



CEPLAS

Cluster of Excellence on Plant Sciences

QBio104

Prof. Dr. M. Lercher



Dr. Vittorio Tracanna



Dr. Mayo Röttger



## Information

Lecture: **Wednesday 15:30-18:00 UoC 0.024 - Biozentrum**

Exercise: **Tuesday 12:30-14:30 HHU 24.21.U1.24**

- Provided each Wednesday after the lecture
- Submit (Ilias) until each Tuesday 8:00
- Random selection of students and presentation of programs

Dr. Vittorio Tracanna: 08.10.24 – 20.11.24

Dr. Mayo Röttger: 26.11.24 – 29.01.25

Exam QBio104: Programming (details at the end of the module)

- 26.02.25 10:30 – 13:30 2541.U1.22 (EDV-Übungsraum)
- 21.03.25 10:30 – 13:30 2541.U1.22 (EDV-Übungsraum)

Obey to the “Merkblatt Bild-und-Ton” that you can find at Ilias

To get in touch with me regarding anything course-related: [vtracann@uni-koeln.de](mailto:vtracann@uni-koeln.de)



CEPLAS

Cluster of Excellence on Plant Sciences

# Introduction to Python

QBio104

Material modified from M. Röttger and A. Schrader





# Programming in Python





# Programming in Python

## Topics

- Variables, character strings, integer, and float objects
- Implicit and explicit type conversion
- Arithmetic operators and functions
- Python modules
- Logical operators
- Create a program
- Reading user input



# Programming in Python

## Basic Python information

- General purpose, **high-level** programming language
- Comprehensive standard library
- Dynamic datatypes
- Object oriented programming (OOP)
- Open community based development model
- Developed in 1990 by Guido van Rossum







# Python

## Origin



<https://humanities.blogs.ie.edu/2013/11/monty-python-to-reunite-for-stage-show.html>





# Coding practices

## A word of warning

“Always code as if the guy who ends up maintaining your code will be a violent psychopath who knows where you live.”

(Damian Conway in Perl Best Practices)

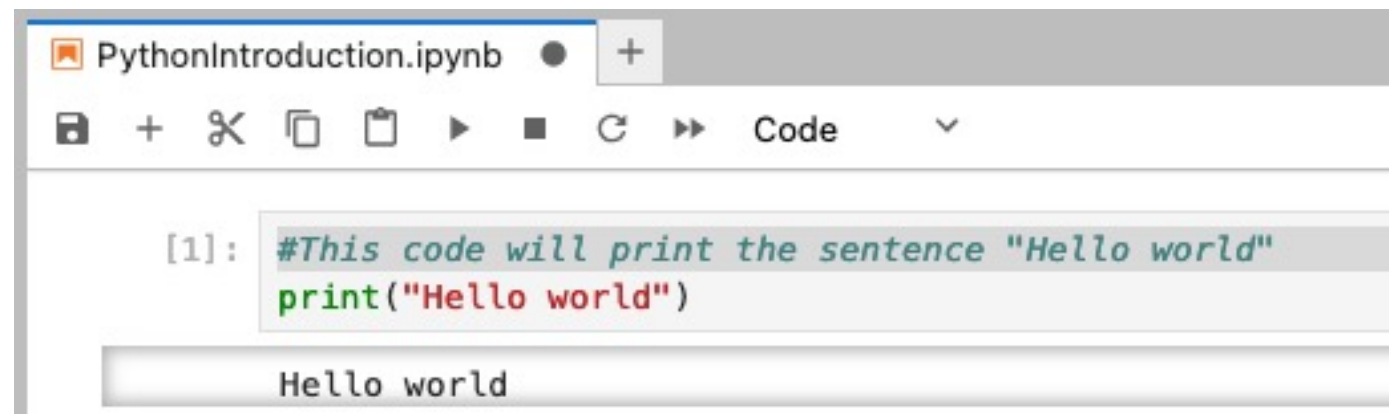




# What is code

## Coding101 in QBio104

- It's a set of instructions written in text (source code)
- We “speak” in Python (the programming language)
- The machine knows how to translate (compiled) the “high-level” instructions into bytecode (less human-readable “lower-level” machine instructions) that can then be interpreted by the machine (think of the 01010101011101001001)



The screenshot shows a Jupyter Notebook window titled "PythonIntroduction.ipynb". The interface includes a toolbar with icons for saving, adding, deleting, copying, pasting, running, and other actions. Below the toolbar, there is a code cell labeled "[1]:". The code inside the cell is: 

```
#This code will print the sentence "Hello world"
print("Hello world")
```

 The output of the code cell is displayed below the code: "Hello world".



# Comments

Keep you safe from the violent psychopath who knows where you live

- Every program should begin with a comment that shortly summarizes its function.
- Text after the # character is detected as comment and ignored by the interpreter.

```
PythonIntroduction.ipynb +
```

Code ▾

```
[1]: #This code will print the sentence "Hello world"  
    print("Hello world")
```

Hello world



# Python is case sensitive but relaxed with spaces

- Python distinguishes between UPPER and lower case.
- “print” is not the same as “Print” or “PRINT”
- However, *print(“Hello world”)* and *print (“Hello world”)* are the same
- Error messages in python are generally informative and clear

```
PythonIntroduction.ipynb +
+ ✂ 📄 ▶ ■ ↺ ⏩ Code ▼

[1]: #This code will print the sentence "Hello world"
    print("Hello world")
    Hello world

[2]: #This code will also print the sentence "Hello world"
    print ("Hello world")
    Hello world

[3]: #this code will not, because Print is not a defined function. Upper and lowercase matters in python!
    Print ("Hello world")

-----
NameError                                Traceback (most recent call last)
Cell In[3], line 2
      1 #this code will not, because Print is not a defined function. Upper and lowercase matters in python!
----> 2 Print ("Hello world")
NameError: name 'Print' is not defined
```



# The function `print (...)`

- The function `print` writes a string to the standard output device
- Function arguments are written within parenthesis after the function name
- If more than one argument is needed, the arguments should be separated by commas

```
PythonIntroduction.ipynb +
+ ✂ 📄 ▶ ■ ↺ ⏩ Code ▾

[1]: #This code will print the sentence "Hello world"
    print("Hello world")
    Hello world

[2]: #This code will also print the sentence "Hello world"
    print ("Hello world")
    Hello world

[3]: #this code will not, because Print is not a defined function. Upper and lowercase matters in python!
    Print ("Hello world")

-----
NameError                                Traceback (most recent call last)
Cell In[3], line 2
      1 #this code will not, because Print is not a defined function. Upper and lowercase matters in python!
----> 2 Print ("Hello world")
NameError: name 'Print' is not defined

• [4]: #this code will print "Hello world" highlighting the syntax to give multiple inputs to one function
    print ("Hello", "world")
    Hello world
```



# The function `print(...)`

- `print(...)` finishes the output with a newline (“\n”), so that further writings will begin in the next line.

```
[5]: #this code will print "Hello" and "world" on two separate lines  
print ("Hello")  
print ("world")
```

```
Hello  
world
```





# Working environment

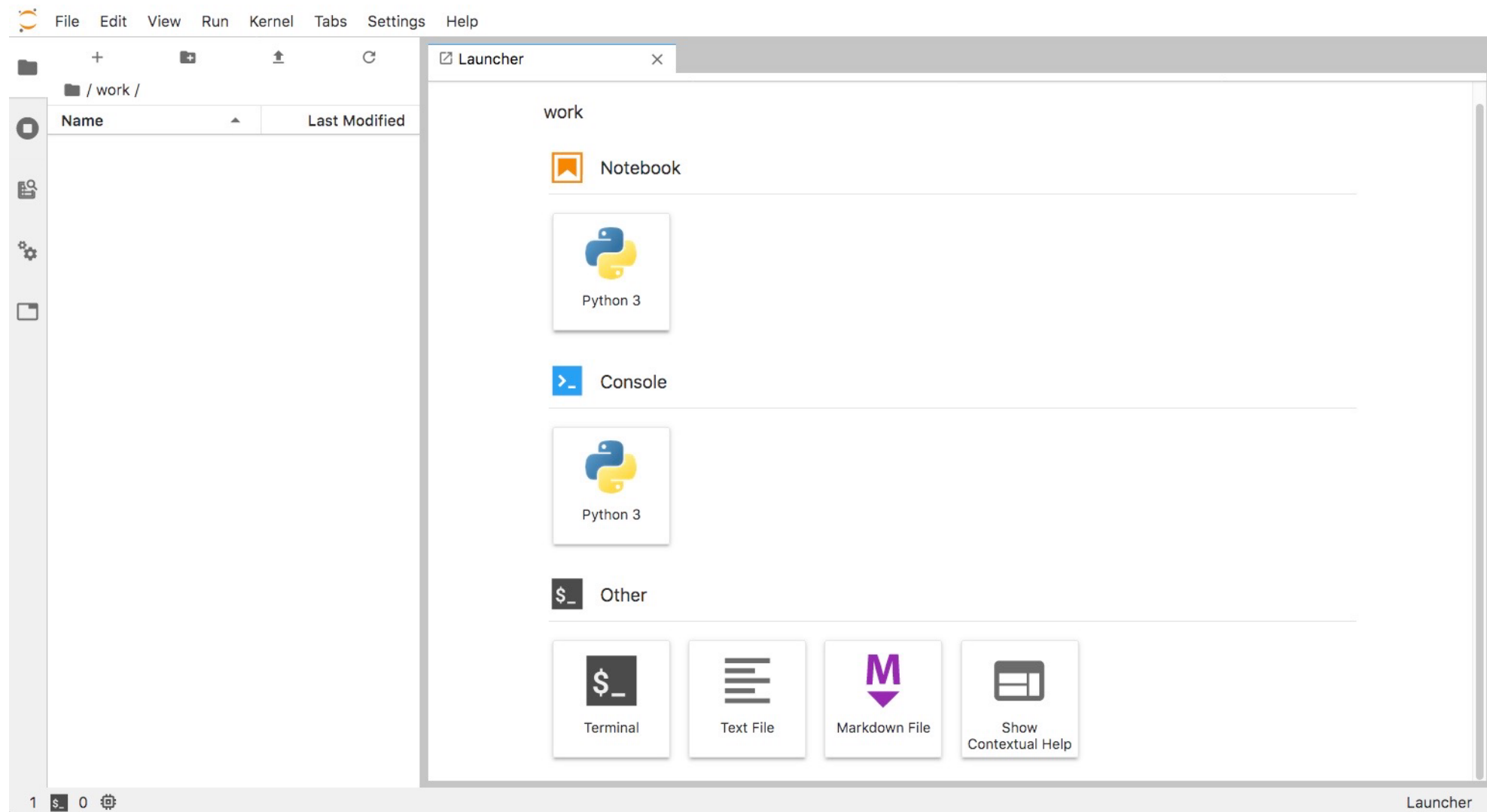
## Jupyter

- For programming in python, we only need a text editor besides the python installation.
- Scripts (.py text files containing the code) are usually run using the command line.
- Here we are using the JupyterLab environment:
  - Working environment as Web-Browser interface
  - We will write code in Jupyter notebooks for clear and structured documentation of the work
  - Integrated file system browser and text editor
  - Allows for live coding
  - Conversion of notebooks into numerous output formats (e.g.: PDF, TeX, HTML)
  - Programming in a notebook is very practical, because it is a detailed record of what is done and which programs or commands were executed in which order



# Working environment

## Creating a jupyter notebook





# Working environment

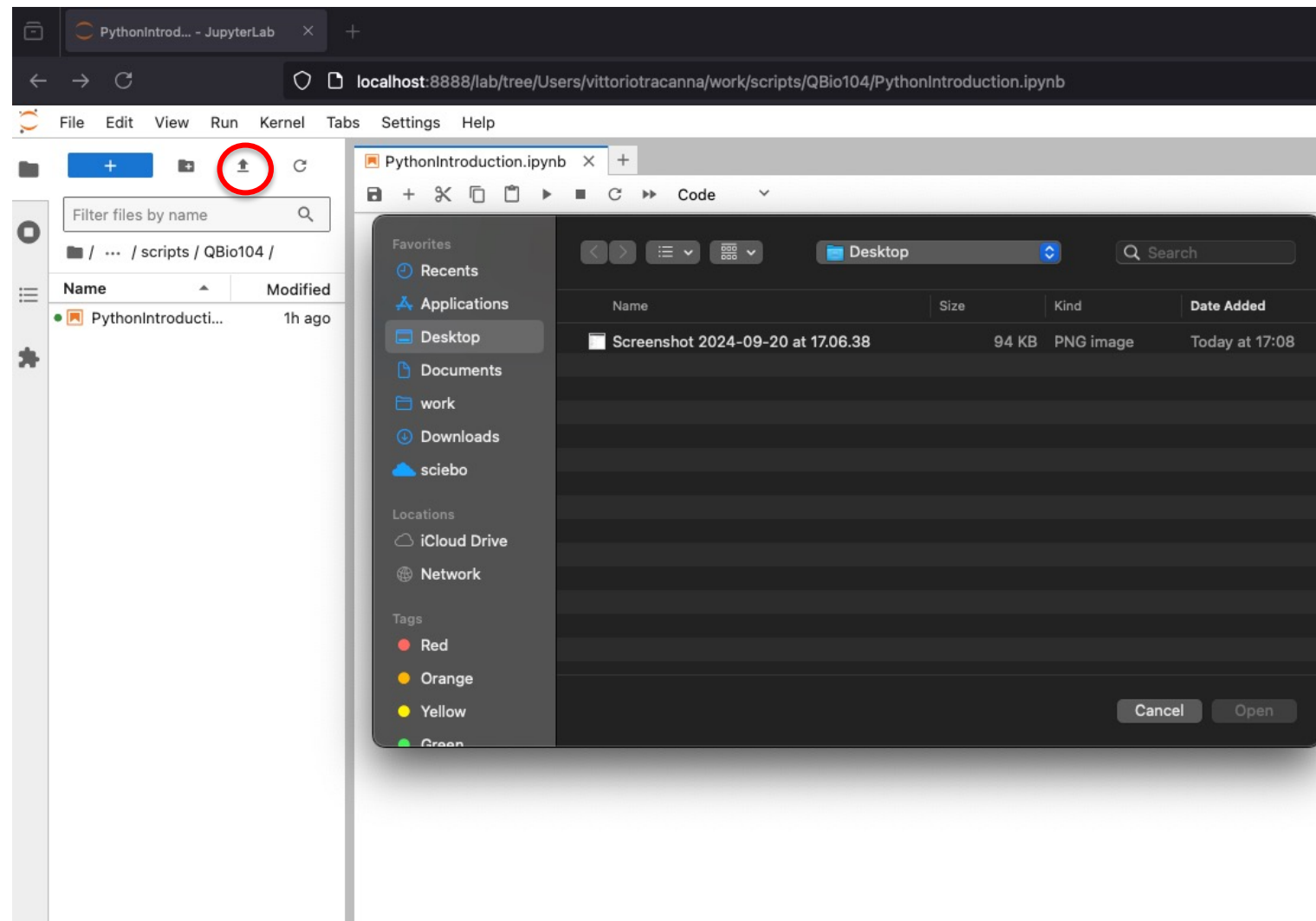
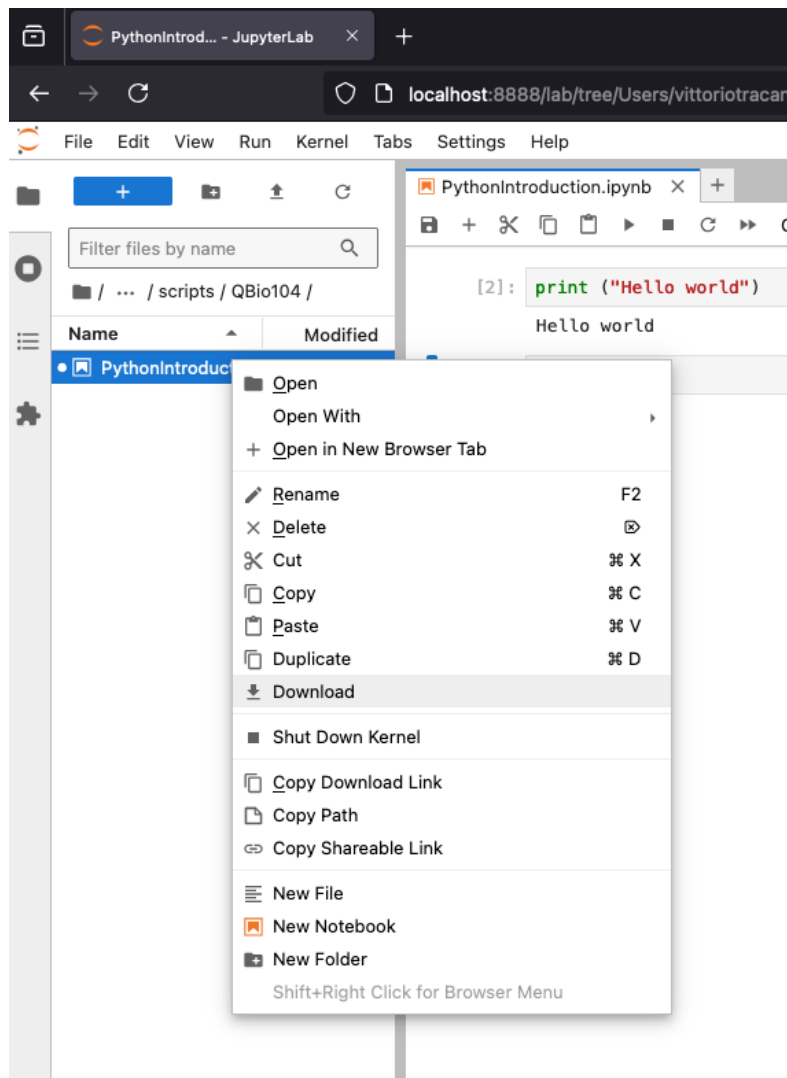
## Download and upload of files

- Use the download function to transfer files from the JupyterLab environment in the browser to your computer.
- This is especially important if JupyterLab is not installed locally but as part of container virtualisation or provided through the internet.
- You can find downloaded files in the standard download folder of your browser.
- Use the upload function to transfer files from your computer into the JupyterLab Environment or using drag & drop.



# Working environment

## Downloading and uploading files from the file browser

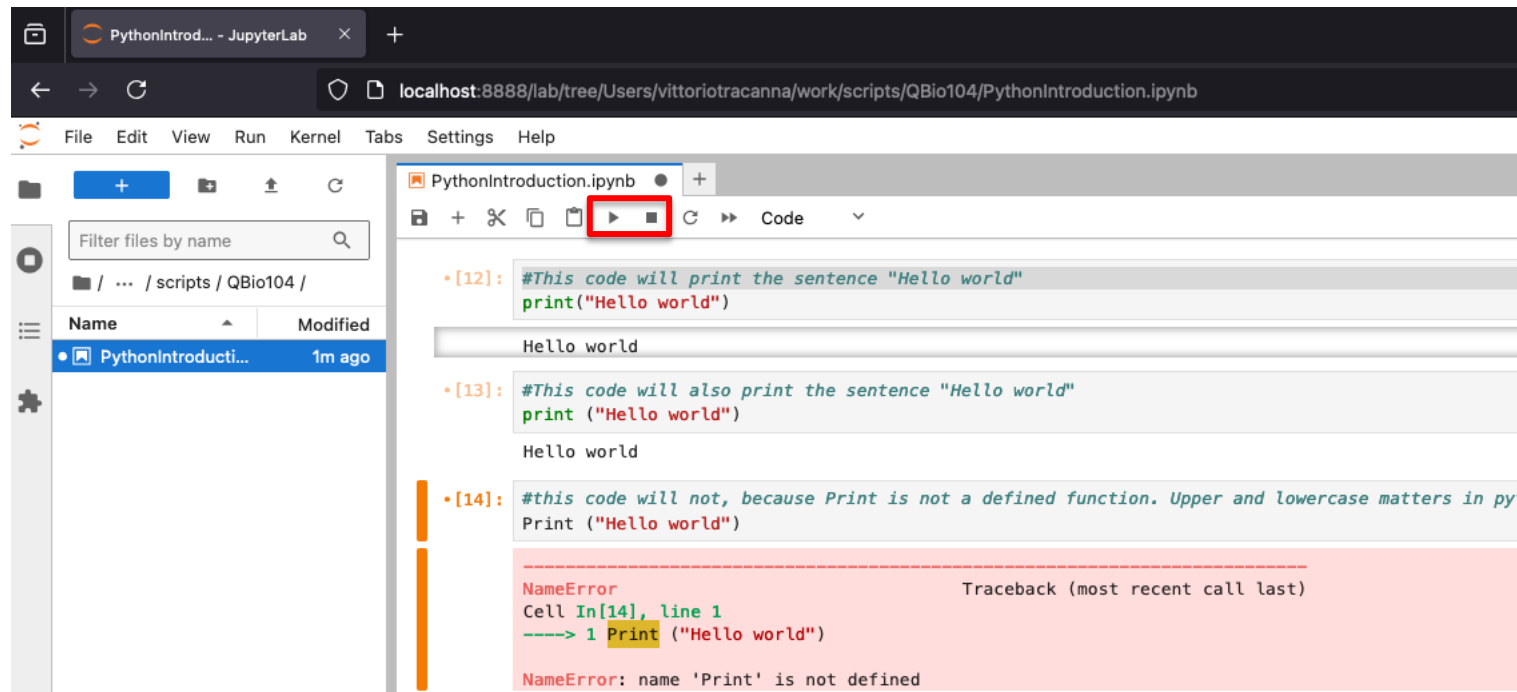




# Working environment

## Running a script

- To run code written in a Jupyter Notebook cell, press shift+enter or ctrl+enter
- The execution of code in a cell will be terminated (e.g. endless loops), when pressing the stop symbol







# Metacharacters

Special character combinations used to generate patterns

## Examples:

- `\n` is used to represent a `newline`. Whenever this combination of characters is found in a string, it will result in the string or displayed text to interrupt the current line and text will resume from the beginning of the next one
- `\t` is used to represent a `tab` (just like the keyboard button)

Hot questions: How would this code be displayed:

- `print ("This tex\t is not clear\n")`
- `print ("\tThis \tex\t is \not clear either")`



# Hot question metacharacters

Hot questions: How would this code be displayed:

- `print ("This tex\t is not clear\n")`
- `print ("\This \tex\t is \not clear either")`

```
[7]: #code from hot question number 1 regarding metacharacters:
print ("This tex\t is not clear\n")
print ("\This \tex\t is \not clear either")

This tex      is not clear

\This  ex      is
ot clear either

<>:3: SyntaxWarning: invalid escape sequence '\T'
<>:3: SyntaxWarning: invalid escape sequence '\T'
/var/folders/hf/xppf8p3d1x7cw6y752x0yy1h0000gn/T/ipykernel_7250/3486670291.py:3: SyntaxWarning: invalid escape sequence '\T'
print ("\This \tex\t is \not clear either")
```

What is the red square on the bottom?

It's a Warning (SyntaxWarning). It does not mean that the code gave an error (see Print() example) and therefore it did run\* to completion



# Docstrings and raw string mode

- Docstrings are used to describe text in a free form without using comments.
- Using `r` (standing for `raw`) before a string, modifies the behaviour of the print function, making it ignore metacharacters

```
[14]: """Using 3 " in a row at the beginning and end of one or more lines makes the whole text within these characters  
to be treated as comments and therefore skipped when running code.  
It is useful when writing extensive descriptions of the code. It is referred to as Docstrings"""  
  
print (r"Some\times we \tneed to prin\t \tex\t \tha\t shows combi\na\tions of me\tacharac\ters")
```

```
Some\times we \tneed to prin\t \tex\t \tha\t shows combi\na\tions of me\tacharac\ters
```



# Objects, Variables, Values and Types

## Technical definition and differences

**“Object”** is defined as the area of memory that holds the **“Values”** and is characterized by a **“Type”**

**“Variable”** is the name of an **“Object”**

**“Type”** specifies how to interpret the **“Values”** of an **“Object”**

**“Values”** are the actual data in an **“Object”**



# What are Object Types?

Python will try and implicitly guess which one to use

- Numerics:
  - int Integer number: `1103`
  - float Floating point number: `3.141592653589793`
  - bool Boolean type, `True` or `False`
- Sequences:
  - str Character string: `"Hello world"`
  - list List: `[1, 2, 3, 4]`
  - tuple Tuple: `('a', 'b', 'c')`
- Mappings:
  - set Set: `{ 'A', 'T', 'C', 'G' }`
  - dict Dictionary: `{ 'Kölsch': 2, 'Altbier': 1 }`





# Assignments

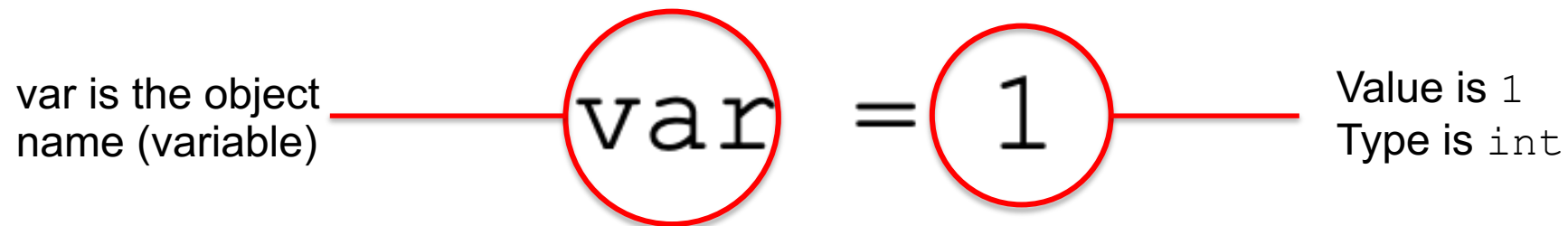
How to create an object, associate values and types

Objects can be assigned to variables using the assignment operator:

- `var = 1` Variable `var` points to an `int` object with value 1.
- `var = 1.0` Variable `var` points to a `float` object with value 1.0.

Explicitly casting the object type during assignment:

- `var = float(1)` Variable `var` points to a `float` object with value 1.0.





# Sequence data type

This data type store a collection of items that can be accessed with an index

**String** is a group of characters, alphanumerical or special symbols. It is defined using “” or “”

```
my_DNA = "ATCGATCG"
```

**List** is a collection of characters where every single entry can have a distinct type. It is defined using square brackets []

```
DNA_list = ["A", "T", "C", "G"]
```

```
Not_DNA_list = ["A", "altbier", 1, ["another", "list"]]
```

You can refer to one or more items in a list by specifying the index using [] after the variable name. Note that the index, position, in a list or string, starts from 0

```
[16]: #store the list of nucleotides as DNA_list.  
#Then print the list. Then print the 3rd element of the list (note that the index, position, in a list or string, starts from 0)  
#Finally, store my_DNA as a sequence of nucleotides and print the first element (note that the index, position, in a list or string, starts from 0)  
DNA_list = ["A", "T", "C", "G"]  
  
print (DNA_list)  
print (DNA_list[2])  
  
my_DNA = "ATCGATCG"  
print (my_DNA[0])  
  
['A', 'T', 'C', 'G']  
C  
A
```





# Hot question index

We can specify a single or multiple position in an index

- To specify a range use ":" excluding the second value
- Indexes behave like integers and can be used with operations
- Negative values return elements from the end starting from -1

+	0	1	2	3	4	5	6	7
str	A	T	C	G	A	T	C	G
-	-8	-7	-6	-5	-4	-3	-2	-1

Given the string `my_DNA = "ATCGATCG"` What will be the result of

```
print (my_DNA[1-6])
```

```
print (my_DNA[2:-2])
```

```
print (my_DNA[11])
```





# Hot question index

## Answers

```
#code from hot question number 1 regarding indexes:
```

```
print (my_DNA[1-6])  
print (my_DNA[2:-2])  
print (my_DNA[11])
```

G  
CGAT

-----  
**IndexError**

Traceback (most recent call last)

Cell In[23], line 5

```
    3 print (my_DNA[1-6])  
    4 print (my_DNA[2:-2])  
----> 5 print (my_DNA[11])
```

**IndexError:** string index out of range



# Building functions

## Automate repetitive tasks

Beyond built-in functions, you can also build your own set of instructions that are called by a function.

We initiate a function using `def`

Followed by the name of the new function and the parameters that we need to execute the function and a colon :

The lines of code within a function will be “indented” (using `tab`)

At the end of a function, we use `return` to define what will come out of the function. Hint, it can be more than just one variable

```
#Here we will build our first function (of many!)  
#We have a specific syntax to create functions
```

```
def cleanDNAString(my_DNA):  
    '''  
    Functions should be accompanied by DocStrings that describe its usage.  
  
    #first we will remove spaces using replace  
    #then we will remove any letter that is not ATGC from the string  
  
    @param my_DNA: string that we want to polish  
    @return my_DNA_polished: string without unwanted characters  
  
    '''  
    my_DNA = my_DNA.replace(' ', '')  
    my_DNA = my_DNA.upper()  
    return my_DNA
```

```
#to call a user defined function, we use the name followed by the arguments within parentheses
```

```
my_DNA = cleanDNAString("  ATcgG AGatC")  
print (my_DNA)
```

```
ATCGGAGATC
```





# Referring to the same object and Boolean operators

## Hot question boolean and operators

- We use = to assign variables
- We use the operator == and != to compare objects

Question, what are the results of the following cell?

```
#Here we show how to use operators and understand booleans
```

```
my_DNA = "ATCG"  
your_DNA = my_DNA
```

```
print (my_DNA)  
print (your_DNA)  
print (my_DNA == your_DNA)  
print (my_DNA != your_DNA)
```

```
my_DNA = "CGTA"  
print (my_DNA == your_DNA)  
print (my_DNA != your_DNA)
```



# Hot question boolean and operators

## Results

```
#Here we show how to use operators and understand booleans
```

```
my_DNA = "ATCG"  
your_DNA = my_DNA
```

```
print (my_DNA)  
print (your_DNA)  
print (my_DNA == your_DNA)  
print (my_DNA != your_DNA)
```

```
my_DNA = "CGTA"  
print (my_DNA == your_DNA)  
print (my_DNA != your_DNA)
```

```
ATCG  
ATCG  
True  
False  
False  
True
```



# Variable name convention

Remember the Psycopath that knows where you live

- Do not use Umlaut
- The chosen variable identifier should already give a hint of what kind of objects or values the variable will reference to.
- Increased readability and context becomes clear
- Sub-optimal variable identifiers:
  - `a, b, c`
  - `myVar, myVar2, myVarFinal, myVarFinalFinal2`
- Better variable identifiers:
  - `protein_sequence, organism_name,`



CEPLAS

Cluster of Excellence on Plant Sciences

# Introduction to Python

Break – 15 minutes





Python knows the basic calculation specifications and laws, e.g. the conventions of operator precedence (point before line calculation) and bracketing for int and float objects.

Operator	Function
+	Addition
-	Subtraction
*	Multiplication
/	Division
%	Modulo (Rest after division)
**	Exponent



# Differences between assigning and referring to an Object

There is a difference between assigning a value to an object versus using the object in a function

```
#Here we show the difference between assigning a value to an Object and using it in a function

#This will not change my_DNA
my_DNA = "ATCG"
print (my_DNA + 'CGTA')
print (my_DNA)

#This will change my_DNA
#note that "+=" in my_DNA += 'CGTA' is equivalent to my_DNA = my_DNA + 'CGTA'
my_DNA += 'CGTA'
print (my_DNA)
```

```
ATCGCGTA
ATCG
ATCGCGTA
```



# Python functions

## Built-in functions and what are they

- As the name implies, they are a set of instructions that perform a specific task (or function). We can provide parameters to functions.
- We have been using functions all along! `print` is a function
- Python comes with multiple built-in functions (they are available as you start Python) with different objectives

```
#Here we show some python built-in functions
```

```
#len stands for length
```

```
print (len(my_DNA))
```

```
print (len('ATGC'))
```

```
#abs stands for absolute value
```

```
print (abs(5))
```

```
print (abs(-5))
```

```
#int stands for integer
```

```
print (int(5.5))
```

```
8  
4  
5  
5  
5
```



# Python methods

## Methods are functions tied to objects

- Methods offer behaviour specific to objects

```
#Here we will show how python methods differ from functions  
  
#to call a function, we state the function, followed by the object within parenthesis  
print (len(my_DNA))  
#to call a method:  
#we first declare which object we want to use, then we use a dot followed by the method name and its arguments  
#replace is a method that allows you to replace the occurrences of the first string with the second  
print (my_DNA.replace('T', 't'))  
  
print (len(my_DNA).replace('A', 'a'))  
  
#What will be the result of print (my_DNA)?
```

```
8  
AtCGCGtA
```

```
-----  
AttributeError                                Traceback (most recent call last)  
Cell In[47], line 10  
      5 #to call a method:  
      6 #we first declare which object we want to use, then we use a dot followed by the method name and its arguments  
      7 #replace is a method that allows you to replace the occurrences of the first string with the second  
      8 print (my_DNA.replace('T', 't'))  
----> 10 print (len(my_DNA).replace('A', 'a'))  
      12 #What will be the result of print (my_DNA)?  
  
AttributeError: 'int' object has no attribute 'replace'
```







# Building functions

## Automate repetitive tasks

Beyond built-in functions, you can also build your own set of instructions that are called by a function.

We initiate a function using `def`

Followed by the name of the new function and the parameters that we need to execute the function and a colon :

The lines of code within a function will be “indented” (using `tab`)

At the end of a function, we use `return` to define what will come out of the function. Hint, it can be more than just one variable

```
#Here we will build our first function (of many!)  
#We have a specific syntax to create functions
```

```
def cleanDNAString(my_DNA):  
    '''  
    Functions should be accompanied by DocStrings that describe its usage.  
  
    #first we will remove spaces using replace  
    #then we will remove any letter that is not ATGC from the string  
  
    @param my_DNA: string that we want to polish  
    @return my_DNA_polished: string without unwanted characters  
  
    '''  
    my_DNA = my_DNA.replace(' ', '')  
    my_DNA = my_DNA.upper()  
    return my_DNA
```

```
#to call a user defined function, we use the name followed by the arguments within parentheses
```

```
my_DNA = cleanDNAString("  ATcgG AGatC")  
print (my_DNA)
```

```
ATCGGAGATC
```





# Importing modules and functions

## Beyond built-in and user defined functions

- Imported modules are a key aspect of Python
- They are a collection of functions and methods to tackle a variety of tasks
- To import a module means to make it available in your code

*#Here we will import the module math and use its functions*

```
import math
logarithm_variable = math.log(10)
print(logarithm_variable)
squareRoot_variable = math.sqrt(10)
print (squareRoot_variable)
```

2.302585092994046

3.1622776601683795

We can also import only one function from the module

*#Here we will import only the log function*

```
from math import log
print (log(10))
```

2.302585092994046



# The help function

## A useful tool that can save time

- Modules can be large and complex with multiple associated functions
- Functions often have less known attributes that can be useful
- To list all the details of a module or function, we use the `help` function

```
help(math)
```

Help on module math:

NAME

math

DESCRIPTION

This module provides access to the mathematical functions defined by the C standard.

FUNCTIONS

`acos(x, /)`

Return the arc cosine (measured in radians) of x.

The result is between 0 and pi.

`acosh(x, /)`

Return the inverse hyperbolic cosine of x.

`asin(x, /)`

Return the arc sine (measured in radians) of x.

The result is between -pi/2 and pi/2.

`asinh(x, /)`

```
help(print)
```

Help on built-in function print in module builtins:

`print(*args, sep=' ', end='\n', file=None, flush=False)`

Prints the values to a stream, or to sys.stdout by default.

sep

string inserted between values, default a space.

end

string appended after the last value, default a newline.

file

a file-like object (stream); defaults to the current sys.stdout.

flush

whether to forcibly flush the stream.

```
help(math.log)
```

Help on built-in function log in module math:

`log(...)`

`log(x, [base=math.e])`

Return the logarithm of x to the given base.

If the base is not specified, returns the natural logarithm (base e) of x.



# Random module

## Pseudo-random number generator

- There are thousands of modules each with multiple functions.
- We do not need to “reinvent the wheel”
- For many functions, someone, somewhere made a module (and hopefully made it available)

```
#Here is another example of a module
```

```
import random
```

```
random_value = random.random()
```

```
print(random_value)
```

```
#some functions have keyword arguments instead of positional, where the function doesn't use the order  
#of the arguments to define their role in the function (for instance replace has positional arguments)
```

```
random_integer = random.randrange(start=1, stop=10, step=2)
```

```
print(random_integer)
```

```
0.6707874515317039
```

```
1
```



# Assignment

## Implicit assignment of object type

- Python tries to guess the object type by interpreting the input
- We can “cast” the object type when they are compatible
- The function `type(...)` can be used to return the current object type

*#Here we show how how to cast object types*

```
growth_temperature = 25
print (type(growth_temperature))
```

```
growth_temperature = float(25)
print (type(growth_temperature))
```

```
plant_species = 'arabidopsis thaliana'
print (type(plant_species))
```

```
plant_species = int('arabidopsis thaliana')
print (type(plant_species))
```

```
<class 'int'>
<class 'float'>
<class 'str'>
```

-----  
ValueError Traceback (most recent call last)

```
Cell In[83], line 10
      7 plant_species = 'arabidopsis thaliana'
      8 print (type(plant_species))
----> 10 plant_species = int('arabidopsis thaliana')
      11 print (type(plant_species))
```

ValueError: invalid literal for int() with base 10: 'arabidopsis thaliana'





# Providing input to a program

## Reading from standard input

- We can create a program that ask the user to provide an input
- For this end we can use the `input (...)` function

```
#Here we learn to give an input to a program  
species = input('What is your species?')  
print("You selected ", end="")  
print(species)
```

What is your species? | ↑↓ for history. Search hi

```
#Here we learn to give an input to a program  
species = input('What is your species?')  
print("You selected ", end="")  
print(species)
```

What is your species? Homo Sapiens  
You selected Homo Sapiens

- Python code can also be run from the command line
- In that case, we can use the argument vector function from the module `sys` (from system)



# Selecting volunteers

## A classic

- The import section goes at the beginning
- We break down the code into functions
- Functions are created when running the program but do not “run” until “called”
- `sys.argv[0]` is the script name (`SelectVolunteers.py`)

SelectVolunteers.py X

Week1 > SelectVolunteers.py

```
1  """
2  This code will be used to select volunteers to present the assignments in class.
3  We list the students and we will remove missing students.
4  The number of students to be selected will be given by the user.
5  The selected students will be printed out.
6  """
7
8  import random
9  import sys
10
11  def read_volunteer_number_from_input():
12      #This function reads the number of volunteers from the user input
13      volunteer_number = int(sys.argv[1])
14      return volunteer_number
15
16  def missing_student_name():
17      #This function reads the names of the students from the user input
18      missing_student = sys.argv[2]
19      return missing_student
20
21  def remove_missing_student(missing_student, student_names):
22      #This function removes the selected volunteers from the list of student names
23      student_names.remove(missing_student)
24      return student_names
25
26  def select_volunteers(volunteer_number, student_names):
27      #This function selects the volunteers
28      selected_volunteers = random.sample(student_names, volunteer_number)
29      return selected_volunteers
30
31  student_names = ['Carole', 'Ulrich', 'Thomas', 'Eva']
32  number_of_volunteers = read_volunteer_number_from_input()
33  missing_student = missing_student_name()
34  student_names = remove_missing_student(missing_student, student_names)
35  selected_volunteers = select_volunteers(number_of_volunteers, student_names)
36  print(selected_volunteers)
```



# How to run a program from the command line

Here we show MacOS/linux terminal

- Most operative systems have python already installed by default
- To run a program we have to type the name of the tool we want to use followed by the additional arguments we want to provide
- The program is usually stored in a text file with an extension that matches the tool we are using

```
[vittoriotracanna@MacBook-Pro-von-Vittorio ~ % python3 /Users/vittoriotracanna/work/scripts/QBio104/Week1/SelectVolunteers.py 2 Ulrich  
['Carole', 'Thomas']  
vittoriotracanna@MacBook-Pro-von-Vittorio ~ % █
```





CEPLAS

Cluster of Excellence on Plant Sciences

## Question time & Recap

Python101

Code comments

JupyterNotebook

Variables, objects, data types

Indexing

Functions and method

Creating a function

Modules

Reading input

Running scripts