. We can add elements at any moment. Both, key and value, can be of any type.**

# Step 1: Variables and modules

- We can add new data types and functionalities through ***Modules***.
- We add a module with the command ***import***.

```
import numpy
import matplotlib
import cv2
import math
import rospy
import shapely
```

- An import can be done at any part of the code (top of the file, inside a function, just before using it, etc).
- For better organization, we place them at the beginning of the Python script.

# Step 1: Variables and modules

- Each time we use a module, we start the sentence with the module name:

```
import numpy
myArray = numpy.array([2, 6, 4, 8], dtype=np.uint8)
```

- Module with long name ? We use ***as***

```
import numpy as np
myArray = np.array([2, 6, 4, 8], dtype=np.uint8)
```

- If we want to import a sub-module of a module we use the ***. (dot)***:

```
import matplotlib.pyplot as plt
plt.figure()
```

- If we want to import a single function, or an object, we use the syntax ***from … import …***

```
from shapely.geometry import Polygon
from rospy import Time
```

# Step 1: Variables and modules

- There is a particular data type in Python: the type ***None***
- It is used to indicate the variable has no type.
- It is mostly used to check if the result of an operation or a function is valid or not, or detect when a function failed in do its task.

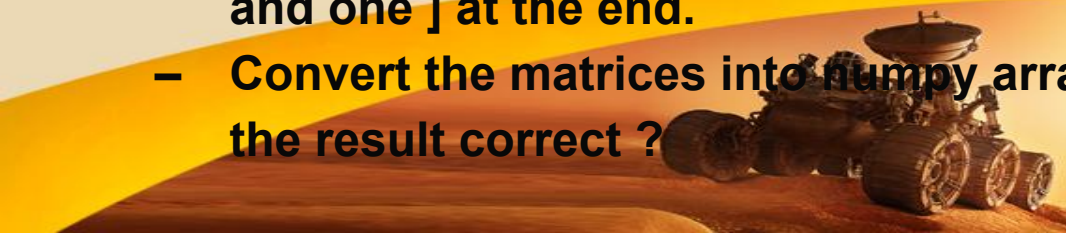
# Step 1: Variables and modules

- With a huge amount of modules available out there, we have an infinite amount of variable types too.

- How do you know which modules to import / use ?
  - **It depends on your application. You need to know which modules are useful for what you want to do → Refer to the module documentation (Google, Bing, Duckduckgo, etc)**

# Exercise 1

- Create a Python script called ***my_first_script.py***
- Practise different operations we have seen:
  - **Import the module numpy and rename it as *np***
  - **Create two numeric variables with the values 51 and 35, and compute the results of +, -, /, *, |, & and %. Check the results with the print function.**
  - **Create TWO 3 x 3 matrices with random numbers using lists (also called list of lists) and add them using the + operator. HELP: a single list will be a row of the matrix. You separate the rows with ‘, ’ (comma). The comma separated lists have to be enclosed by one [ at the beginning and one ] at the end.**
  - **Convert the matrices into numpy arrays using NumPy, and add them. Is the result correct ?**



# Step 2: Data manipulation

- In lists, we can access to one or several elements with the [ ] operator:

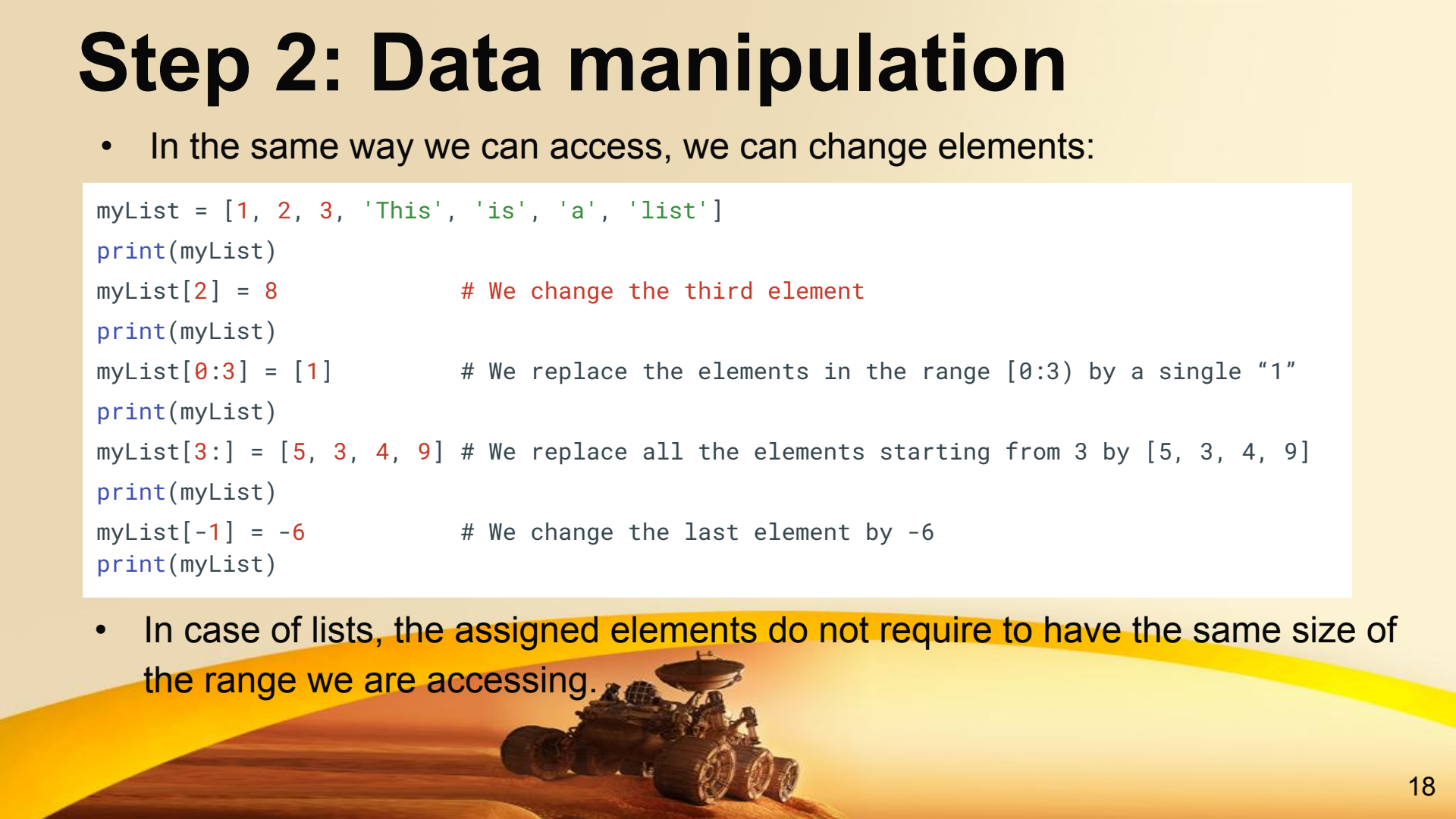
```
myList = [1, 2, 3, 'This', 'is', 'a', 'list']
print (myList)

myList[2] = 8
print (myList)

print(myList[2])          # We access to the third element
print(myList[0:3])        # We access to the elements in the range [0, 3)
print(myList[3:])         # We access to the elements, starting from the 4th element
print(myList[:3])         # We access to the elements in the range [0:3)
print(myList[-1])         # We access to the last element only
```

# Step 2: Data manipulation

- In the same way we can access, we can change elements:

```
myList = [1, 2, 3, 'This', 'is', 'a', 'list']
print(myList)
myList[2] = 8              # We change the third element
print(myList)
myList[0:3] = [1]          # We replace the elements in the range [0:3) by a single “1”
print(myList)
myList[3:] = [5, 3, 4, 9] # We replace all the elements starting from 3 by [5, 3, 4, 9]
print(myList)
myList[-1] = -6            # We change the last element by -6
print(myList)
```

- In case of lists, the assigned elements do not require to have the same size of the range we are accessing.



# Step 2: Data manipulation

- Example of list manipulation:

```
# We define a list of numbers called my_list
my_list = [5, 10, 6, 7, 15, 21, 35, 1, -2, -8]
# We show the variable content with print
print(my_list)
# We change the third element of the list (index 2)
my_list[2] = 55
# We show the modified list
print(my_list)
# We add the fourth and fifth elements, and we print the result
sum_result = my_list[3] + my_list[4]
print(sum_result)
```

# Step 2: Data manipulation

- Always be sure of the type of variable you are handling, and which operation you want to do with the data.
- Example: Addition of lists. What do you expect it will happen if we do this ?

```
# We define a list of numbers called my_list
my_list = [5, 10, 6, 7, 15, 21, 35, 1, -2, -8]
# We define a second list of numbers called my_new_list
my_new_list = [1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
# We add them
my_list + my_new_list
```

- Result:

```
[5, 10, 6, 7, 15, 21, 35, 1, -2, -8, 1, 2, 3, 4, 5, 6, 7, 8, 9, 0]
```

# Step 2: Data manipulation

- For lists, the **+ *operator*** is the ***concatenation*** of lists.
- If we want the addition of two lists, we have to define them as numpy arrays, and they must have the same amount of elements.

```
import numpy as np
# We define a list of numbers called my_list
my_list = np.array([5, 10, 6, 7, 15, 21, 35, 1, -2, -8])
# We define a second list of numbers called my_new_list
my_new_list = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 0])
# We add them
my_list + my_new_list

→ array([ 6, 12,  9, 11, 20, 27, 42,  9,  7, -8])
```

# Step 2: Data manipulation

- If you do not know what an operation does, ***the interpreter is your friend!***
- Go to a terminal, type ***python3,*** and then write a piece of code to test, and see the results to understand what your operation is doing:

```
ter) ✗ python3
Python 3.6.9 (default, Oct  8 2020, 12:12:24)
[GCC 8.4.0] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> list_1 = [2, 5, 7, 8, 9]
>>> list_2 = [2,2,2,1,3,5,7,9]
>>> list_1 + list_2
[2, 5, 7, 8, 9, 2, 2, 2, 1, 3, 5, 7, 9]
>>>
```

- To exit, just type: ***quit()***.

# Step 2: Data manipulation

- String is a data type with several functions included.
- The most useful ones are:
  - **Place a variable value in the string (no matter the type)**

```
‘My variable contains the following value: {}’.format(myVar)
```

  - **Split string**

```
'my string is going to be split'.split(" ")
```

  - **Concatenate strings (as we did for lists)**

```
str1 = 'Hello world'
str2 = ' is what I said'
str1 + str2
```

# Step 2: Data manipulation

- Strings can be considered as a list of characters.
- We can access to a particular character with the [ ].

```
myStr = 'I am writing a string'
print(myStr)
print(myStr[3])
```

# Step 2: Data manipulation

- Dictionary is an useful tool in Python.
- We access to the elements through a key, not a position.
- In python2, the values are not ordered → If we run the program twice, and we query the keys, it is not ensured the elements are in the same position each time.
- In Python 3.6+, the elements keep insertion order.
- Only existing keys can be queried.

```
myDict = {"Jon": 25, "Daenerys": 22, "Cersei": 31, "Night King": 35}
print(myDict['Jon'])      # It returns 25
print(myDict['Ken'])      # It throws an KeyError exception
```

# Step 2: Data manipulation

- **To avoid this problem, we can use the function *get.***

```
myDict = {"Jon": 25, "Daenerys": 22, "Cersei": 31, "Night King": 35}
print(myDict.get('Jon'))      # It returns 25
print(myDict.get('Ken'))      # It returns None. When None is returned, the print function
                              # does nothing
```

- It is recommended to use the get function to avoid the exceptions in the code.

# Step 2: Data manipulation

- When writing a dictionary:
  - **If the key exists, when we assign something to it, it will update the value at that key.**
  - **If the key does not exists, a new entry is created, and the value is assigned.**

```
myDict = {"Jon": 25, "Daenerys": 22, "Cersei": 31, "Night King": 35}
myDict['Jon'] = 32
myDict['Ken'] = 10
print(myDict)
```

# Step 2: Data manipulation

- Operators
  - **Assignment: The final value is assigned to the left side variable**

| Operator | Example | Same As |
|---|---|---|
| = | x = 5 | x = 5 |
| += | x += 3 | x = x + 3 |
| -= | x -= 3 | x = x - 3 |
| *= | x *= 3 | x = x * 3 |
| /= | x /= 3 | x = x / 3 |
| %= | x %= 3 | x = x % 3 |

Source: Python3 for Robotics: https://app.theconstructsim.com/#/Desktop

# Step 2: Data manipulation

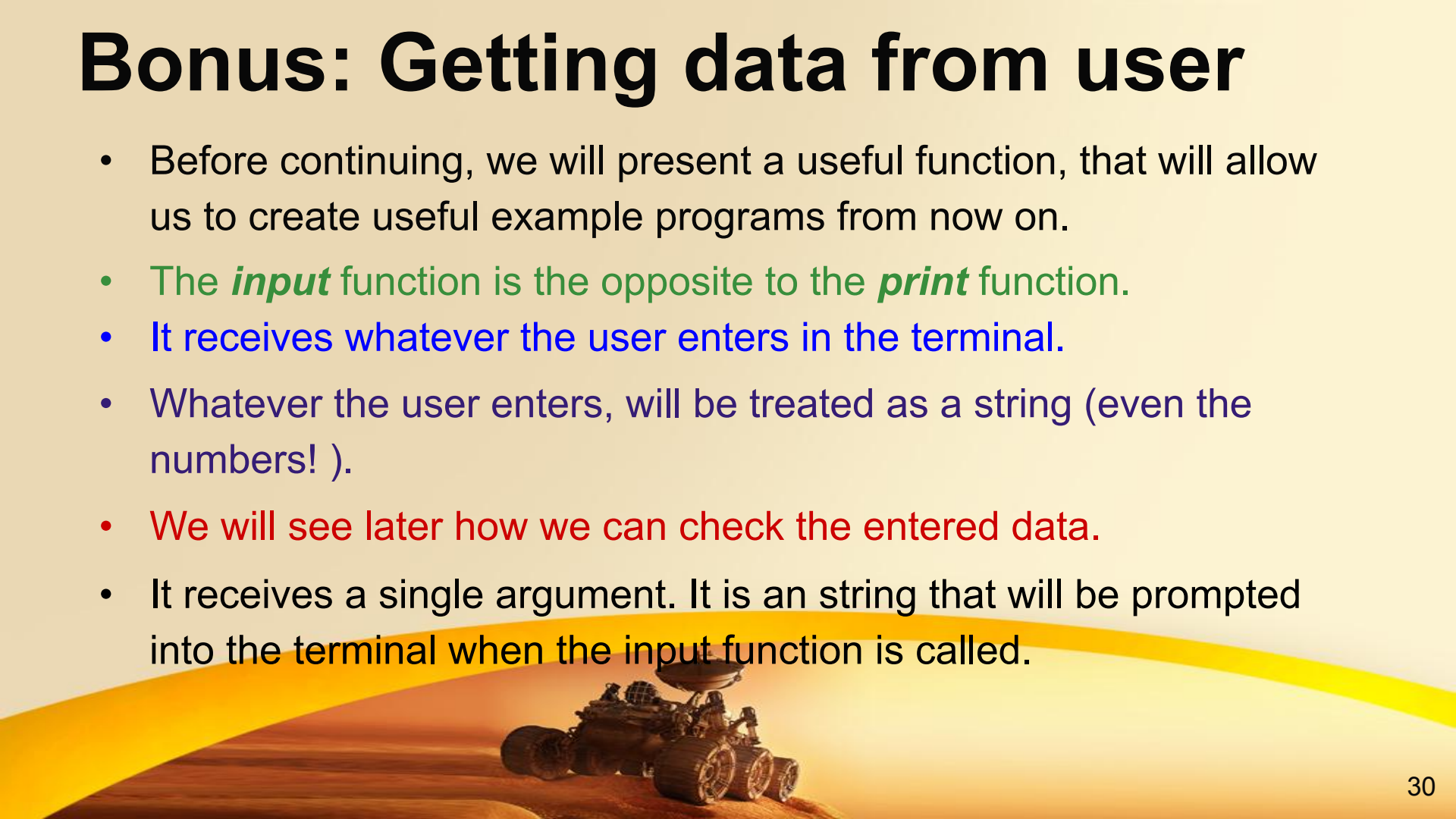
- Operators
  - **Comparison: They return a boolean value, i.e., either True or False**

| Operator | Means | Same As |
|---|---|---|
| == | Equal | 5 == 5 |
| != | Not Equal | 4 != 5 |
| > | Greater than | 5 > 4 |
| < | Less than | 4 < 5 |
| >= | Greater than or equal to | 5 >= 4 |
| <= | Less than or equal to | 4 <= 5 |

Source: Python3 for Robotics: https://app.theconstructsim.com/#/Desktop

# Bonus: Getting data from user

- Before continuing, we will present a useful function, that will allow us to create useful example programs from now on.
- The ***input*** function is the opposite to the ***print*** function.
- It receives whatever the user enters in the terminal.
- Whatever the user enters, will be treated as a string (even the numbers! ).
- We will see later how we can check the entered data.
- It receives a single argument. It is an string that will be prompted into the terminal when the input function is called.



# Bonus: Getting data from user

- Example:

```
# This line will not show anything when executed
userIn = input()
print(“The user entered: {}”.format(userIn))

# This line prints the string when input is called.
userInWithPrompt = input("Please enter a number between 0 and 10")
print(“The user entered: {}”.format(userInWithPrompt))
```

# Bonus: Getting data from user

- The input function blocks the execution indefinitely until the user press the key “Enter” from the keyboard (End of line).
- It is recommended to always include the message argument, so the user knows what type of data the program expects.
- Always check if the data entered is correct, to avoid unexpected crashes later on in the program.
- If the input has to be an integer or a float, we can cast the string by using ***int(myString)*** and ***float(myString)***, respectively.

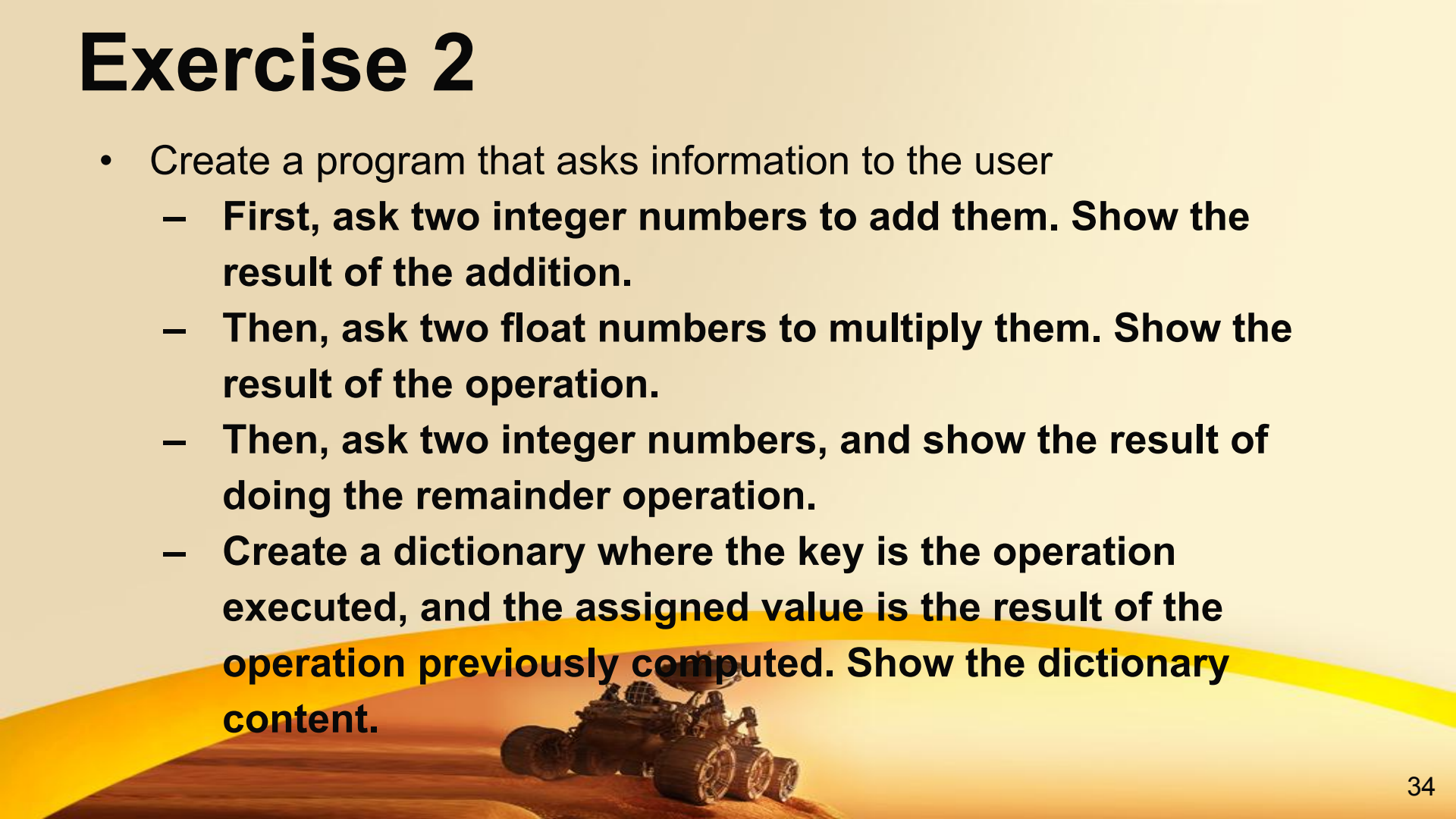
# Bonus: Getting data from user

- Example:

```
myString = input("Please, enter an integer number:")
myInt = int(myString)
myString = input("Please, enter a float number:")
myFloat = float(myString)
print("The integer entered is: {}”.format(myInt))
print("The float number entered is: {}".format(myFloat))
```

# Exercise 2

- Create a program that asks information to the user
  - **First, ask two integer numbers to add them. Show the result of the addition.**
  - **Then, ask two float numbers to multiply them. Show the result of the operation.**
  - **Then, ask two integer numbers, and show the result of doing the remainder operation.**
  - **Create a dictionary where the key is the operation executed, and the assigned value is the result of the operation previously computed. Show the dictionary content.**



# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **They evaluate a boolean condition, and based on the answer, the program executes (or not) a *block of code.***
  - **A block of code is composed by all the lines with a greater indentation level than the if line**
  - **The different indentation levels are achieved through spaces placed at the beginning of the line:**

```
|Ind. level 0
|    Ind. level 1
|        Ind. level 2
|            Ind. level 3
|            ...
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **As soon as the indentation level is reduced, the block of code is considered as finished**
- Syntax:

```
if condition_1:
    ## Execute this piece of code
elif condition_2:
    ## Execute this other code
elif condition_3:
    ...
else:
    # If none of the previous statements are true,
    # execute this code
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - *condition_i* **are boolean expressions (Simple or compound)**
  - **Only the first** *if* **is mandatory.** *elif* **and** *else* **are optional**
  - **If the condition_1 is true, the first block of code executed, and the rest is not executed.**

```
if condition_1:
    ## Execute this piece of code
elif condition_2:
    ## Execute this other code
elif condition_3:
    ...
else:
    # If none of the previous statements are true,
    # execute this code
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **If the *condition_1* is False, and the *condition_2* is True, the second block is executed, and so on.**
  - **If none of the *condition_i* are true, the else block is executed**

```
if condition_1:
    ## Execute this piece of code
elif condition_2:
    ## Execute this other code
elif condition_3:
    ...
else:
    # If none of the previous statements are true,
    # execute this code
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **Example:**

```
number = 20
if number > 50:
    print("The number is greater than 50")
elif number > 30:
    print("The number is greater than 30 and less than 50")
else:
    print("The number is less than 30")

import cv2
img = cv2.imread('/path/to/my/image.png', cv2.IMREAD_GRAYSCALE)
if img is None:
    exit -1
# DO IMAGE PROCESSING!
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **Each built-in datatype has a particular value that can be considered as False, and all the other cases are considered as True**
  - **Integers and float: 0 is false, any other number is true**
  - **Tuple, Lists, dictionaries, string: if it is empty is false, any other case is true.**
  - **"*None*" is always false**
  - **More complex datatypes might include a function called empty, or isEmpty, to check this situation.**

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **Examples:**

```
number = 0
myString = ''
if number:              # It is equivalent to “if number != 0”
    print("The number is non-zero")
else:
    print("The number is zero")

if myString:
    print("The string is not empty")
else:
    print("The string is empty")
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **Using the logic operators, we can create complex statements to evaluate:**

```
number = 0
if not number:
    print("The number is zero")
else:
    print("The number is not zero")
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **Using the logic operators, we can create complex statements to evaluate:**

```
vector = [5.5, 2.1, 9]
if (vector) and (len(vector) == 3 ) and (vector[2] != 0):
    vector[0] = vector[0] / vector[2]
    vector[1] = vector[1] / vector[2]
    vector[2] = 1
else:
    print("ERROR: We cannot compute the homogeneous coordinates. Z coordinate is 0")
print("Resulting vector: {}".format(vector))
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **There are two operations that are frequently used with flow control: IS and IN.**
  - **IS is used to check whether both the operands *refer* to the same object or not.**
  - **Example:**

```
list1 = []
list2 = []
print(list1 is list2)  # This operation gives False
list1 = list2
print(list1 is list2)  # This operation gives True
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **The second case gives True since big data structures (like lists, vectors, or images) are not copied when assigned, but a reference is given to the assigned variable. As a consequence, they are pointing to the same memory space. This is called *Shallow copy***
  - **Example:**

```
list1 = []
list2 = []
list1 = list2   # We create a shallow copy of list2 into list1
list1.append(2) # We add an element into list1
print(list1)    # Since list1 is a shallow copy, both variables
print(list2)    # contain the appended value
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **If we want to create a real copy, to not change the original data, we need to do a deep copy**

```
import copy
list1 = []
list2 = [2,3,4,5,6]
list1 = copy.deepcopy(list2)    # We create a deep copy of list2 into list1
list2.append(2)                 # We add an element into list2
print(list1)                    # Since list1 is a deep copy, only list2 contains the new value
print(list2)                    # list1 stays untouched
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **One common use of the IS operator is to check if a variable is of type None.**

```
myDict = {"Jon": 25, "Daenerys": 22, "Cersei": 31, "Night King": 35}
ret_val = myDict.get("Jon")
if not ret_val is None:
    print("The key Jon is present in the dictionary")
else:
    print("The key Jon is NOT present in the dictionary")

ret_val = myDict.get("Vibot")
if not ret_val is None:
    print("The key Vibot is present in the dictionary")
else:
    print("The key Vibot is NOT present in the dictionary")
```

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **The IN operator serves to know if a specific element is present in a variable.**
  - **It can only be used with iterable variables, as strings, lists, tuples and dictionaries.**
  - **Any iterable object can be seen as a set of values. Then, the IN operator tell us if certain element is present in the set or not.**

# Step 3: Flow control statements

- IF / ELIF / ELSE statements
  - **Example:**

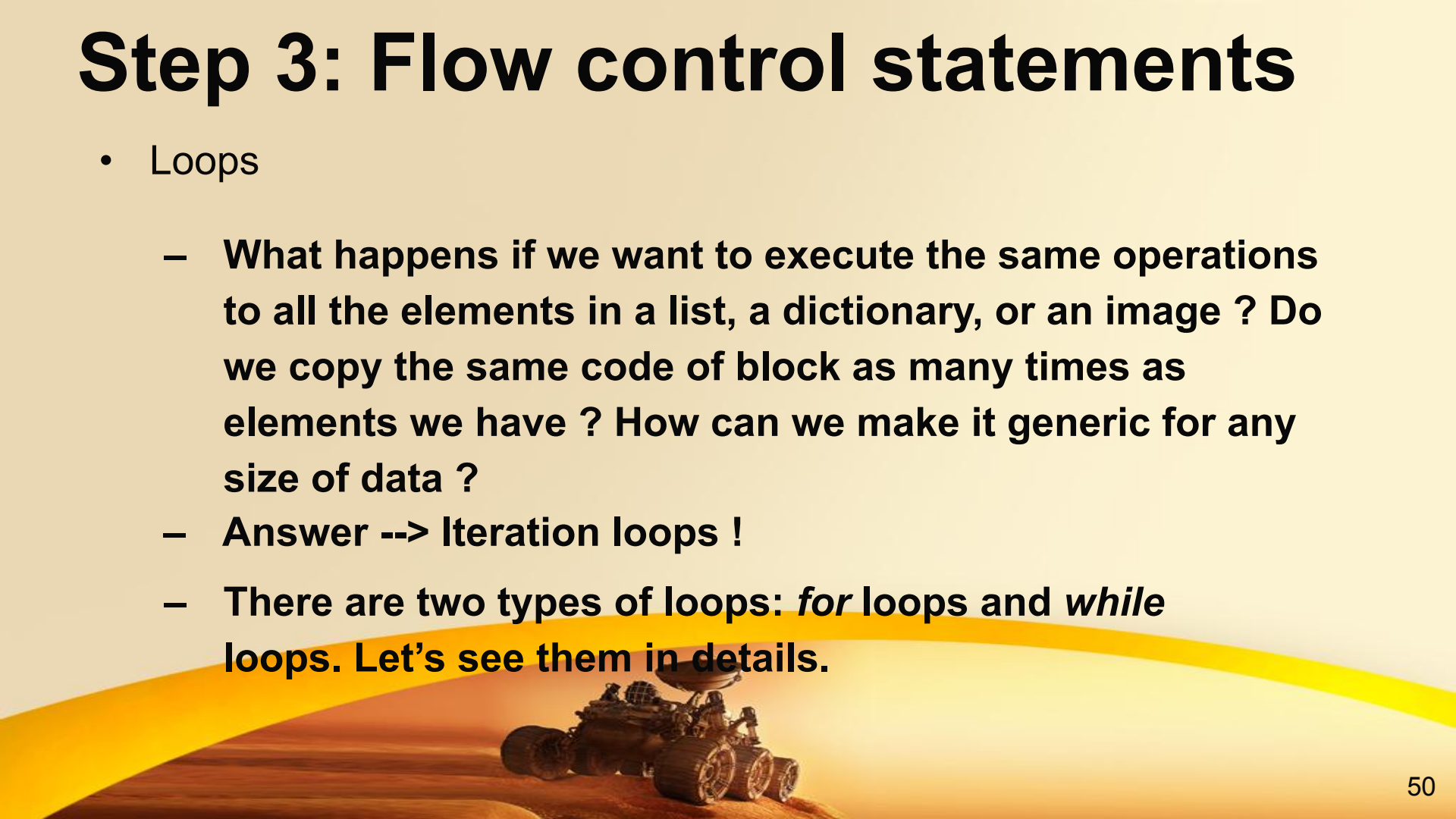
```
myString = "Today it is a sunny day"
if "sun" in myString:
    print("I found the word sun in the string")
else:
    print("I did not find the word sun in the string")
```

- For the dictionary case, the operator IN will check if the given value is within the keys, and the value given must match exactly with the key value.

```
myDict = {"Jon": 25, "Daenerys": 22, "Cersei": 31, "Night King": 35}
print("Jon" in myDict)  # This statement will give True
print("J" in myDict)    # This statement will give False
```

# Step 3: Flow control statements

- Loops
  - **What happens if we want to execute the same operations to all the elements in a list, a dictionary, or an image ? Do we copy the same code of block as many times as elements we have ? How can we make it generic for any size of data ?**
  - **Answer --> Iteration loops !**
  - **There are two types of loops: *for* loops and *while* loops. Let's see them in details.**

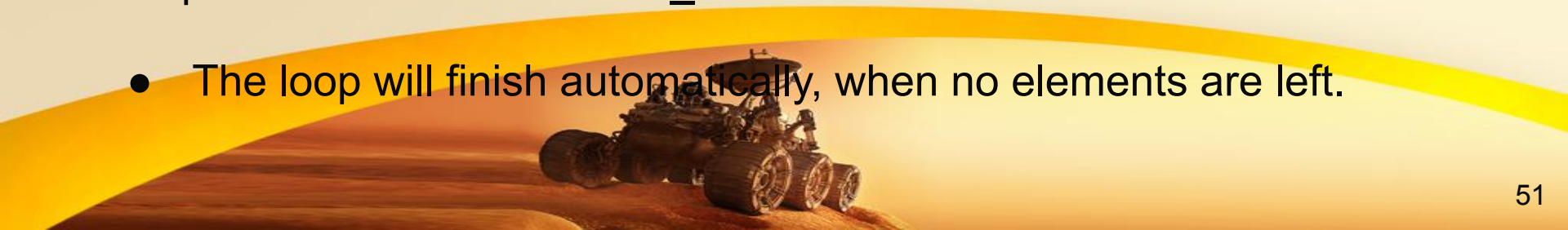


# Step 3: Flow control statements

- Loops
  - **For loops are used when we want to iterate over a fixed amount of elements.**
  - **Syntax:**

```
for var_name in set_of_elements:
    # tasks to do
```

- At each iteration, one element from ***set_of_elements*** will be placed in the variable ***var_name.***
- The loop will finish automatically, when no elements are left.



# Step 3: Flow control statements

- Loops
  - **Example:**

```
myList = [4,6,2,12,50,32,33]
#### Average of numbers
avg = 0
N = len(myList)
for val in myList:
    avg += val
avg = avg / N
print("The average of {} is {}".format(myList, avg))

#### Show element index and value
N = len(myList)
for i in range(N):
    print("The position {} contains the value {}".format(i, myList[i]))
```

# Step 3: Flow control statements

- Loops
  - **Question: What does it happen if our set of values is a variable of type String ?**
  - **In FOR loops, each iteration can define several variables. For example:**

```
myDict = {"Jon": 25, "Daenerys": 22, "Cersei": 31, "Night King": 35}
for key, val in myDict.items():
    print("The key {} holds the value {}".format(key, val))

x_coord = [6,7,2,5,1,5,6,7,8,1]
y_coord = [1,1,2,2,3,3,4,1,5,5]
for x, y in zip(x_coord, y_coord):
    print("Point coordinates: ({}, {})".format(x, y))
```

# Step 3: Flow control statements

- Loops
  - **The second type of iteration loops is the WHILE loop.**
  - **In this case, the loop will be executed while a given condition is True.**
  - **BE CAREFUL with infinite loop, unless it is your intention.**
  - **Syntax of a while loop:**

```
while condition:
    # Code
    # Update condition
```

# Step 3: Flow control statements

- Loops
  - **At each iteration, the condition will be evaluated. If at any moment its result is False, the loop finishes, and the program continues.**
  - **This is the reason why the variables involved in the condition has to be updated at each iteration, to ensure the loop will finish at some point.**

# Step 3: Flow control statements

- Loops
  - **Example: Loop that will be executed 10 times**

```
counter = 0
N = 10
while counter < N:
    counter = counter + 1
    print("This loop has been executed {} times".format(counter))
```

  - **Example: Infinite loop (disaster !!)**

```
counter = 0
is_running = True
while is_running == True:
    counter = counter + 1
    print("This loop has been executed {} times".format(counter))
```

# Step 3: Flow control statements

- Loops
  - **Loops equivalent code**

```
myList = [5,2,6,7,2,2]
for elem in myList:
    # Run operations for each list's element
```

```
myList = [5,2,6,7,2,2]
N = len(myList)
i = 0
while i < N:
    elem = myList[i]
    # Run operations for each list's element
    i += 1      ## Condition update
```

# Step 3: Flow control statements

- Loops
  - **In any loop statement, we can omit a case, or we can end the iterations abruptly, if an exceptional condition happens.**
  - **Example: We make a program to capture images continuously from a camera. The camera always give us an image EXCEPT IF it gets disconnected. If that's the case, the camera function returns None.**
  - **If the camera gets disconnected, we want to stop the program.**

# Step 3: Flow control statements

- Loops
  - **Two possible solutions:**

**1. While loop and check the image in the condition**

```
# We import the required modules, and we initialize the
camera.
# The way to get a new image is by calling the function
# get_image()
img = get_image()
while not image is None:
    # Do something with the image
    img = get_image()          ## Update condition
```

# Step 3: Flow control statements

- Loops

  **2. Using break**

```
# We import the required modules, and we initialize the camera.
# The way to get a new image is by calling the function
# get_image()
# NOTE: Infinite loop!
while True:
    img = get_image()
    if img is None:
        break
    # Do something with the image
```

# Step 3: Flow control statements

- Loops:

```
N = 0
while N < 5:
    for i in range(10):
        break
        print("I passed the break!")
    N += 1
```

  - **“break” will stop the execution of first loop to which the block of code belongs to.**
  - **If we have a loop inside another loop, only the inner one will be stopped**

# Step 3: Flow control statements

- Loops:
  - **As in the case of “break”, we can change the flow of a loop by skipping an iteration. This will be done by using the word “continue”.**

```
N = -5
dividend = 10.0
while N < 5:
    if N == 0:
        N += 1      # What happens if we remove this line ?
        continue
    print("Result of dividing {} by {} is {}".format(dividend, N, dividend / N))
    N += 1
```

# Exercise 3

- Find 5 color images in internet, download them, and create a Python script with a list that contains the full-path to each of them.
- Open the images in color mode using OpenCV, and convert them into gray-scale images using for loops (one iteration per image). Show resulting image.
- Compute the gray-scale histogram of the images using while loops. Print it in terminal.

- Open image in color mode:

```
import cv2
myImg = cv2.imread('filepath.png', cv2.IMREAD_COLOR)
```

- Access to single pixel in color images:

```
# In color images, "color" is a list with 3 elements
color = myImg[i, j]   # The channels order is B - G - R
```

- Convert color into Grayscale

```
Gray = np.array(0.299 * img[:, :, 2] + 0.587 * img[:, :, 1] +  0.114 * img[:, :, 0], dtype=np.uint8)
```

- Show image

```
import cv2
cv2.imshow("Image name", myGrayImg)
cv2.waitKey()
cv2.destroyAllWindows()
```

# Step 4: Functions

- Piece of code that you have to use a lot → Function
- Syntax:

```
def func_name(arg1, arg2):
    # My code
    return
    # return output_var
```

- It receives none or N arguments (list of elements between parenthesis), and it can return none or M variables.
- Usage:

```
var_1 = 20
var_2 = 30
# ...
ret_val = func_name(var_1, var_2)
```

# Step 4: Functions

- Example: Function for the distance between two vectors

```
import numpy as np
def compute_distance(v1, v2):
    # We compute the difference
    diff = [v1[0] - v2[0], v1[1] - v2[1], v1[2] - v2[2]]
    # We compute the square of the difference, element-wise
    power_of_diff = [diff[0] ** 2, diff[1] ** 2, diff[2] ** 2]
    # We sum all the components
    sum = power_of_diff[0] + power_of_diff[1] + power_of_diff[2]
    # We return the square root of the previous sum
    return np.sqrt(sum)
```

# Step 4: Functions

- Usage:

```
vector_1 = [2,3,4]
vector_2 = [1,1,1]
vector_3 = [-1,2,-5]
# This function is called, but the return value is stored nowhere
compute_distance(vector_1, vector_2)
# This function is called, and the returned distance is stored in the variable dist_1_3
dist_1_3 = compute_distance(vector_1, vector_3)
# In this case, the result of the function is directly printed in the terminal
print("The distance between {} and {} is {}".format(
      vector_2,
      vector_3,
      compute_distance(vector_2, vector_3)))
```

# Step 4: Functions

- When functions appear, the used variables can be locally or globally defined.
- A local variable is a variable defined inside a function, and it cannot be accessed from the outside of the function block.
- A global variable is accessible from any part of the program.
- ***NOTE: Never, but NEVER use global variable. Either pass the variables as arguments, or define them locally. If anybody can access to a global variable, it can be read and written by anybody, at any time, and this type of variables are prone to errors. Only very specific cases justify the usage of global variables. In general, they can be avoided.***

# Step 4: Functions

```
myGlobalVar = [1,5,67]
def my_func():
    myLocalVar = [2,5,6]
    return myLocalVar

# This print is executed correctly
print("A global variable is {}".format(myGlobalVar))
# This print fails
print("A local variable is {}".format(myLocalVar))
```

```
myGlobalVar = "Initialized"
def my_func_2():
    global myGlobalVar
    myGlobalVar = "Global Variable"

print(myGlobalVar)
my_func_2()
print(myGlobalVar)
```

# Exercise 4

- Modify the distance function so that it works independently of the vector size (i.e., I want to use the same function for vectors of size 2, 3, 4, 5, 6, etc)
- Add checks: We cannot compute distances with vectors that are of type None, or their size are not the same (I cannot compute the size between a 2D and a 3D vector). If any of these cases appear, the function must return None
- Create some test cases, and print the results.

# Bonus: Try / Except blocks

- In Python, when a logic problem appears, and it cannot be resolved, the system will throw an ***Exception***.
- If the exception is not handled, the program ends, and generally, a message will explain what happens.
- Examples:

```
In [28]: a = 'This is an string'

In [29]: int (a)
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-29-a15cb3214373> in <module>()
----> 1 int (a)

ValueError: invalid literal for int() with base 10: 'This is an string'
```

# Bonus: Try / Except blocks

- Examples:

```
In [30]: 5 / 0
---------------------------------------------------------------------------
ZeroDivisionError                         Traceback (most recent call last)
<ipython-input-30-adafc2937013> in <module>()
----> 1 5 / 0

ZeroDivisionError: division by zero
```

```
In [31]: myDict = {}

In [32]: myDict['Jon']
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
<ipython-input-32-d4f91d11b69b> in <module>()
----> 1 myDict['Jon']

KeyError: 'Jon'
```

# Bonus: Try / Except blocks

- If we create our own functions, we can generate exceptions too, when a situation cannot be solved, or the variables have a wrong type.

```
def my_func(var_arg):
    if var_arg == 0:
        raise ValueError("This function will not work if the input variable is zero!")
    return 100 / var_arg

my_func(0)
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-33-28a915b5501e> in <module>()
      4     return 100 / var_arg
      5
----> 6 my_func(0)

<ipython-input-33-28a915b5501e> in my_func(var_arg)
      1 def my_func(var_arg):
      2     if var_arg == 0:
----> 3         raise ValueError("This function will not work if the input variable is zero!")
      4     return 100 / var_arg
      5

ValueError: This function will not work if the input variable is zero!
```

# Bonus: Try / Except blocks

- To handle the different exceptions, it exists a block called ***try and except***, that works similarly to an IF statement.
- Syntax:

```
try:
    # Code to be tested
except type_of_exception:
    # What to do in case of error
```

- ***type_of_exception*** is the type of error we want to handle (KeyError, ValueError, ZeroDivisionError, etc). If we do not specify the type of exception, it will catch all the cases.

# Bonus: Try / Except blocks

- In general, it is recommended to specify the type of exception we want to catch. Then, if another type of exception appears, and we did not consider it, we will know.
- A good practise is to place in the try block a little amount of lines that might fail (in the best case, only one), and not all the code into a single try block. This will help to debug the code, and it will help us to identify the weak points of each piece of code.

# Bonus: Try / Except blocks

- Example: function to detect if the user entered a valid number.

```
def convert_string_to_number(myStr):
    outputValue = None
    try:
        outputValue = int(myStr)
        print("The input string {} is an integer".format(myStr))
    except ValueError:
        print("The input string {} is not an integer".format(myStr))

    if outputValue is None:
        try:
            outputValue = float(myStr)
            print("The input string {} is a float".format(myStr))
        except ValueError:
            print("The input string {} is not a float".format(myStr))

    return outputValue
```

# Bonus: Try / Except blocks

- Example: function to detect if the user entered a valid number.

```
userInput = input(“Please, enter a number”)
print(convert_string_to_number(userInput))
```

- ***Note that, if an exception occurs, the failing instruction and the ones that follow, are not executed!!***

- In the previous example, if the value is not an integer, for instance, ***outputValue*** does not change its value, and the print that follows is not executed. The interpreter jumps to the exception block directly.

# Bonus 2: Comments and documentation

- When we write code, it is important for everybody, to understand what the program is doing while we read it.
- If we work into a team, we write a program that our teammates have to understand later on, or if we start a project, we stop it, and then we restart again a few months later on, we should be able to understand what we have done so far.
- Documenting the code is important.
- We can write comments to explain what a piece of code is doing, or what a function will do, the arguments it requires and what it returns.

# Bonus 2: Comments and documentation

- Example: documenting a piece of code in the middle of the program

```
# This code does the histogram transfer by doing the equalization by quartiles
for g_d in np.arange(m_d, M_d, 1):
    while g_r < M_r and P_d[g_d+1] < 1.0 and P_r[g_r+1] < P_d[g_d+1]:
        g_r = g_r + 1
    ## The function WHERE returns us a matrix where the first row is the row
    # and the second row is the column of an element in dis_image_matrix
    # that has the same value as g_d
    K[np.where(dis_image_matrix == g_d)] += g_r

K = np.array(K, dtype=np.uint8)
return K
```

# Bonus 2: Comments and documentation

- Example: documenting a function with a block of comments

```
import numpy as np
def compute_distance(v1, v2):
    '''
    Compute the Euclidean distance between two vectors.
    The two input vectors must have the same size,
    and each element must be a numeric value (no strings).
    The size of the input vector must be N > 0.

    Args:
        v1: First vector.
        v2: Second vector.

    Returns:
        Float value that represents the Euclidean distance
        between v1 and v2.
    '''
    # What happens without these two lines ?
    ...
```

# Bonus 2: Comments and documentation

- Not every line of code have to be documented.
- If the operation done by the line is obvious (for instance, creating an array or the name of the function / variables are explanatory enough), we do not documenth them.
- Generally, exceptional lines that are unusual for the type of operation we are doing, or a general explanation of the block of code, have to be included.
- Functions are always documented at the beginning, explaining its task, the arguments and the return values.

# Step 5: Classes and OOP

- What we have done until now is called “Functional programming”, since the program is organized as functions that we call in order to get the results we want.
- Python is, in fact, an Object Oriented Programming (OOP) language.
- The idea behind this paradigm is that we can encapsulate (i.e., enclose) all the code that corresponds to a single object.
- In OOP, an ***object*** is nothing else than a variable. This variable has a type that is described by a Class definition.

# Step 5: Classes and OOP

- The ***class definition*** is a description of what the object has (different variables that hold certain data), and what we can do with it (functions that work this data).
- The variables of the class are called ***Members***, and the functions that belong to the class are called ***Methods***.
- We can create several objects of the same class (imagine that we are declaring several integer variables, or some string variables).
- Each of these objects will have the same members, but each object can have a different values on those variables.
- Each object created is said to be an ***instance*** of the class we want to use.

# Step 5: Classes and OOP

- Syntax:

```
class myObjectClass(object):
    def __init__(self):
        print("An object of type myObjectClass has been created")

    def method_1(self, arg_1, arg_2):
        # Do something
        return

    def method_2(self, arg_1, arg_2, arg_3, arg_4):
        # Do something
        return

    def method_3(self):
        # Do something
        return

myObj = myObjectClass()
```

# Step 5: Classes and OOP

- Each time a new object is created, the function called ***__init__*** will be executed automatically. It is called ***Constructor***.
- We can define as many methods as we want, they can receive as many arguments, and return as many outputs as we want.
- To define a method, there are some considerations:
  - **The first argument is the argument called *self*. It indicates that this method belongs to the class.**
  - **All the methods are defined as any other function, indented with regard to the class statement.**
  - **We should include an __init__ method. In it, we initialize the class members.**

# Step 5: Classes and OOP

- To define a method, there are some considerations:
  - **Each variable that belongs to the class, have to be defined by adding “self.” as a prefix. If self is not added as prefix, the variable is local, and it will disappear when the method finishes.**
  - **Any variable that belongs to the class can be defined in any method. As a rule of thumb, it is better to define all of them in the __init__ function, to avoid wrong-order-function-call errors (we call a function that expects a variable to exists, and it has not been created nor initialized yet).**

# Step 5: Classes and OOP

- To define a method, there are some considerations:
  - **Each time we call a method inside the class definition, we have to add the prefix “self.”. If not, Python will search this function as defined outside the class.**
  - **When we want to use the class, we define an object by assigning the Class name to a variable, and then if we want to use the class, we write the object name, we add a “ . ”, and then we write the method or member name we want to use.**
  - **As soon as the indentation is reduced further than the class definition level, the class statement is finished.**
  - **When we create an instance of a class, we can pass arguments to the constructor, as if it where a function.**

# Step 5: Classes and OOP

- Example: Point2D class

```
import numpy as np
class Point2D(object):
    def __init__(self, x, y):
        self.x = self.convert_to_number(x)
        self.y = self.convert_to_number(y)
        if self.x is None or self.y is None:
            raise ValueError("X and Y have to be numeric values!")

    def convert_to_number(self, inVal):
        outputValue = None
        try:
            outputValue = int(inVal)
        except ValueError:
            print("The input string {} is not an integer".format(inVal))

        if outputValue is None:
            try:
                outputValue = float(inVal)
            except ValueError:
                print("The input string {} is not a float".format(inVal))

        return outputValue

      def get_point(self):
          return (self.x, self.y)
```

# Step 5: Classes and OOP

```
def distance_to_coordinates(self, target_x, target_y):
    np_p1 = np.array((self.x, self.y))
    np_p2 = np.array((target_x, target_y))

    # We compute the difference, element-wise
    diff = np_p1 - np_p2
    # We compute the square, element-wise
    power_of_diff = np.power(diff, 2)
    # We sum all the components
    sum = np.sum(power_of_diff)
    # We return the square root of the previous sum
    return np.sqrt(sum)

def distance_to_point(self, p):
    return self.distance_to_coordinates(p.x, p.y)

def distance_to_origin(self):
    return self.distance_to_point(0,0)
```

# Step 5: Classes and OOP

- Usage:

```
class Point2D(object):
    def __init__(self, x, y):
...

p1 = Point2D(5,2)
p2 = Point2D(3,3)
x3 = 8
y3 = 7
dist1 = p1.distance_to_coordinates(x3, y3)
print("Distance from {} to ({}, {}) is: {}".format(p1.get_point(), x3, y3, dist1))

dist2 = p1.distance_to_point(p2)
print("Distance from {} to {} is: {}".format(p1.get_point(), p2.get_point(), dist2))
```

# Step 5: Classes and OOP

- Why should we use classes ?
  - **It helps to make the main code more readable.**
  - **It is a way to better organize the code. All the functions related to a single object are included in the class definition.**
  - **It is easier to debug and to maintain the code. Each time there is a bug, we can focus our attention to a single class, and not the entire code.**
  - **For the programmer, it serves to break the problem into small pieces, and then we can work on each of them separately.**

# Step 5: Classes and OOP

- Why should we use classes ?
  - **There is another reason, that we will not explore in this course, but it is important to know. This is the concept of *inheritance*.**
  - **A class can inherit from another class (called base or parent class), and this means that all the methods and members defined in the parent now are also present in our class. Our class is now a *child* class of the *parent*.**
  - **Another consequence of inheritance is that if the parent is of type A, then the child B can also be considered as of type A. The contrary is not true.**
  - **The difference between A and B will be the methods and members we add into the class B, which will not be present in A.**
  - **We can inherit several classes into a single child.**
  - **We have to call the constructor of each parent class during construction of the child class.**

# Step 5: Classes and OOP

- Why should we use classes ?
  - **In Python, to inherit another class, when we define the class, after the name, we place between parenthesis, the name of the parent class.**
  - **If several classes want to be inherited, we can separate them with “, ”.**

```
class A:
    def __init__(self):
        # Do something
        return

    def method_1(self, arg_1):
        # Do something
        return

    def method_2(self):
        # Do something
        return
```

```
class B(object, A):
    def __init__(self):
        A.__init__(self, arg_1, arg_2,...)
        # Do something
        return

    def method_3(self, arg_1):
        # Do something
        return

    def method_4(self):
        # Do something
        return
```

# Exercise 5

- Create the student class. This class have to contain the members student ID (string - a combination of letters and numbers), student name (string), student last name (string), nationality (string), address (string), course it is signed up (string - license, bachelor, master or PhD).
- All the fields have to be set up in the constructor, and then we have several accessors to the different variables (methods like get_last_name, get_address, get_date_of_birth).
- print_student_info method: Print nicely the student data into the terminal.
- Add a user interface in the terminal, using print and input. The menu should include the options: Create new student and Retrieve student information.
- Each student entered has to be stored  into a dictionary. The key value to access to them is the student ID.
- When the user decides to create a new student, each field has to be requrested, stored into an object of type Student, and stored into the dictionary.
- When the user decides to check an existing student, it provides the ID, and all the student data has to be shown in the terminal.

# Weekly project

- Create a Robot class, a control class, and a user interaction class. The aim is to simulate a go-to-target program.
- The robot localization is done by a 3D vector: (X, Y, theta). (X,Y) is the localization, theta is the orientation with regard to the horizontal line.
- With the Robot class, we should be able to: query and set the robot name, query the current position, set and query the forward speed (positive number in m/s), a rotational speed (in rad / sec). We should also be update the robot position every a fixed amount of time (discretization time).
- ***The Robot has a maximum speed of 0.3 m/s and 0.3 rad / sec***. If higher speeds want to be set, this maximum value must be used.

# Weekly project

- The control class would be a proportional controller, and at each iteration, it computes the rotational speed, and the forward speed. We should be able to set the proportional gains, and to call a function that receives the actual position, the final position, and it computes the required speeds.
- The output of the controller must first correct the robot orientation in place (i.e., only change the theta, while the forward speed is 0). If the theta error is smaller than 5 degrees, the forward speed can be non-zero, while the theta orientation stills being corrected. If at any time, the theta error becomes larger than 5 degrees, the robot must stop and correct the angle again.
- Set an iteration time of 1 mS, and the approximation error to 1 cm.

# Weekly project

- Main code snippet:

```
import time

robot = myRobotClassName(robot_name, max_speed, initial_pos)
controller = myControlClassName(forward_speed_gain, rotational_speed_gain)

iteration_time_sec = 0.001
target_pos = [target_x, target_y]

# Initialize robot time:
t_k = time.time()
robot.init_task(t_k)
while True:
    if distance_to_target(actual_robot_pos, target_pos) < 0.01:
        break
    else:
        # Compute forward and rotation speed with controller
        # set speed to robot
        t_k = current_time
        time.sleep(iteration_time_sec)
        currrent_time = time.time()
        # update robot pos with t_k and current_time

# Set robot speed to zero
print("{} arrived to the target!".format(robot_name))
```

# Thank you!