Python Basic Course

Part I

Stefano Alberto Russo

Why should you listen to me?

An hybrid profile: BSc in Computer Science + MSc in Computational Physics

Started at CERN, as research fellow working on data analysis & Big Data

Then, 5 years in startups.

- Core team member of an IoT energy metering and analytics startup,
- Joined Entrepreneur First, Europe's best deep tech startup accelerator

Now back into research:

- INAF and UniTS, working on resource-intensive data analysis
- adjunct prof. of computer science at University of Trieste (Python)
- plus, experienced consultant for a number of private companies

Introduction

The course is structured to give you both:

- an overview of Python
- an approach to programming in general

This course does not aim at being exhaustive: we will leave out several topics.

Instead, the idea is to give you the approach and basics to let you go more in deep by yourself when you will need it!

The deal

1) Let’s try to keep it interactive.
2) Always interrupt if you have question, doubts, curiosities.
3) Try to carry out the exercises, or at least to sketch them.

Course structure

4h

1h30m	Lecture (part I)
30m	Exercise
15m	*Break*
1h15m	Lecture (part II)
30’	Exercise

4h

1h30m	Lecture (part III)
30m	Exercise
15m	*Break*
1h15m	Lecture (part IV)
30’	Exercise

Outline

- Part I: introduction and basics
  - What is Python
  - Tools and “hello world”
  - Basic syntax and data types
    - assignments, types and operators
    - conditional blocks and loops
- Part II: architecture
  - Functions
  - Scope
  - Built-ins
  - Modules
- Part IV: manipulating data
  - List operations
  - String operations
  - Reading and writing files
  - Dealing with wrong data
- Part VI: Pandas
  - Series and Dataframes
  - Common operations
  - How to read documentation

Outline

- Part I: introduction and basics
  - What is Python
  - Tools and “hello world”
  - Basic syntax and data types
    - assignments, types and operators
    - conditional blocks and loops
- Part II: architecture
  - Functions
  - Scope
  - Built-ins
  - Modules
- Part IV: manipulating data
  - List operations
  - String operations
  - Reading and writing files
  - Dealing with wrong data
- Part VI: Pandas
  - Series and Dataframes
  - Common operations
  - How to read documentation

What is Python

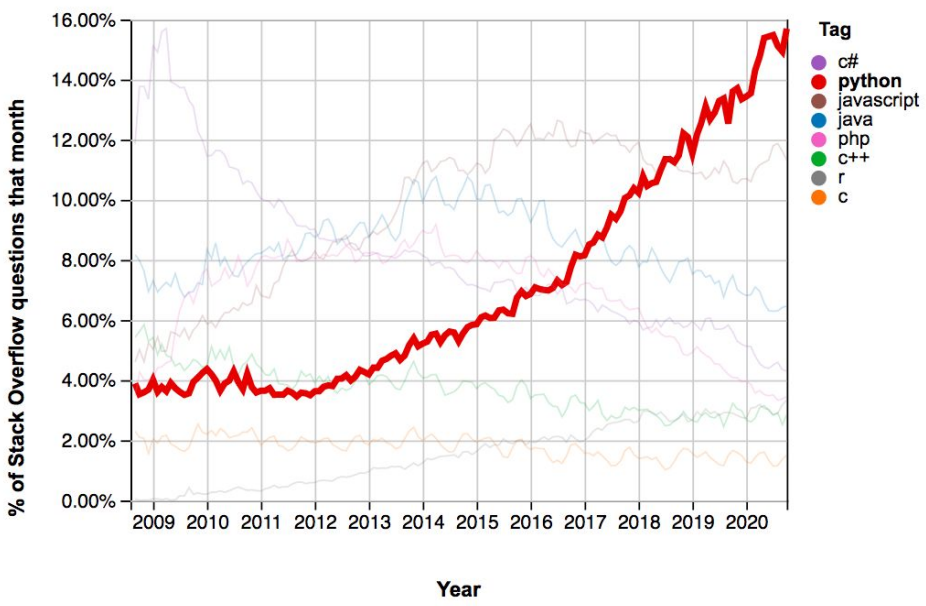
➔ *An interpreted language*

- Python is an interpreted language. This means that it does not need to be “compiled” into a machine language to be executed, like C, C++ or Fortran.
- Instead, Python code is directly “interpreted” and executed by the computer.
- For this reason, Python is much easier to use, in particular at the beginning and in general for interactive tasks.
- Python is also very powerful and has an enormous ecosystem of packages and libraries built around it.

What is Python

➔ *A constantly growing language*

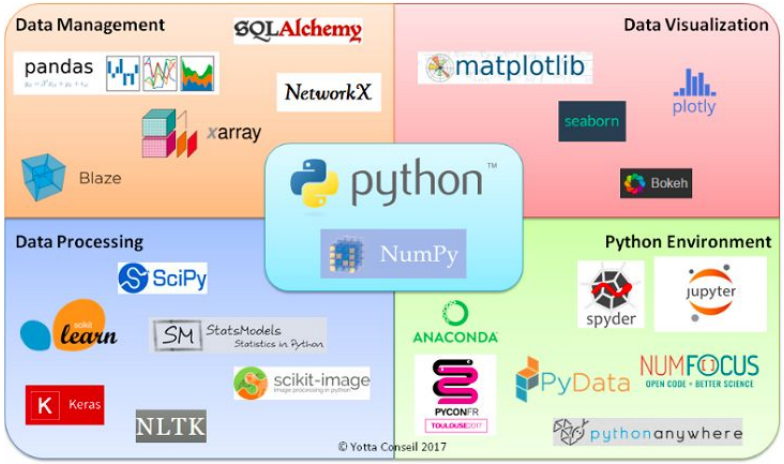
- Python adoption is constantly growing. Even if there are programming languages which might be “better” (e.g. Rust, Go), Python is still one of the most used ones.



What is Python

➔ *The language of the data science and A.I.*

- Python is the “de facto” standard language for data science and Artificial Intelligence, with an extensive ecosystem of numerical and data analysis libraries.



The Python data science ecosystem (source: Yotta Conseil)

What is Python

➔ *A nearly pseudocode language*

- Pseudocode is a form of abstract coding which allows to focus on the goal instead of the implementation details. There are no standards for the pseudocode, it is up to you.

```
given a list of numbers containing 13,4,51,8

for each element in the list:
    if the element is lower than 5:
        print the element
```

What is Python

➔ *A nearly pseudocode language*

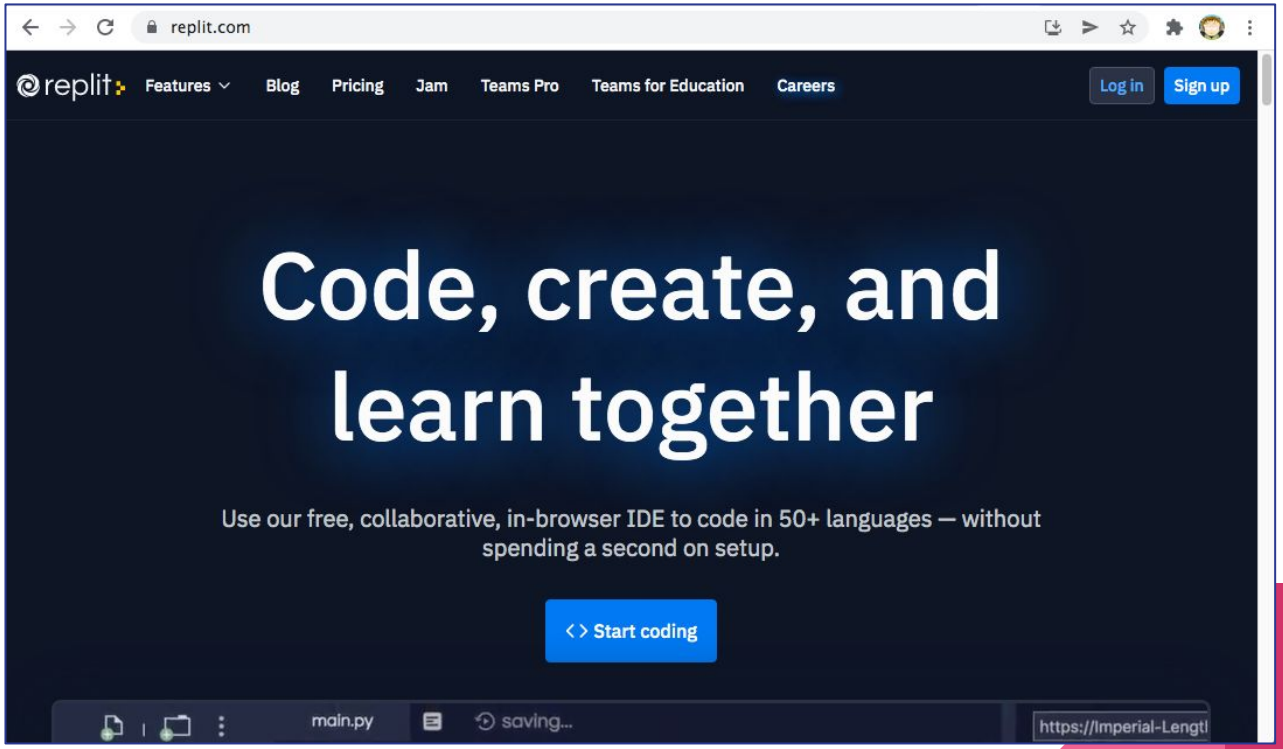
- Python allows to write code which is close to pseudocode. This allows to focus on its logic instead of getting lost in implementation details, and greatly improves readability.

```
number_list = [13,4,51,8]

for element in number_list:
    if element < 5:
        print(element)
```

Tools and “hello world”

→ *Repl.it*



Tools and “hello world”

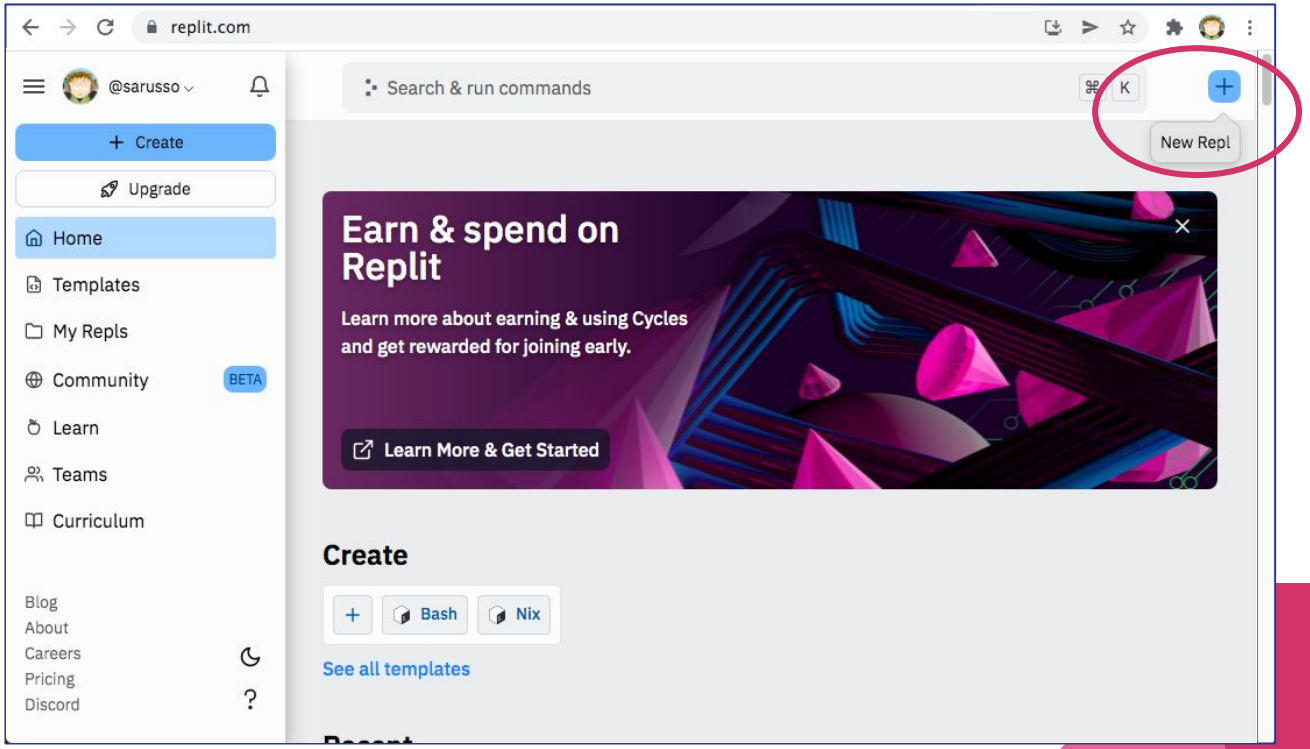
➔ *Repl.it*

- Repl is a browser-based mini IDE (Integrated Development Environment)
- Repl (actually REPL) stands for Read, Evaluate and Print Loop.
- Provides a code editor, a shell, a console, and even versioning integration
- Every “Repl” is a micro-computer in the Cloud based on Linux
- Free to use for public “Repls”

→ create an account now if you haven’t already

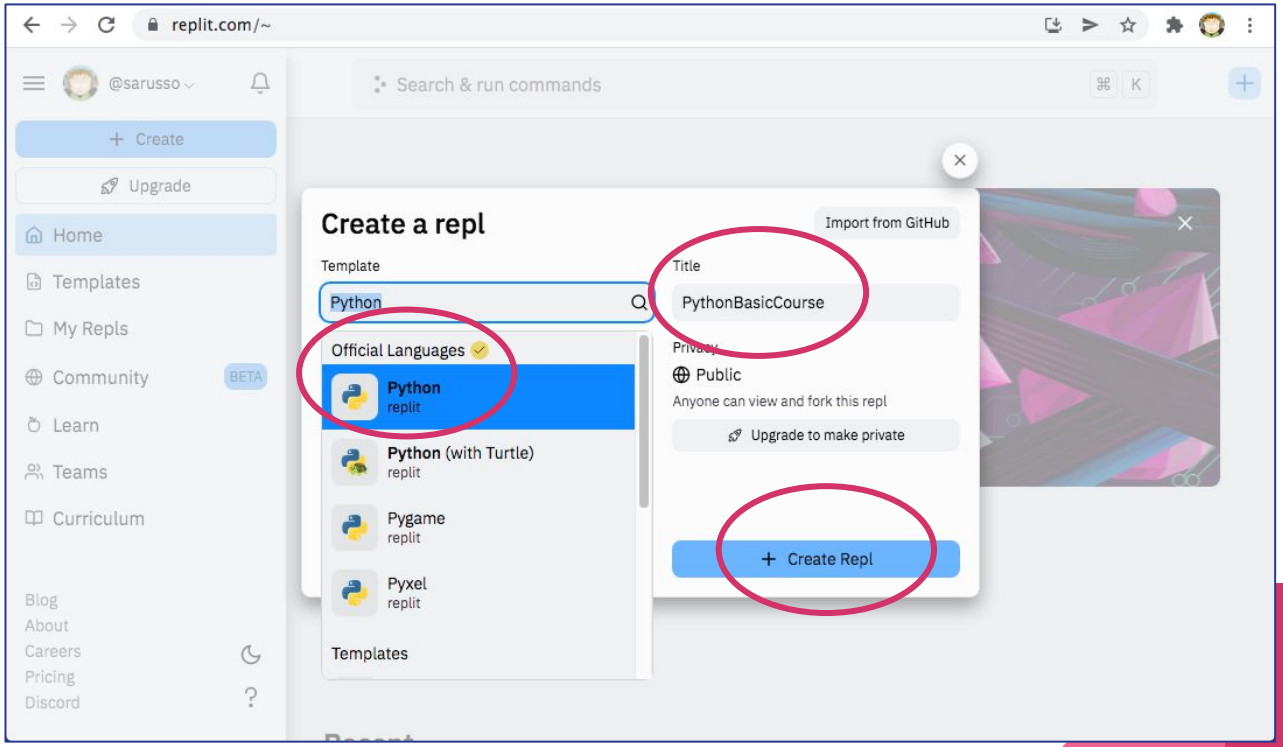
Tools and “hello world”

→ *Repl.it*



Tools and “hello world”

→ *Repl.it*



Tools and “hello world”

→ *Repl.it*

replit.com/@sarusso/PythonBasicCourse#main.py

sarusso / PythonBasicCourse

Run

Invite

Files

main.py

README.md

Packager files

poetry...

pyproj...

CPU RAM Stor...

main.py

```
1  Not sure what to do? Run some examples
   (start typing to dismiss)
```

Console Shell Markdown

Preview of README.md

Python on Replit

This is a template to get you started with Python on Replit. It's ready to go so you can just hit run and start coding!

Running the repl

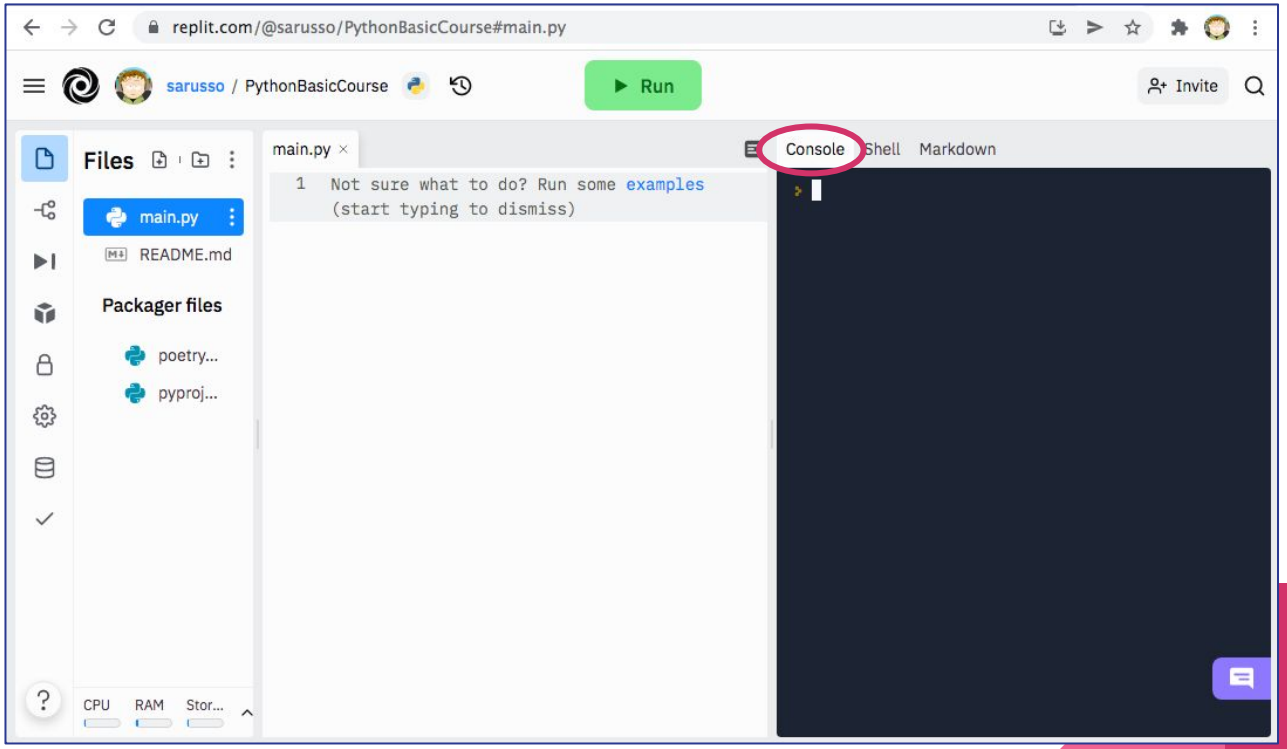
Simply hit run! You can edit the run command from the `.replit` file.

Installing packages

To add packages to your repl, you can just import directly in the file you want to use the

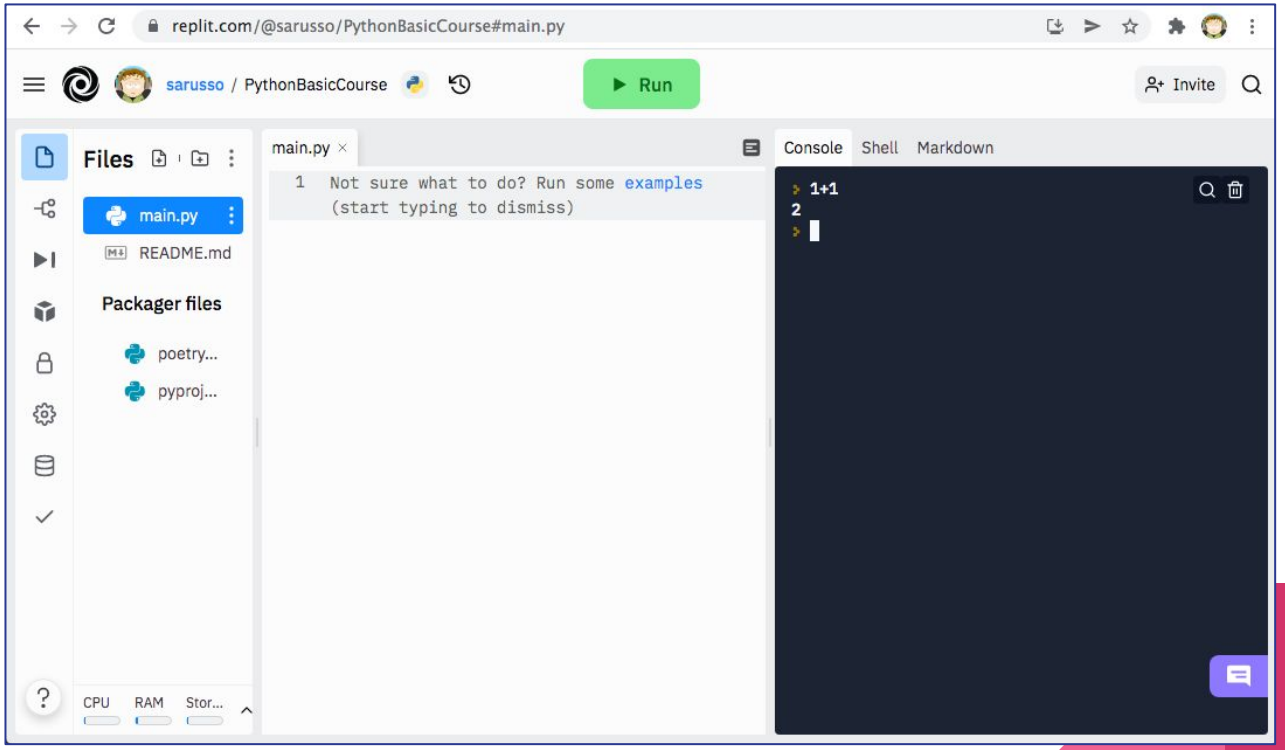
Tools and “hello world”

→ *Repl.it*



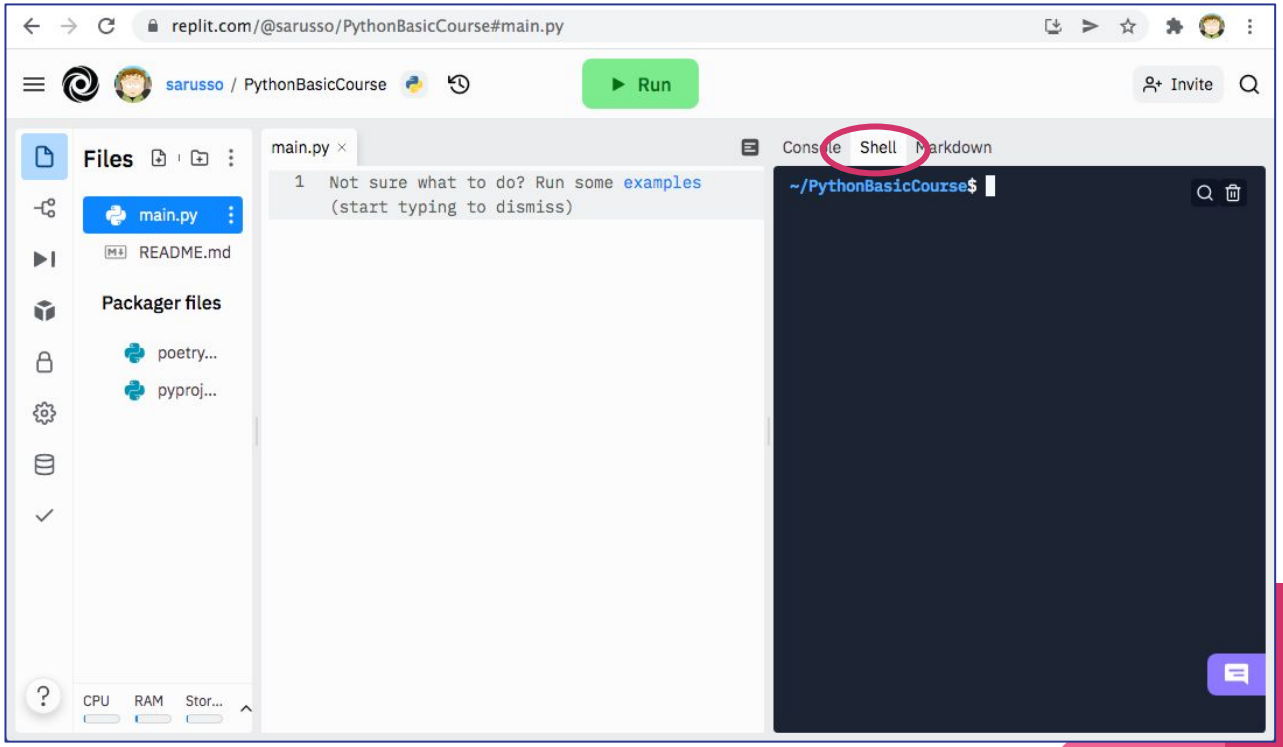
Tools and “hello world”

→ *Repl.it*



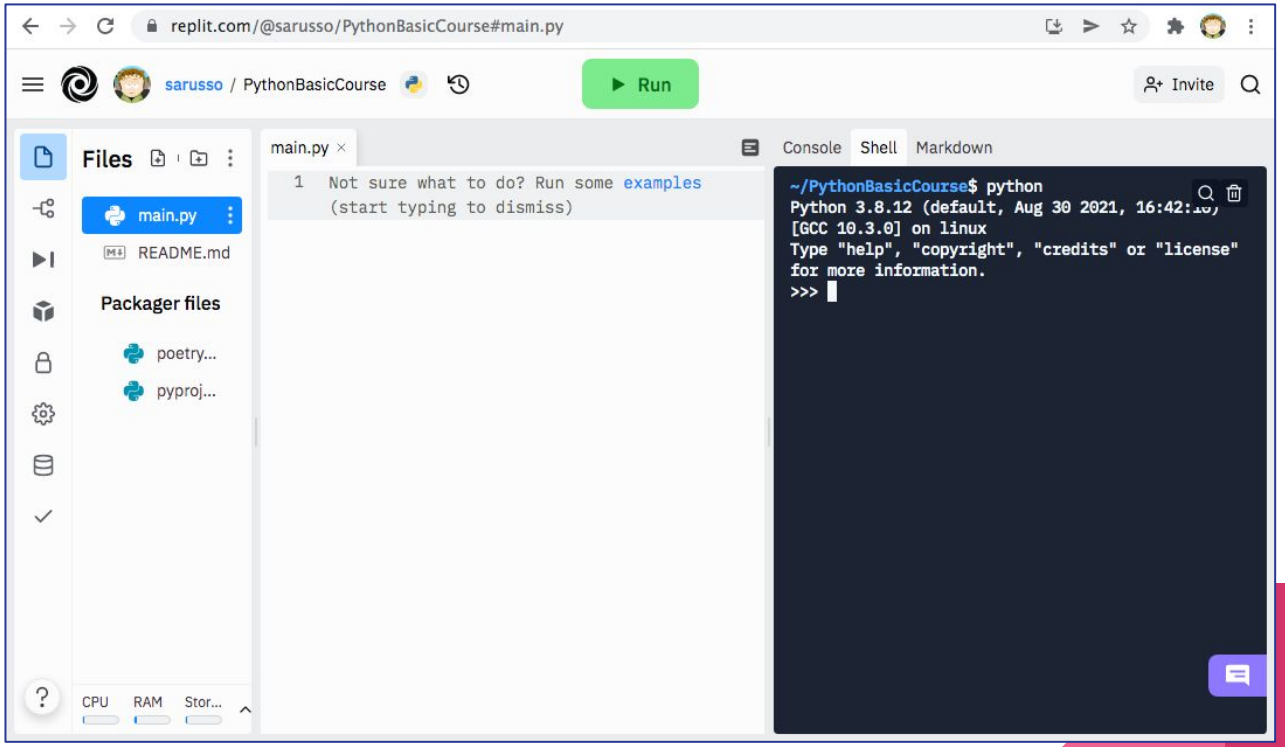
Tools and “hello world”

→ *Repl.it*



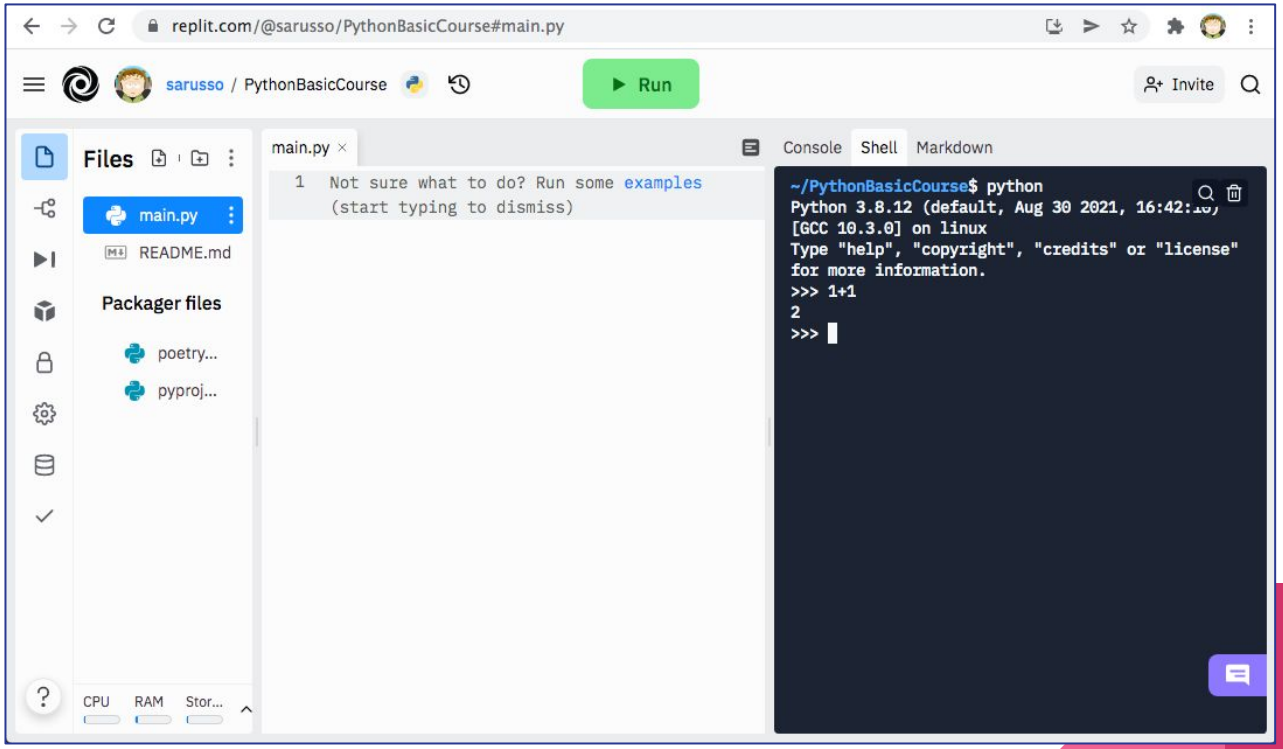
Tools and “hello world”

→ *Repl.it*



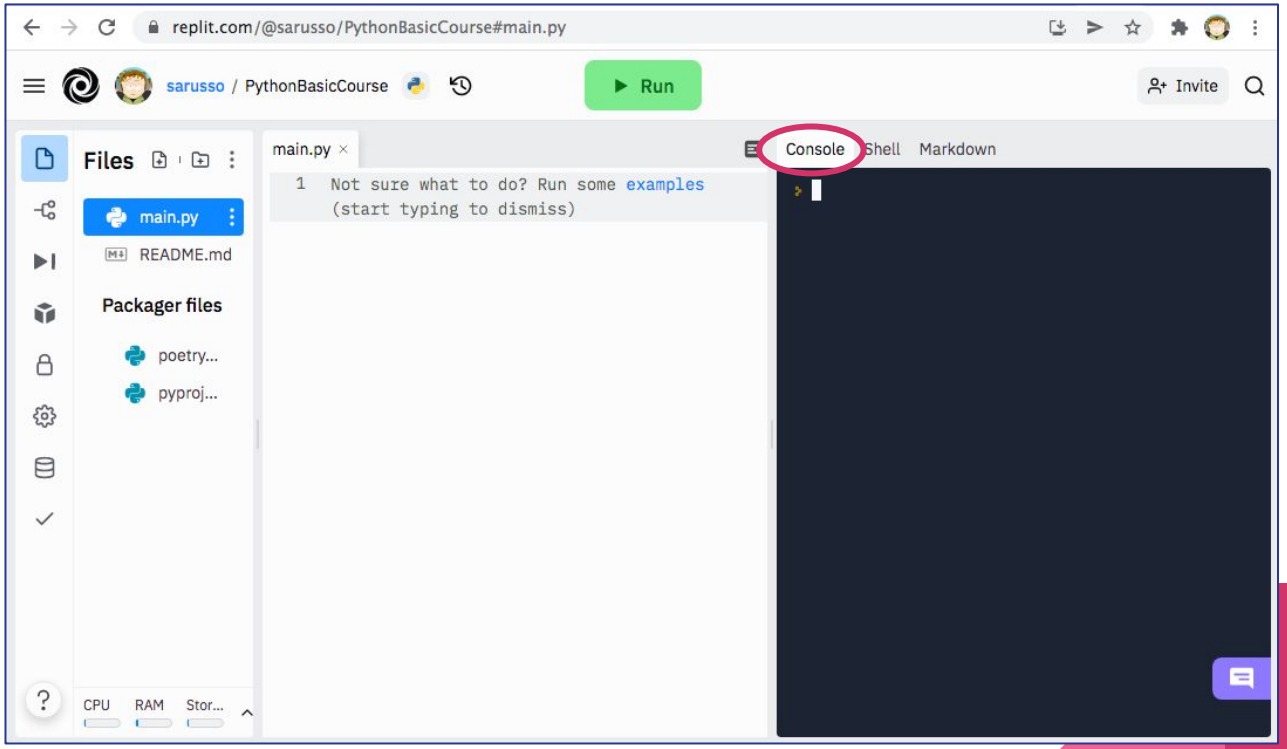
Tools and “hello world”

→ *Repl.it*



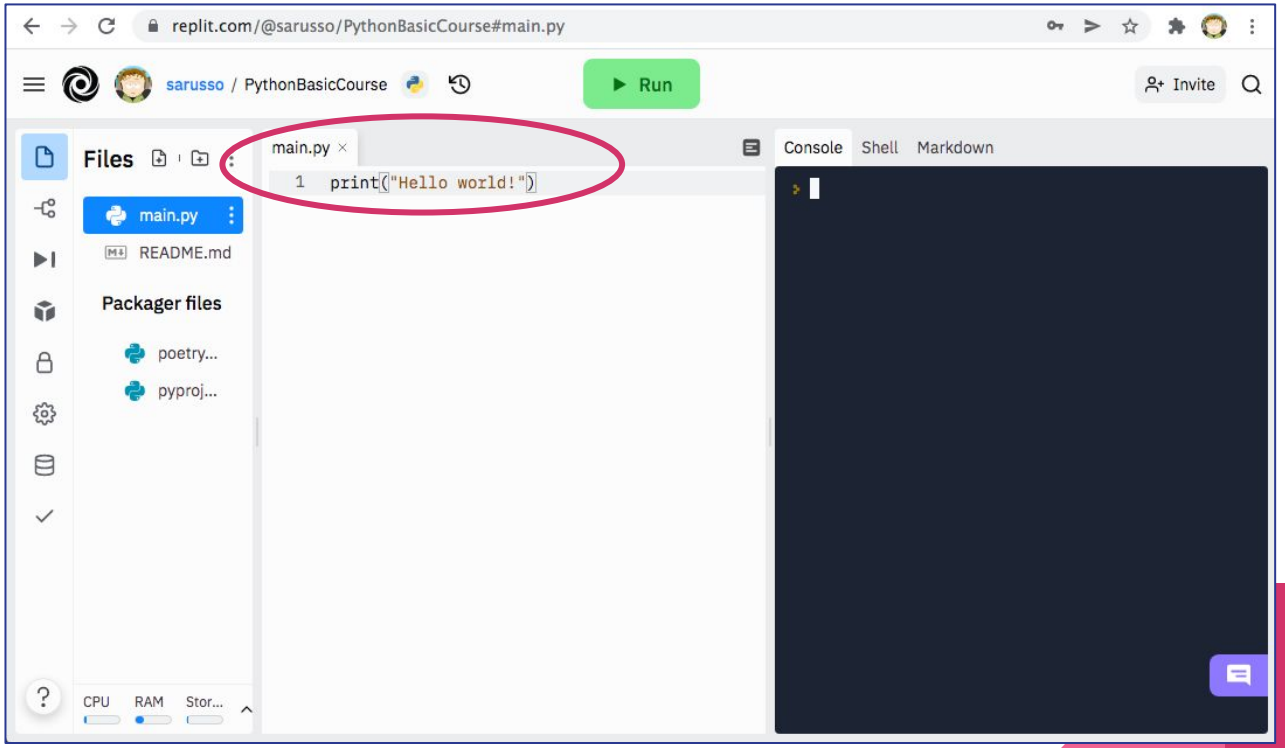
Tools and “hello world”

→ *Repl.it*



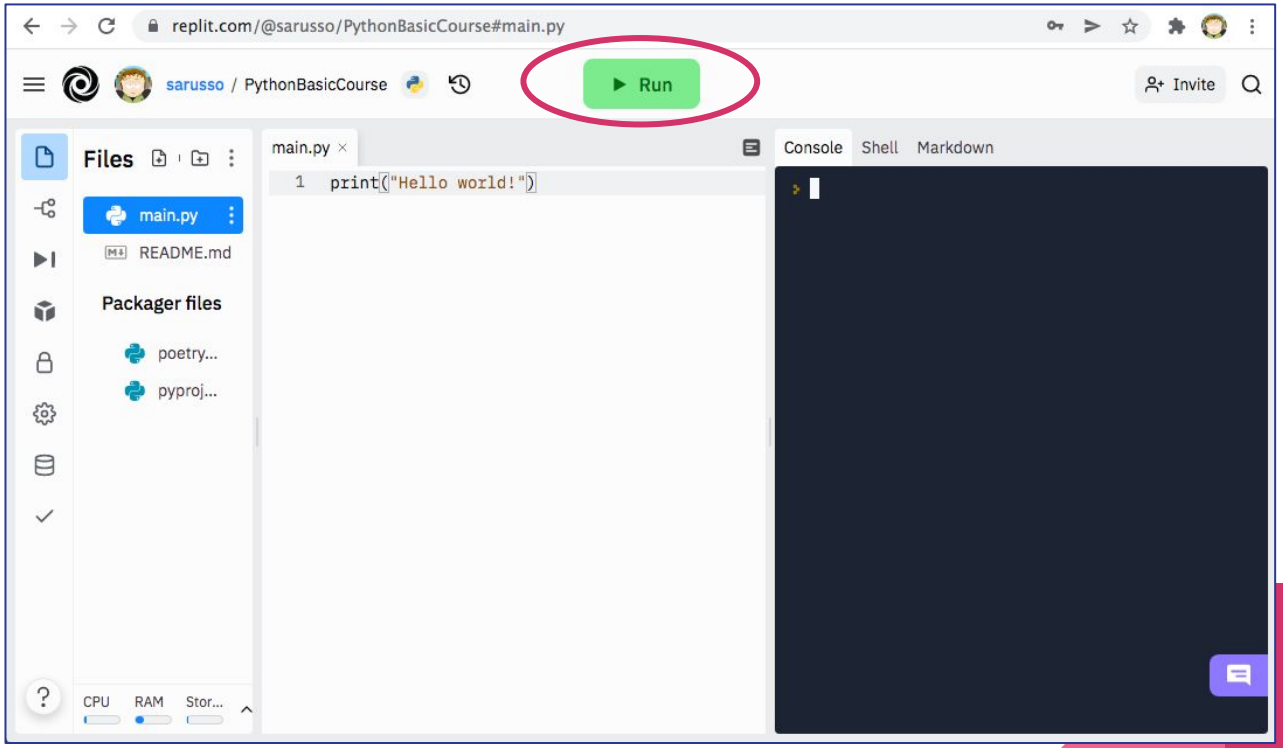
Tools and “hello world”

→ *Repl.it*



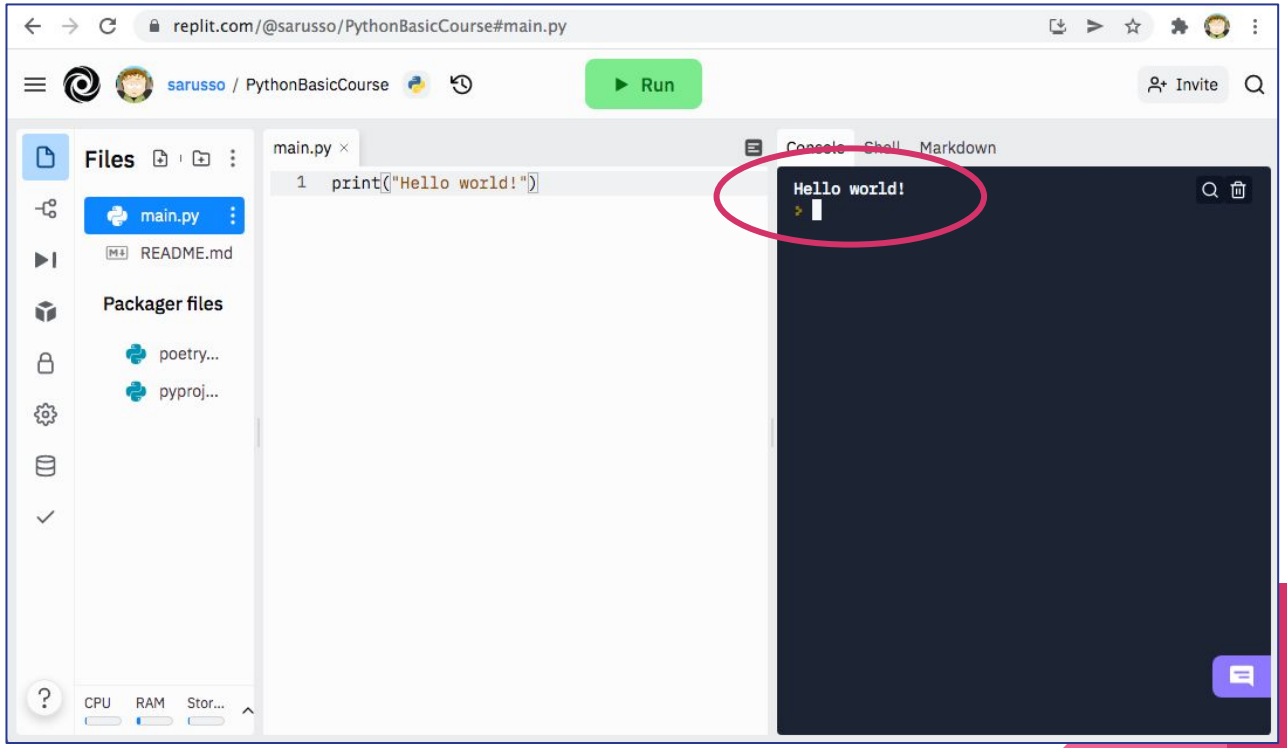
Tools and “hello world”

→ *Repl.it*



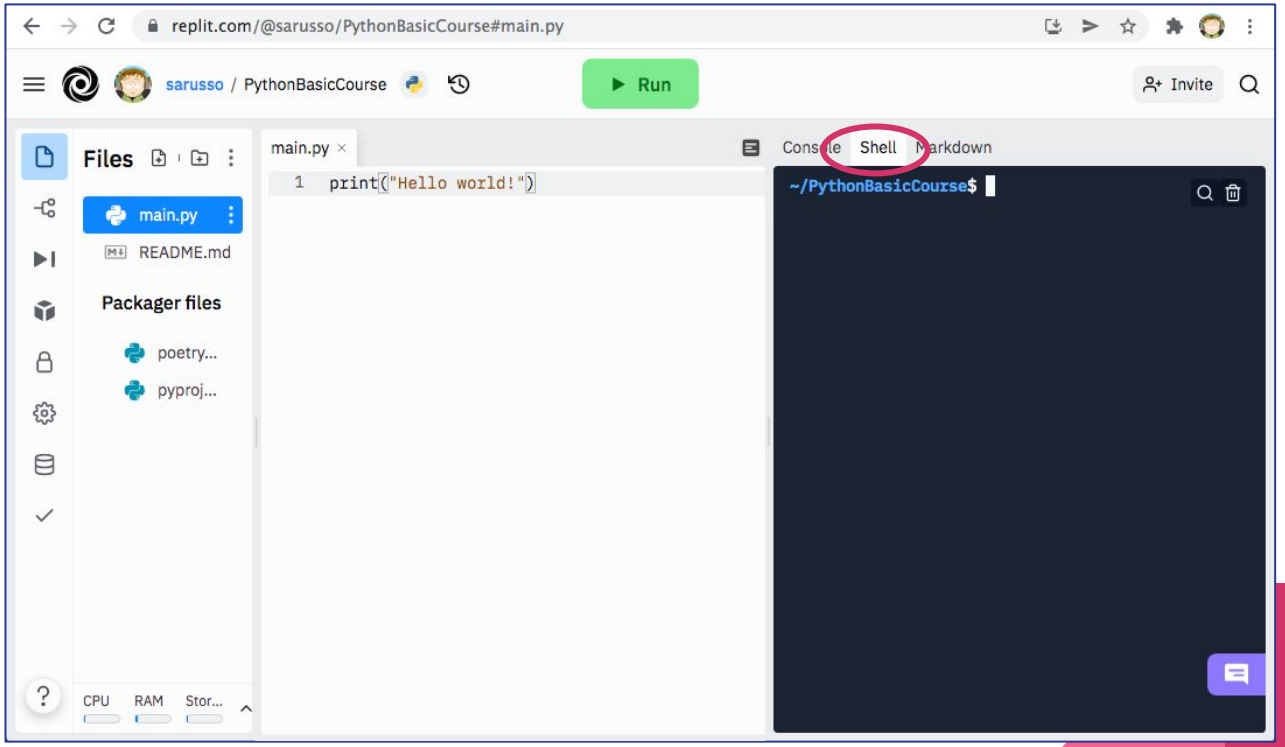
Tools and “hello world”

→ *Repl.it*



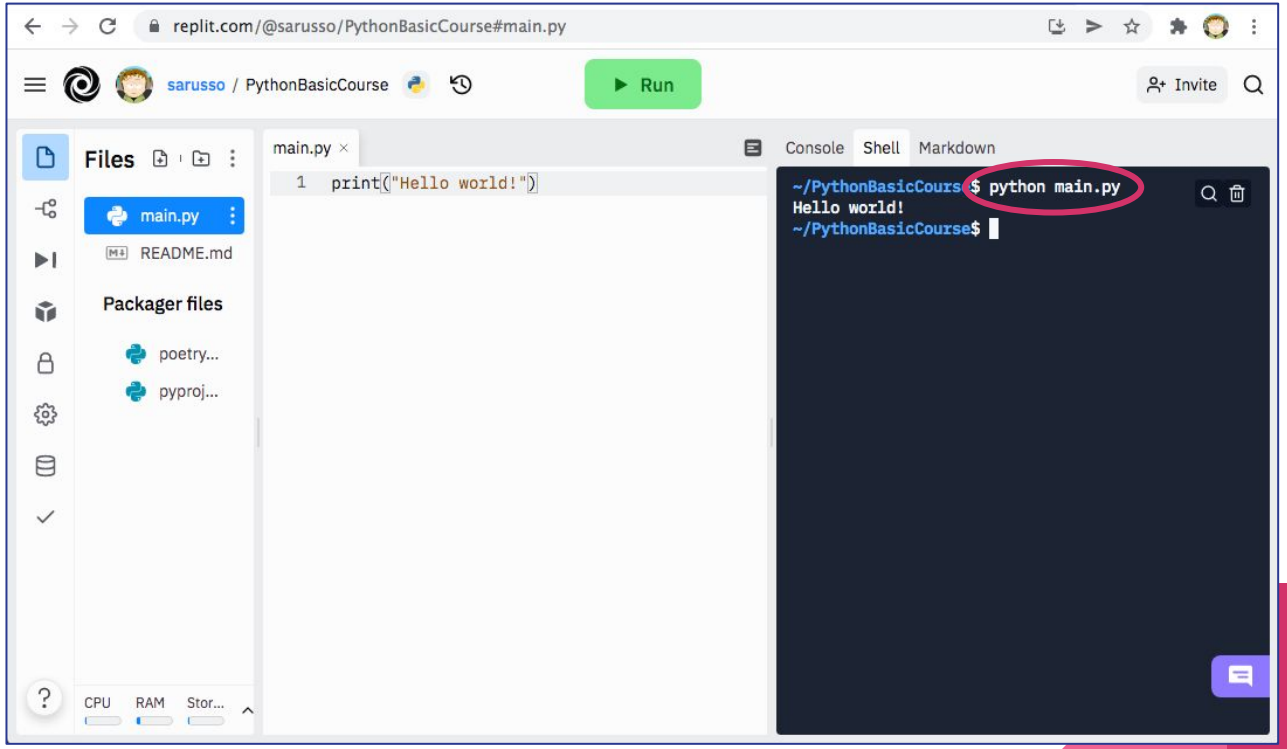
Tools and “hello world”

→ *Repl.it*



Tools and “hello world”

→ *Repl.it*



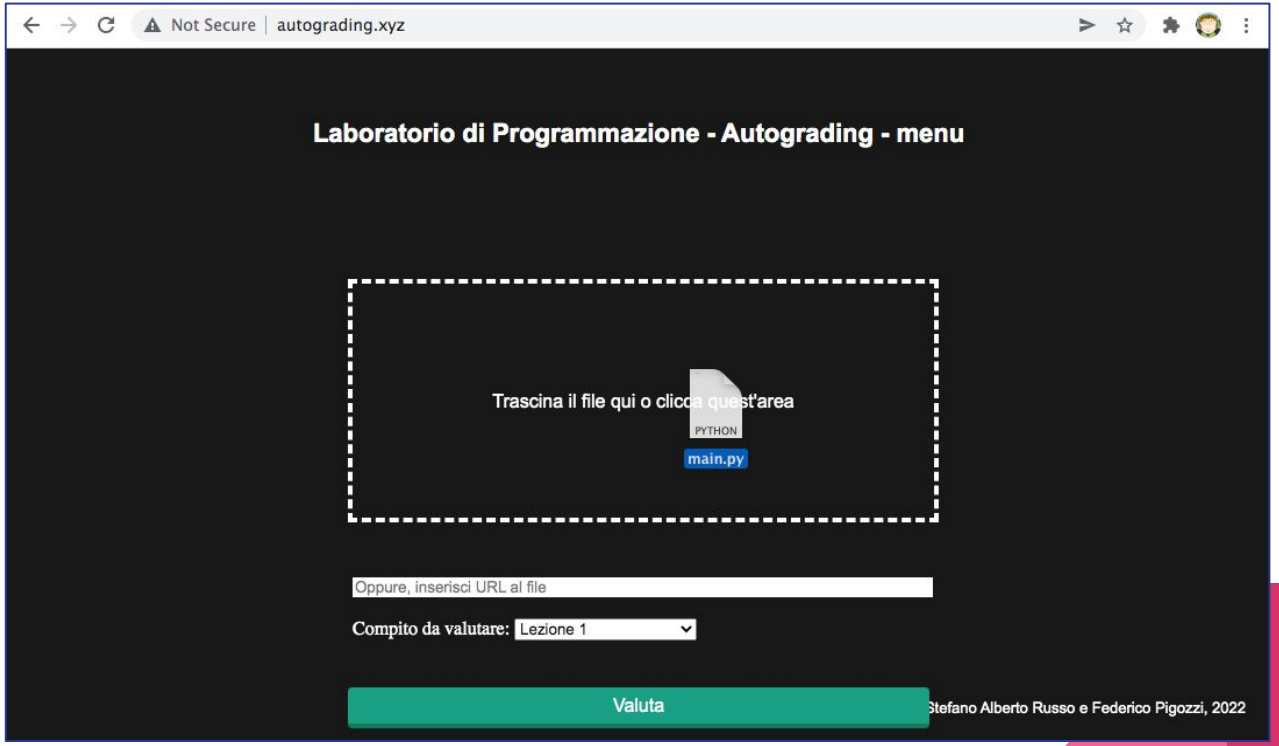
Tools and “hello world”

➔ *Autograding.xyz*

- A simple web application to evaluate your code
- No signup required
- Requires you to download from Repl.it the file you are working on and then upload it on Autograding.xyz to get the score
- You will use it for evaluating your exercises and see how you did
- Try it now with the “Hello world!” (beware capital letters)

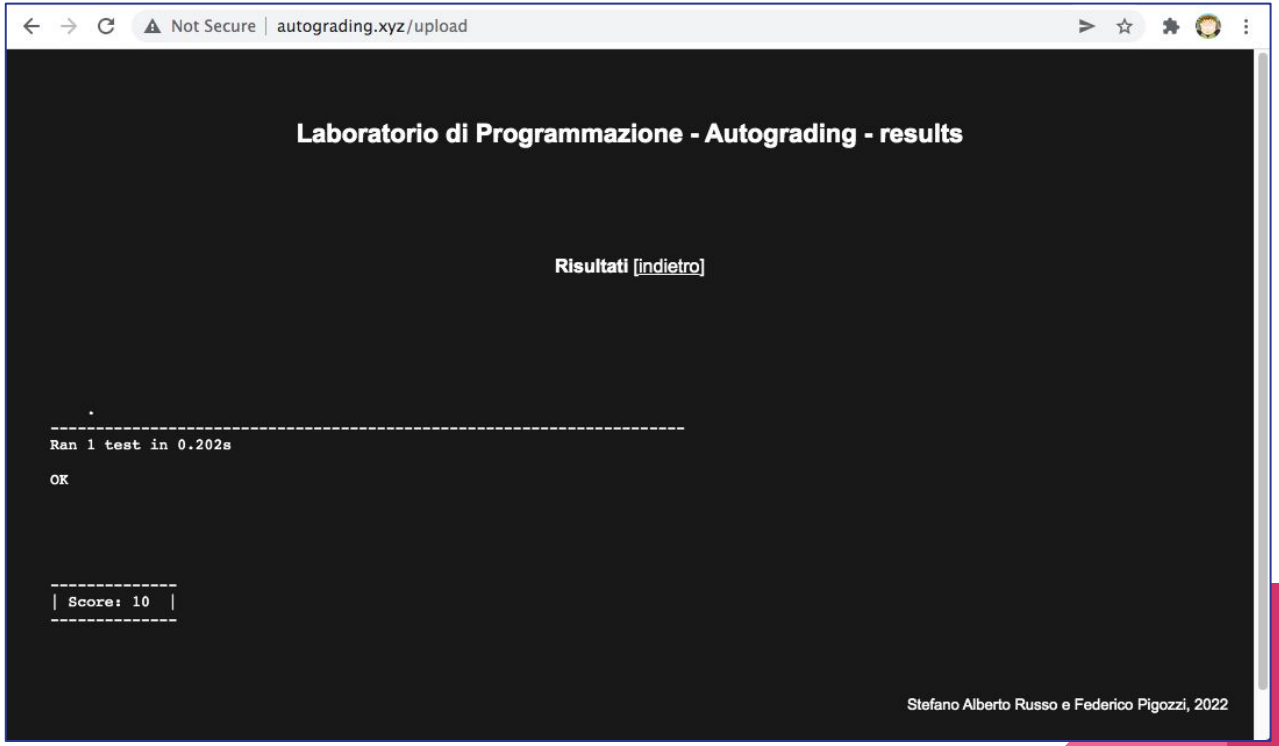
Tools and “hello world”

→ *Autograding.xyz*



Tools and “hello world”

→ *Autograding.xyz*



Basic syntax and data types

➔ *Let's start*

- We will now start digging into the basic syntax and data types of Python
- It is assumed some familiarity with imperative programming
  - *variables*
  - *if-then statements*
  - *for and while loops*
  - *etc.*
- However, we start taking nearly no other assumptions on your knowledge
- A good support tutorial is available here, if you get lost: https://www.w3schools.com/python/default.asp

Basic syntax and data types

➔ *The print function*

- The *print()* function allows to “print” something on the console or the shell. We saw it in the “Hello world” example, but I can also print variables:

```
my_var = 1
print(my_var)
```

- To print a mix of text and variables, I can use the *format()* function:

```
my_var = 1
print('My variable: {}'.format(my_var))
```

Basic syntax and data types

➔ *Assignments*

- In Python, variables are assigned with the equal sign:

```
my_var = 1          # Example of an integer type variable
my_var = 1.1        # Example of a floating point type variable
my_var = 'ciao'     # Example of a string type variable
my_var = True       # Example of a boolean type variable
my_var = None       # Example of an "undefined" variable
```

Basic syntax and data types

➔ *Assignments*

- In Python, variables are assigned with the equal sign:

```
my_var = 1          # Example of an integer type variable
my_var = 1.1        # Example of a floating point type variable
my_var = 'ciao'     # Example of a string type variable
my_var = True       # Example of a boolean type variable
my_var = None       # Example of an "undefined" variable
```

Comments are inserted with the “hash” character. Everything following an hash is treated as a comment.

Basic syntax and data types

→ *Types*

- Python does not require to explicitly set the variable type.

```
my_var = 1          # Example of an integer type variable
my_var = 1.1        # Example of a floating point type variable
my_var = 'ciao'     # Example of a string type variable
my_var = True       # Example of a boolean type variable
my_var = None       # Example of an "undefined" variable
```

→ This feature is called “dynamic typing”

Basic syntax and data types

➔ *Types*

- The philosophy of Python with respect to data types follows the “Duck typing” paradigm: *if it walks like a duck and it quacks like a duck, then it must be a duck*.

```
my_var = 1                # Integer
my_other_var = 1.1        # Float
my_var + my_other_var     # Computes 2.1
```

Basic syntax and data types

➔ *Types*

- Python supports other two more advanced classes of data types:

```
my_list = [1,2,'ciao']       # List (array)
my_tuple = (1,2,7.28,None)   # Tuple, unchangeable
```

```
my_dict = {'name': 'John', 'age': 43} # Dictionary (key-value)
```

→ Dictionaries are **never** ordered!

(unless you use a special OrderedDict type)

Basic syntax and data types

→ *Types*

- Accessing array-like data types (lists and tuples):

```
my_list = [1,2,'ciao']       # List (array)
my_tuple = (1,2,7.28,None)   # Tuple, unchangeable
```

```
my_list[0]         # Returns the element in position zero
my_list.pop()      # Removes and return the last list element
my_list.append(8)  # Adds an element at the end of the list
```

Basic syntax and data types

→ *Types*

- Accessing array-like data types (dictionaries):

```
my_dict = {'name': 'John', 'age': 43} # Dictionary (key-value)
```

```
my_dict['name']          # Returns the value of the name key (John)
my_dict['age'] = 56      # Changes the value of the "age" key to 56
my_dict['role'] = 'PM'   # Creates a new key "role" with value "PM"
```

Basic syntax and data types

→ *Types*

- Nested types example: list of dictionaries

```
persons = [ {'name': 'John', 'age': 43},
            {'name': 'Zoe', 'age': 31},
            {'name': 'Steve', 'age': 65} ]
```

```
persons[1]          # Returns {'name': 'Zoe', 'age': 31}
persons[1]['name']  # Returns 'Zoe'
```

Basic syntax and data types

→ *Operators*

- Python supports all the standard comparison operators:

Operator	Name	Example
==	Equal	x == y
!=	Not equal	x != y
>	Greater than	x > y
<	Less than	x < y
>=	Greater than or equal to	x >= y
<=	Less than or equal to	x <= y

Basic syntax and data types

→ *Operators*

- Python supports all the standard “numerical” operators as well:

Operator	Name	Example
+	Addition	x + y
-	Subtraction	x - y
*	Multiplication	x * y
/	Division	x / y
%	Modulus	x % y
**	Exponentiation	x ** y

Basic syntax and data types

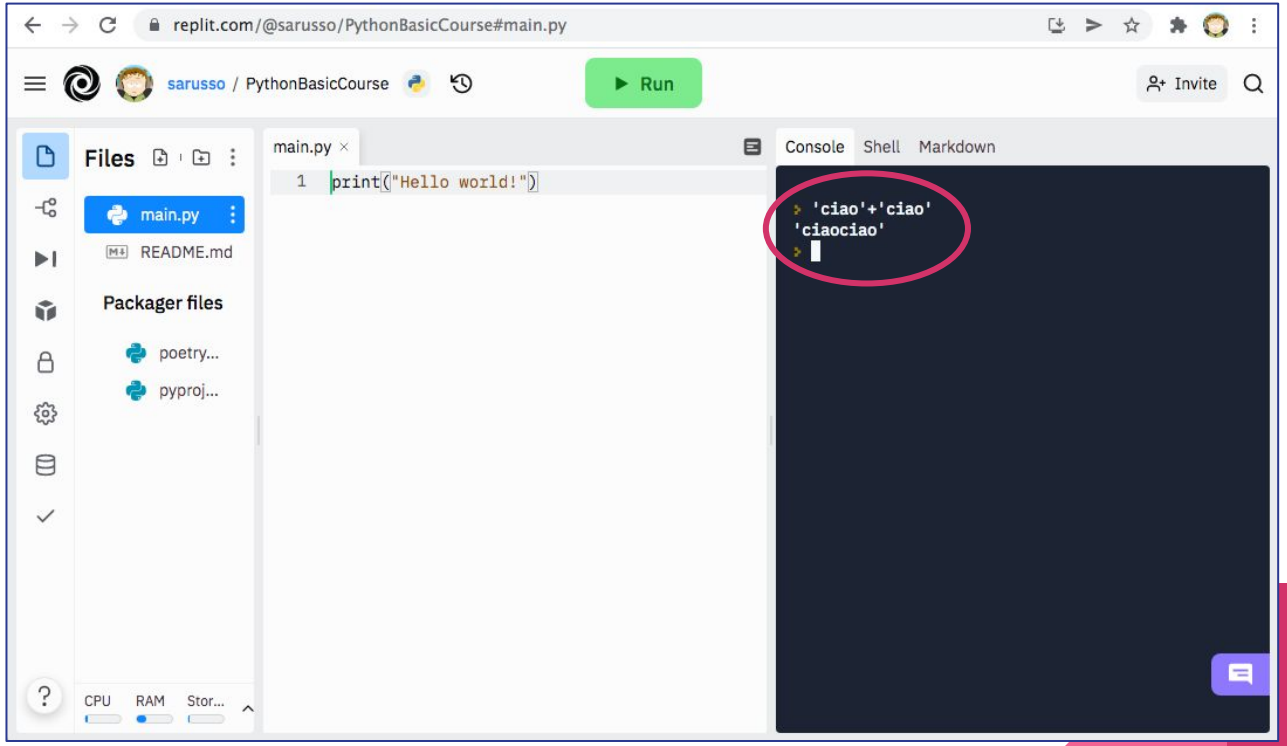
➔ *Operators*

- But keep in mind that these are extended to work with much many types

→ example: can I sum two strings?

Basic syntax and data types

→ *Operators*



Basic syntax and data types

→ *Operators*

- Also the classic *and*, *or* and *not* logical operators are supported:

Operator	Description	Example
and	Returns True if both statements are true	x < 5 and x < 10
or	Returns True if one of the statements is true	x < 5 or x < 4
not	Reverse the result, returns False if the result is true	not(x < 5 and x < 10)

Basic syntax and data types

→ *Operators*

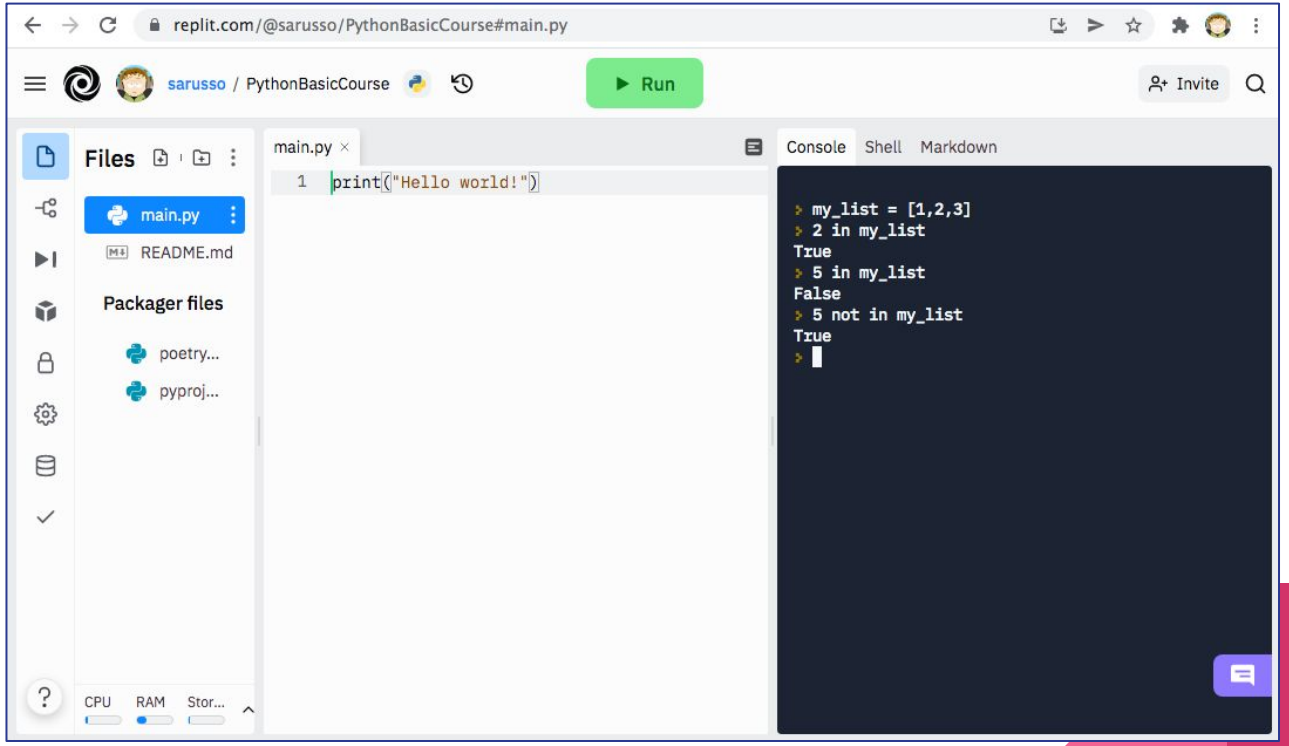
- Python provides other interesting operators when it comes to array-like types:

Operator	Description	Example
in	Returns True if a sequence with the specified value is present in the object	x in y
not in	Returns True if a sequence with the specified value is not present in the object	x not in y

Note: for the dictionaries, the inclusion (in) check is done on the keys

Basic syntax and data types

→ *Operators*



Basic syntax and data types

➔ *Conditional blocks*

- Conditional blocks are handled in Python with *indentation*:

```
if (my_var > your_var):
    print("My var is bigger than yours")

    if (my_var - your_var) <= 1:
        print("...but not so much")
```

Basic syntax and data types

➔ *Conditional blocks*

- Conditional blocks are handled in Python with *indentation*:

```
if (my_var > your_var):
    print("My var is bigger than yours")

    if (my_var - your_var) <= 1:
        print("...but not so much")
```

4 spaces

8 spaces
(4+4)

Basic syntax and data types

➔ *Conditional blocks*

- Conditional blocks allow for more conditions with the “elif” statement:

```
if (my_var > your_var):
    print("My var is bigger than yours")
    if (my_var-your_var) <= 1:
        print("...but not so much")
    elif (my_var-your_var) <= 5:
        print("...quite a bit")
    else:
        print("...a lot")
```

Basic syntax and data types

➔ *Loops*

- Python support the classic for and while loops:

```
for i in range(10):
    print(i) # Prints 0 1 2 3 ... 9
```

```
i = 0
while i < 10:
    print(i) # Prints 0 1 2 3 ... 9
    i = i+1
```

Basic syntax and data types

➔ *Loops*

- However, it make things much easier when it comes to iterate:

```
my_list = [1,2,3]
for item in my_list:
    print(item)
```

vs.

```
my_list = [1,2,3]
for i in range(len(my_list)):
    print(my_list[i])
```

- Using the style on the left means to be *"pythonic"*.

Basic syntax and data types

➔ *Loops*

- The “for” loop supports any *iterable* data type: this is the duck typing concept.

```
my_dict = {'a':1,'b':2}
for key in my_dict:
    print(key)
```

```
my_dict = 'ciao'
for char in my_string:
    print(char)
```

Basic syntax and data types

➔ *Loops*

- Some iterations are made easier by some helper functions:

```
my_list = [1,2,3]
for i, item in enumerate(my_list):
    print('Position #{}: element "{}"'.format(i, item))
```

```
my_dict = {'a':1,'b':2}
for key, value in my_dict.items():
    print('Key "{}": value "{}"'.format(key, value))
```

Basic syntax and data types

➔ *Loops*

- Some iterations are made easier by some helper functions:

```
my_list = [1,2,3]
for (i, item) in enumerate(my_list):
    print('Position #{}: element "{}"'.format(i, item))
```

This is a tuple

```
my_dict = {'a':1,'b':2}
for (key, value) in my_dict.items():
    print('Key "{}": value "{}"'.format(key, value))
```

This is a tuple

End of part I

➔ *Questions?*

Next: exercise 1

Exercise 1

Write a code that prints, for each month of the year, its number, its name and how many days it contains.

- The output format must be:

```
1: January, 31
2: February, 28
...
12: December, 31
```

- We assume a February of 28 days