

# **Introduction à Python\***

**Tristan Crolard**

Laboratoire CEDRIC  
Equipe « Systèmes Sûrs »

**tristan.crolard@cnam.fr**

**cedric.cnam.fr/sys/crolard**

---

\*. Ces supports sont adaptés de *Python for Computational Science* (2024)

# The Python prompt

- ▶ Python prompt waits for input:  
>>>
- ▶ Read, Evaluate, Print, Loop → REPL

# Hello World program

Standard greeting:

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

Entered interactively in Python prompt:

```
>>> print("Hello World")
```

```
Hello World
```

```
>>>
```

# A calculator

```
>>> 2 + 3
```

5

```
>>> 42 - 15.3
```

26.7

```
>>> 100 * 11
```

1100

```
>>> 2400 / 20
```

120.0

```
>>> 2 ** 3          # 2 to the power of 3
```

8

```
>>> 9 ** 0.5        # sqrt of 9
```

3.0

# Create variables through initialization

```
>>> a = 10
```

```
>>> b = 20
```

```
>>> a
```

10

```
>>> b
```

20

```
>>> a + b
```

30

```
>>> ab2 = (a + b) / 2
```

```
>>> ab2
```

15.0

# Basic data types – command type

```
>>> a = 1
```

```
>>> type(a)
```

```
<class 'int'>
```

```
>>> b = 1.0
```

```
>>> type(b)
```

```
<class 'float'>
```

```
>>> c = "1.0"
```

```
>>> type(c)
```

```
<class 'str'>
```

# Summary useful commands – introspection

- ▶ `print(x)` to display the object `x`
- ▶ `type(x)` to determine the type of object `x`
- ▶ `help(x)` to obtain the documentation string

## Example

```
>>> help("abs")
```

```
Help on built-in function abs in module builtins:
```

```
abs(x, /)
```

```
Return the absolute value of the argument.
```

# First use of functions

## Example 1

*type hints*      *type hint*  
    ↙      ↘              ↓

```
>>> def mysum(a: int, b: int) -> int:  
      return a + b
```

```
>>> # main program starts here
```

```
>>> print("The sum of 3 and 4 is", mysum(3, 4))
```

The sum of 3 and 4 is 7

**Note.** The Python runtime does not enforce [type annotations](#) (also called [type hints](#)). However, they are [strongly recommended](#) since they can be used by third party tools such as [type checkers](#), IDEs, compilers, etc.

# Functions can – and should – be documented

```
>>> def mysum(a: int, b: int) -> int:
    """Return the sum of parameters a and b."""
    return a + b
```

```
>>> # main program starts here
```

```
>>> print("The sum of 3 and 4 is", mysum(3, 4))
```

The sum of 3 and 4 is 7

Can now use the help function for our new function:

```
>>> help(mysum)
```

Help on function mysum:

```
mysum(a: int, b: int) -> int
```

```
    Return the sum of parameters a and b.
```

# Functions – terminology

```
>>> x = -1.5
```

```
>>> y = abs(x)
```

- ▶ `x` is the **actual argument** given to the function
- ▶ `y` is assigned to the **return value** (the result of the function's computation)
- ▶ Functions may expect zero, one or more arguments
- ▶ Not all functions seem to return a value.  
If no **return** keyword is used, the special object **None** is implicitly returned.

# Functions – example

```
>>> def plus42(n: int) -> int:
    """_Add_42_to_n_and_return_"""
    m = n + 42                # body of the
    return m                  # function
```

```
>>> a = 8
```

```
>>> b = plus42(a)
```

```
>>> b
```

50

After execution, b carries the value 50 (and a = 8).

**Note.** Variable n is a **parameter** (or **formal argument**) of function plus42, and variable a is the **argument** (or **actual parameter**).

# Functions – summary

- ▶ Functions provide (black boxes of) functionality: crucial building blocks that hide complexity
- ▶ interaction (input, output) through input arguments and return values (printing and returning values is not the same!)
- ▶ docstring, together with type hints, provides the specification (contract) of the function's input, output and behaviour
- ▶ a function should (normally) not modify input arguments (watch out for lists, dicts, more complex data structures as input arguments)

# Printing vs returning values

Given the following two function definitions:

```
>>> def print42():  
    print(42)
```

```
>>> def return42():  
    return 42
```

Let's try to explore the difference using the Python prompt:

```
>>> print42()
```

42

```
>>> return42()
```

42

Let's try again with temporary variables:

```
>>> b = return42()    # return 42, which is assigned to b
```

```
>>> print(b)         # and b is then printed
```

42

```
>>> a = print42()    # prints 42 to screen and return None
```

42

```
>>> print(a)
```

None

# Printing vs returning values – type hints

Add type hints to explicit the difference with return types:

```
>>> def print42() -> None:  
    print(42)
```

```
>>> def return42() -> int:  
    return 42
```

# The math module – import math

```
>>> import math
```

```
>>> math.sqrt(4)
```

```
2.0
```

```
>>> math.pi
```

```
3.141592653589793
```

```
>>> help(math.sqrt) # ask for help on sqrt
```

```
Help on built-in function sqrt in module math:
```

```
sqrt(x, /)
```

```
    Return the square root of x.
```

# Name spaces and modules – 3 good options

1. use the full name:

```
>>> import math
```

```
>>> print(math.pi)
```

```
3.141592653589793
```

2. use some abbreviation

```
>>> import math as m
```

```
>>> print(m.pi)
```

```
3.141592653589793
```

3. import all objects we need explicitly

```
>>> from math import sin, pi
```

```
>>> print(pi)
```

```
3.141592653589793
```

# Integer division

```
>>> 1 / 2
```

```
0.5
```

```
>>>
```

Dividing 2 integers returns a float:

```
>>> 4 / 2
```

```
2.0
```

```
>>> type(4 / 2)
```

```
<class 'float'>
```

If we want **integer division** (an operation that returns an integer), we use `//`:

```
>>> 1 // 2
```

```
0
```

# Python style guide: PEP8

<http://www.python.org/dev/peps/pep-0008>

- ▶ Python programs **must** follow Python syntax.
- ▶ Python programs **should** follow Python style guide, because
  - readability is key (debugging, documentation, team effort)
  - conventions improve effectiveness

# PEP8 Style guide

- ▶ Indentation: use 4 spaces (no tab character)
- ▶ One space around assignment operator (=) operator:  
c = 5 but not c=5.
- ▶ Spaces around arithmetic operators can vary:  
Both x = 3\*a + 4\*b and x=3\*a+4\*b are okay.
- ▶ No space before and after parentheses:  
x = sin(x) but not x = sin( x )
- ▶ A space after comma: range(5, 10) but not range(5,10).
- ▶ No whitespace at end of line
- ▶ No whitespace in empty line

- ▶ One or no empty line between statements within function
- ▶ Two empty lines between functions
- ▶ One import statement per line
- ▶ import first standard Python library (such as `math`), then third-party packages (`numpy`, `scipy`, ...), then our own modules
- ▶ no spaces around `=` when used in keyword arguments  
`"A_B_C".split(sep='_')` but not `"A_B_C".split(sep = '_')`

# PEP8 Style – summary

- ▶ Try to follow PEP8 guide, in particular for new code
- ▶ Use tools to help you, for instance:  
autopep8 program<sup>1</sup> available to format code from the command line.
- ▶ Tools/plugins are also available for most IDEs and editors, for instance: autopep8 extension for VS Code<sup>2</sup>

---

1. <https://pypi.org/project/autopep8>

2. <https://marketplace.visualstudio.com/items?itemName=ms-python.autopep8>

# Truth values

The python constants `True` and `False` are special built-in values:

```
>>> a = True
```

```
>>> print(a)
```

```
True
```

```
>>> type(a)
```

```
<class 'bool'>
```

```
>>> b = False
```

```
>>> print(b)
```

```
False
```

```
>>> type(b)
```

```
<class 'bool'>
```

We can operate with these two logical values using boolean logic, for example the logical and operation (`and`):

```
>>> True and True          # logical 'and' operation
```

```
True
```

```
>>> True and False
```

```
False
```

```
>>> False and True
```

```
False
```

```
>>> False and False
```

```
False
```

There is also logical or (**or**) and the negation (**not**):

```
>>> True or False
```

```
True
```

```
>>> not True
```

```
False
```

```
>>> not False
```

```
True
```

```
>>> True and not False
```

```
True
```

In computer code, we often need to evaluate some expression that is either true or false (sometimes called a “predicate”). For example:

```
>>> x = 30          # assign 30 to x
```

```
>>> x >= 30        # is x greater than or equal to 30
```

True

```
>>> x > 30         # is x greater than 30
```

False

```
>>> x == 30       # is x the same as 30
```

True

```
>>> not x == 42   # is x not the same as 42
```

True

```
>>> x != 42       # is x not the same as 42
```

True

# Conditional

The `if-else` command allows to branch the execution path depending on a condition. For example:

```
>>> x = 30                # assign 30 to x
>>> if x > 30:            # predicate: is x > 30
    print("Yes")          # if True, do this
else:
    print("No")           # if False, do this
```

No

**Note.** The `else` part of the command is optional.

The general structure of the `if-else` statement is

```
if A:  
    B  
else:  
    C
```

where `A` is the predicate.

- ▶ If `A` evaluates to `True`, then all commands `B` are carried out (and `C` is skipped).
- ▶ If `A` evaluates to `False`, then all commands `C` are carried out (and `B` is skipped).
- ▶ `if` and `else` are Python keywords.

`A` and `B` can each consist of multiple lines, and are grouped through indentation as usual in Python.

## if-else example

```
>>> def slength1(s: str) -> str:
    """Returns a string describing the length of the s"""
    ans: str
    if len(s) > 10:
        ans = "very long"
    else:
        ans = "normal"
    return ans
```

```
>>> print(slength1("Hello"))
```

normal

```
>>> print(slength1("HelloHello"))
```

normal

```
>>> print(slength1("Hello again"))
```

very long

# Printing basics

- ▶ the `print` function sends content to the “standard output” (usually the screen)
- ▶ `print()` prints an empty line:

```
>>> print()
```

- ▶ Given a single string argument, it is printed, followed by a new line character:

```
>>> print("Hello")
```

```
Hello
```

- ▶ Given another object (not a string), the print function will ask the object for its preferred way to be represented as a string:

```
>>> print(42)
```

42

- ▶ Given multiple objects separated by commas, they will be printed separated by a space character:

```
>>> print("dog", "cat", 42)
```

dog cat 42

- ▶ To suppress printing of a new line, use the end='' option:

```
>>> print("Dog", end='')  
      print("Cat")
```

DogCat

# Default argument values

## ► Motivation

- suppose we need to compute the area of rectangles and we know the side lengths  $a$  and  $b$ .
- Most of the time,  $b=1$  but sometimes  $b$  can take other values.

## ► Solution 1:

```
>>> def area(a: int, b: int) -> int:  
        return a * b
```

```
>>> print("The area is", area(3, 1))
```

The area is 3

```
>>> print("The area is", area(4, 1))
```

The area is 4

- We can make the function more user friendly by providing a default value for  $b$ . We then only have to specify  $b$  if it is different from this default value.

► **Solution 2** (with a default value for argument b):

```
>>> def area(a: int, b: int = 1) -> int:  
        return a * b
```

```
>>> print("The area is", area(3))
```

The area is 3

```
>>> print("The area is", area(4))
```

The area is 4

```
>>> print("The area is", area(4, 2))
```

The area is 8

- If a default value is defined, then this parameter (here b) is **optional** when the function is called.
- Default parameters have to be **at the end of the argument list** in the function definition.

# Default argument values – examples

You have met default arguments in use before, for example:

- ▶ the `print` function uses the following default values:  
`end = '\n'`  
`sep = '␣'`
- ▶ the `list.pop` method uses the following default value:  
`index = -1`

# Keyword argument values

- ▶ We can call functions with a “keyword” and a value (the “keyword” is the name of the **formal parameter** in the function definition).

- ▶ Here is an example:

```
>>> def f(a: int, b: int, c: int) -> None:  
        print("a=", a, ", b=", b, ", c=", c, sep = "")
```

```
>>> f(1, 2, 3)
```

```
a=1, b=2, c=3
```

```
>>> f(a=3, b=1, c=2)
```

```
a=3, b=1, c=2
```

```
>>> f(1, b=3, c=2)
```

```
a=1, b=3, c=2
```

- ▶ keyword arguments are often used in combination with default values.

# Common strategy for the print command

- ▶ Construct some string `s`, then print it using the `print` function:

```
>>> s = "I am the string to be printed"
```

```
>>> print(s)
```

I am the string to be printed

- ▶ The question is, how can we construct the string `s`?
  - using simple conversions (`str`) and concatenation (+)
  - using **advanced string formatting**

# String formatting overview

- ▶ 1991: % operator (Python 2) – **deprecated**
- ▶ 2006: `str.format()` “new style” or “advanced string formatting” (Python 3)

Further Reading:

- <http://docs.python.org/library/string.html#format-examples>
  - Python Enhancement Proposal 3101
- 
- ▶ **2016: f-strings (Python 3.6)**

# f-strings: formatted string literals

- ▶ Introduced in Python 3.6 (2016)
- ▶ Described in PEP-498  
<https://www.python.org/dev/peps/pep-0498>
- ▶ compatible with previous `str.format` syntax
- ▶ also called `Literal String Interpolation`

## f-strings – examples

```
>>> name = "Fred"
```

```
>>> print(f"He said his name is {name}.")
```

```
He said his name is Fred.
```

```
>>> value = 12.34567
```

```
>>> print(f"result: {value}")
```

```
result: 12.34567
```

We can evaluate Python expressions in the f-strings:

```
>>> import math
```

```
>>> print(f"The diagonal has length {math.sqrt(2)}.")
```

```
The diagonal has length 1.4142135623730951.
```