

Python: Introduction, variables and simple data types

Introduction to Computer Programming
Bachelor in Data Science

Roberto Guidi

roberto.guidi@supsi.ch

September 2021

SUPSI University of Applied Sciences and Arts
of Southern Switzerland

Programming languages

A **programming language** is a language **built to communicate instructions to a machine**. They can be used to create programs that control the computer behaviour.

Programming languages, in contrast to natural languages, **cannot be ambiguous** and must be **formalized in no uncertain terms**.

They are composed by:

- **Syntax**: set of rules that define what is valid to write.
- **Semantics**: meaning of what is valid to write.
- **Pragmatics**: ability to know which sentences (instructions) it is best to use depending on the context.

Refresher

Python main features

High level language

- basic data types, variables and operators,
- control flow and loop instructions (if, for, while, ...).
- functions,
- lists, dictionaries, tuples, objects, classes,
- exceptions management,
- modules and packages.

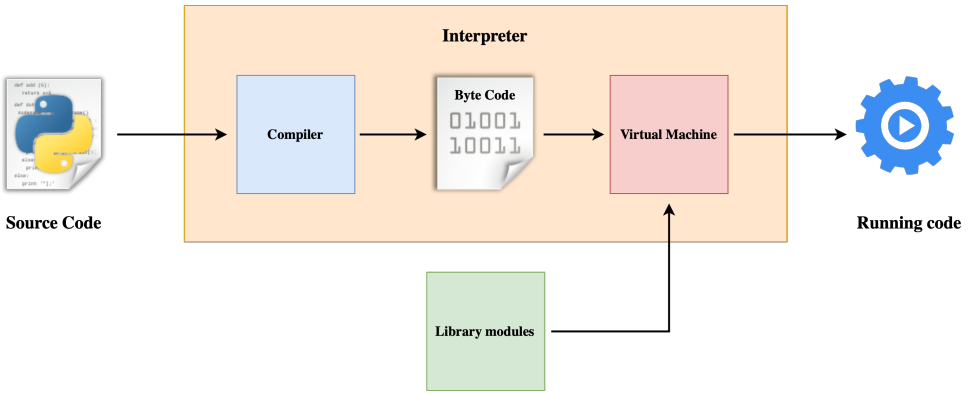
Object oriented

- classes, inheritance, abstraction, polymorphism

Input and output features

Refresher

Python Interpreter



Refresher

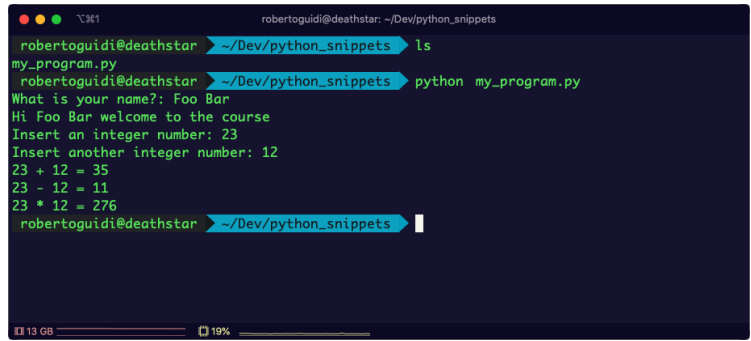
Example of a Python program

```
if __name__ == '__main__':  
    # request user input  
    name = input("What is your name?: ")  
    print("Hi " + name + " welcome to the course")  
  
    num1 = input("Insert an integer number: ")  
    num2 = input("Insert another integer number: ")  
    num1 = int(num1)  
    num2 = int(num2)  
  
    # do calculations  
    num_sum = num1 + num2  
    num_diff = num1 - num2  
    num_multi = num1 * num2  
  
    # print results  
    print(str(num1) + " + " + str(num2) + " = " + str(num_sum))  
    print(str(num1) + " - " + str(num2) + " = " + str(num_diff))  
    print(str(num1) + " * " + str(num2) + " = " + str(num_multi))
```

Program execution

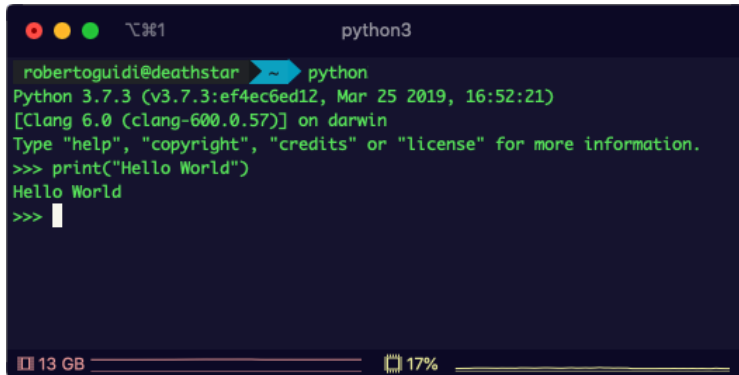
In order to **execute** a Python program, we must call the Python interpreter with our **source code file** (e.g. my_program.py)

```
python my_program.py
```



Program execution: the interactive prompt

Another way to execute your programs is to use the **python interactive prompt**, that can be started only by calling the interpreter with python.



```
robertoguidi@deathstar ~$ python
Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 16:52:21)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello World")
Hello World
>>> 
```

Program execution: the interactive prompt

After printing the version information, the interpreter prints the characters > > > and **waits for an input**.

It is then possible to insert **single statements and directly see the results** after pressing the **Enter** key.

We will use it often to experiment with the language and to run some simple examples.

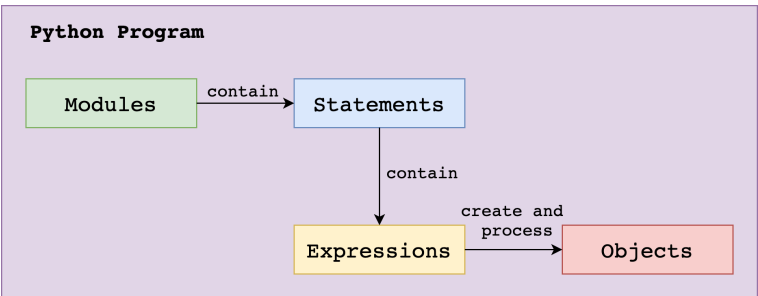
This tool is perfect because it allows us to follow this computer science rule:

If you have a doubt on how something works, you can always try it!

Structure of a Python program

Python programs consists of the following components:

- modules
- statements
- expressions
- objects



Python source code files

The source code for Python program is defined inside a file, with the **.py** extension.

For convention, file names in Python are **lowercase** and **without spaces**.

The character **_** is used instead of spaces. e.g. (calculator_program.py)

Inside a .py file we will typically find different kinds of instructions including:

- comments
- variable declarations
- function calls
- module imports
- class definitions
- ...

Statements

Statements (colloquially "instructions") are the units that are **executed** in a Python program.

Usually, each statement is a **complete line of code** that finishes at the **end of the line**.

In Python there are also ways to **make statements span on multiple lines** by using:

- the **** character
- implicit line continuation using **parentheses ()**, **brackets []** or **braces { }**

A statement in Python is also defined as **logical line**

Statements

Statements can be of different kinds, the most significant are:

- **Expression statements**: used to compute and write a value, or to call a procedure
- **Assignment** statements: instructions to bind identifiers (names) to values
- **Compound statements**: they contain other statements and they affect or control the execution of those statements. **Flow control statements** such as **if**, **while**, **for**, ... are an example.

Statements: example

```
# example of a program with various statement
if __name__ == '__main__':

    value = 0
    for j in range(0, 30):
        if j == 9:
            value += 1
            print("10")
        elif j == 19:
            value += 1
            print("20")
    print("Final value: " + str(value))
```

Blocks of code and indentation

Multiple statements are grouped in what we call **code blocks**.

In Python code blocks consists of **statements** that present all the **same indentation level**.

In other languages in contrast, it is common to use curly brackets {...}, to define block start and end.

Indentation is done in the source code by using either **whitespaces or tabs**.

For the moment, just keep in mind that **indentation is important**. We will see more later on.

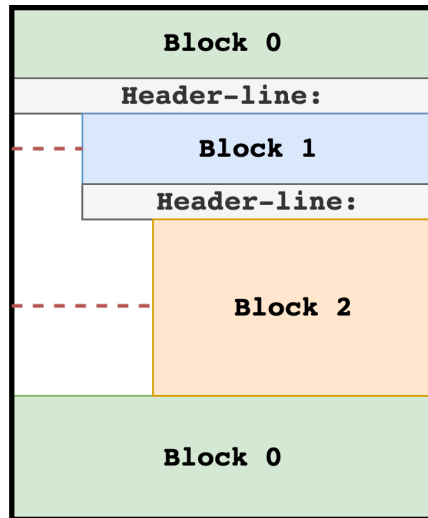
Blocks of code and indentation

Indentation level 0

Indentation level 1

Indentation level 2

Indentation level 0



Comments

In Python it is possible to use **comments** inside our programs.
Comments are special instructions that **the interpreter will ignore**, since they are **notes for the developer itself**.

It is very important, as your programs evolve and become more complicated, to **describe what you are doing using comments**. This will help the "future you" and also **the people that will work with you** to understand your approach.

Comments can be inserted using the special character **#**.

```
# comment
diameter = 20 # comment 2
```

Everything what is at the **right of the symbol # until the end of the line** is considered a comment and thus **ignored**.

Comments

If we need to write a lot of description text it is possible to have comments that spans to multiple lines.

To define **multi-line comments** there is no special symbol in Python.

Multi-line comments is achieved using the **#** symbol at the beginning of each comment line.

```
# Comment that spans
# from the first line to
# the third line
area = 40
```

The `__main__` execution context in Python

Although it is not strictly necessary in order to run a Python program, you should **always place all the code inside the following block.**

```
if __name__ == "__main__":  
    # some instructions
```

As we will see more in detail later, Python works with **modules** that can be **imported** by other modules.

Using the block above, we are telling the Python interpreter to **run our code only when we are running our file as a top-level program (in the `__main__` context).**

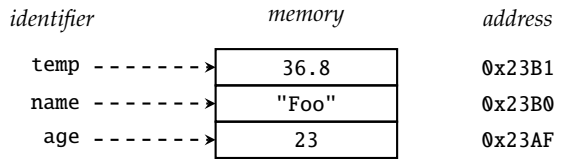
This will allow us to **safely use our file also as import** in other modules in the future.

Variables

A variable is an element of the program that, in different moments during the execution of the application, can **assume different values**. Each variable points to a certain memory address.

A variable can be defined as a couple composed by:

- an **identifier** (also known as symbolic name)
- a **memory address containing the value** of the variable



In a program, the variable identifier make possible the **read and/or modify, during the program execution, the associated value**.

Variables

In Python, in order to create a variable, it is necessary to **define the identifier (name)** that we want to use.

In contrast to other languages, it is not needed to specify the data type of the variable. Since it is a **dynamically typed** language, Python will figure out automatically the data type.

When declaring a new variable, it is also necessary to **initialize it with a value**.

```
message = "my message"  
print(message)
```

The variable is then accessible, using the identifier, **from the moment of the declaration until the end of the code block in which it is contained**.

It is possible to **redefine the value** held by a variable at **any time**.

Identifiers

The identifiers are the **names of variables, functions, classes, ...**

A few rules about identifiers:

- Identifiers can contain only letters, numbers and underscores. They can start with a letter or an underscore, but not with a number. You can call a variable **message_1** but **not 1_message**.
- Spaces are not allowed in identifiers. Use the character **_** instead. (**greeting_message**)
- Some keywords are reserved for particular purpose in Python. Absolutely avoid using those as identifiers.
- Identifiers must be short and descriptive. For instance, the variable name **year** is better than **y**

Remember that the code you write must be readable also for other people that work with you.

Identifiers

Python is case sensitive!

The identifier `My_number` is different than `MY_NUMBER` and also different than `my_number` and so on.

The Python naming convention is the following:

- For **variables and functions names**, the syntax to use is **snake_case**, with **words separated from underscores** (e.g. `variable_name`)
- For **class names**, the syntax to use is **PascalCase** (or **CapWords**), with each word's first letter **capitalized** and with all the words one after the other. (e.g. `MyClassName`)

Memory management: How do Python stores variables?

Python memory management, similar to what happens in other languages, works using two memory spaces.

The **stack** memory is where all the **variables references and method calls** are stored. The **heap** memory is where the **values of the objects** are stored.

When **assigning a value to a new variable**, Python creates the **variable in the stack** and **reference** it to a **value object stored in the heap**.

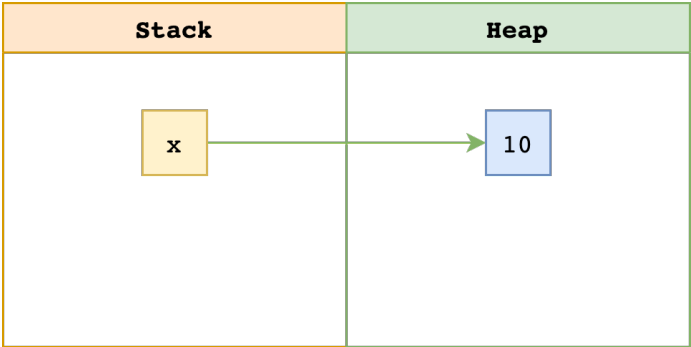
Let's see together how that works with an example.

Memory management: How do Python stores variables?

Python code:

```
x = 10
```

Memory:



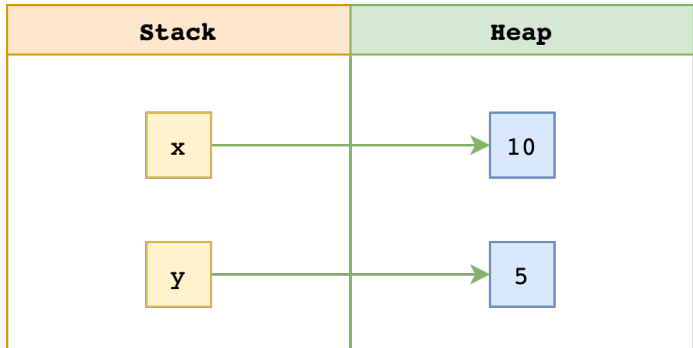
An object with value **10** is created in the heap, the variable with identifier **x** is referenced to the value and stored in the stack.

Memory management: How do Python stores variables?

Python code:

```
x = 10  
y = 5
```

Memory:



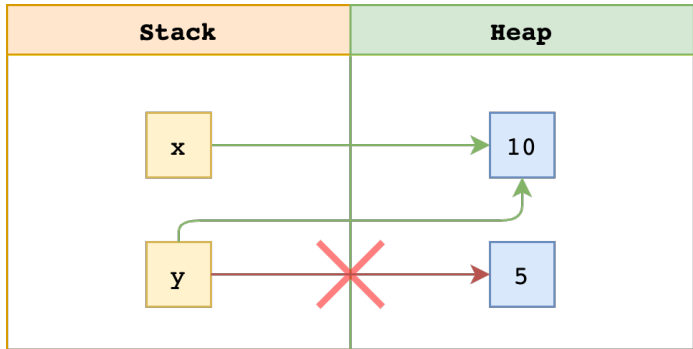
If a second variable with different value is created, these managed using stack and heap as seen before.

Memory management: How do Python stores variables?

Python code:

```
x = 10
y = 10
```

Memory:

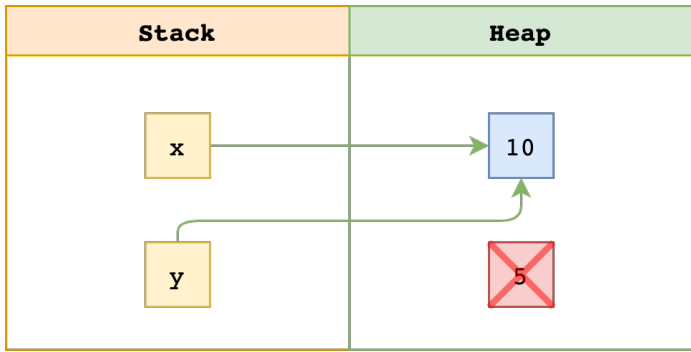


In the case that two variables have the same value, Python avoids duplication of value to save memory. It simply change the reference the point to the existing value of 10.

Memory management: How do Python stores variables?

The value 5 is now **without any reference** and thus unused.

The Python **garbage collector** is in charge to **deallocate (free)** the corresponding memory.



Getting the id of an object

Each object has a different identification number, which is **unique and constant during the object lifetime**. The same object will have different ids during different runs of a program.

In the CPython implementation, this number corresponds to the **address of the object in memory**.

We can use the **id(object)** function to get the id number.
Let's verify what seen in the memory management section using id.

```
>>> x = 10
>>> id(x)
4534361360

>>> y = 5
>>> id(y)
4534361392

>>> y = 10
>>> id(y)
4534361360
```

Output instructions

As already seen previously in the examples, in order to output text on the console, we can use the **print()** instruction:

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

We can pass to this instruction the data we want to print.

It is possible to provide directly a string or a variable containing a string or a data type that can be converted to a string.

Input instructions

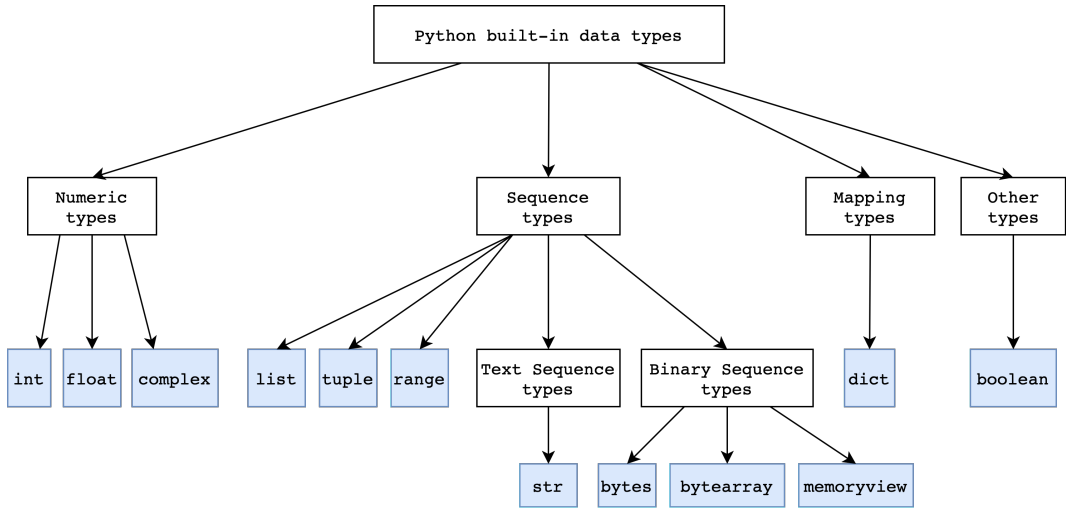
To let the user input text from the console we use the `input()` instruction.

```
name = input("Insert your name")
```

As visible in the example, we can also pass to the instruction the text that we want to be prompted to the user.

The value inserted by the user will be available in our name variable as a `string` data type.

Python data types



Strings

The **string** datatype represent a sequence of characters.

All the characters that are defined inside quotes is considered part of the string.

In Python you can choose whether to use **single or double quotes** around a string.

```
"this is a valid string"  
'this is also a valid string'
```

The fact that it is possible to choose single or double quotes allows to use **quotes and apostrophes inside a string**.

```
"I can use 'single quotes' inside a string!"  
'if I start my string with single quotes, I can use "double  
quotes" inside!'
```


Strings: some operations

When working with strings it is often necessary to manipulate them in different ways.

Some of the typically operations on strings include:

- changing case
- concatenation
- adding whitespaces
- striping whitespaces

Strings: methods, an overview

In Python, to perform some of the operation mentioned above, we can use already available **methods** that we can call on string objects.

Generally speaking, methods are **functions** that are **attached to and act upon a specific object**, which are triggered with a **call expression**.

They are called using the symbol dot (.) after a variable name, then specifying the **name of the method** to execute followed by a **set of perentheses**.

Let's assume we want to use the method `bar()` on a variable named `foo`. The instruction to use will look like this:

```
foo.bar()
```

We will go much more in detail about methods usage later in this course.

Strings: changing case

To change the case of the **first character of each word** it is possible to use the method **title()**.

If we have for example a string with the person name `albert einstein`, and we want to transform it into `Albert Einstein`, we can do the following:

```
name = "albert einstein"
print(name.title())
```

What happens is that the **method title** gets **called on our name variable**, returning **the new value** with capital letters.

Our print statement will show this output:

```
Albert Einstein
```

String: changing case

To change the case of the whole string to all uppercase or all lowercase letters we can use the methods **upper()** and **lower()**.

```
name = "Albert Einstein"
print(name.upper())
print(name.lower())
```

Our print statement will show this output:

```
ALBERT EINSTEIN
albert einstein
```

Strings: combining strings

When programming it is often necessary to combine strings that are saved in different variables.

Let's assume that we have the variables `first_name` and `last_name`, each one containing a piece of the full name.

In Python, we can combine strings using the plus (+) operator.

```
first_name = "albert"
last_name = "einstein"
full_name = first_name + " " + last_name
print(full_name.title())
```

Our print statement will show this output:

Albert Einstein

Strings: adding whitespaces

A *whitespace* is a non printable character such as space, tab, or end-of-line symbols.

The usage of whitespaces is important useful for example to enhance the readability for the users.

To add a **tab** to a string, just add the characters `\t`

```
print("Programming")  
print("\tStrings manipulation")
```

Our print statement will show this output:

```
Programming  
  Strings manipulation
```

Strings: adding whitespaces

To add a **newline** in a string, just add the characters `\n`

```
print("Programming languages:\nPython\nJava\nJavaScript")
```

Our print statement will show this output:

```
Programming languages:  
Python  
Java  
JavaScript
```

Strings: adding whitespaces

It is of course possible to combine newlines and tabs as best suited for our needs.

If for instance we use the combination `\n\t` we will tell Python to **go to the next line and add a tab**.

```
print("Programming languages:\n\tPython\n\tJava\n\tJavaScript")
```

Our print statement will produce this output:

```
Programming languages:
    Python
    Java
    JavaScript
```


Strings: adding whitespaces

Be careful, what happens if I print the following path?

```
print('C:\some\name')
```

In this case we can use the *raw string* by adding an **r** before our string.

```
print(r'C:\some\name')
```

Strings: stripping whitespaces

Sometimes we have the need to **remove whitespaces** that shouldn't be in our string. This operation is important in particular when we want to **compare strings** as we will see.

If we have a string containing undesired whitespaces at the end (right strip), we can use the method **rstrip()** to remove them.

```
favorite_course = "Programming "  
print(favorite_course.rstrip())
```

The method `rstrip()` will remove the space at the end of the string, printing only "Programming".

Strings: stripping whitespaces

Pay attention that the **rstrip()** method is not actually modifying the string!

If we print again our variable `favorite_course`, we will obtain "Programming".

This is because the method returns a new string (a modified copy of the original one).

If we want to keep the modified version of the string we need to re-assign our variable with the result of the instruction as follows:

```
favorite_course = "Programming "  
favorite_course = favorite_course.rstrip()
```

Strings: stripping whitespaces

Other methods available to strip whitespaces from a string are the following

- `lstrip()`: strip whitespaces at the beginning of the string (left strip)
- `strip()`: strip whitespaces from both the beginning and the end of the string

Strings: other methods

There are of course a lot of other operations that can be performed on strings that we will encounter later during the course.

For a complete list of methods on strings, please refer to the official documentation:

<https://docs.python.org/3/library/stdtypes.html#string-methods>

Numbers: integers

Integer numbers are managed in Python using the **integer** data type.

To put an integer value in a variable just assign the number needed.

```
num = 5
```

It is possible to work with integer numbers using the following operators:
add: (+), subtract: (-), multiply: (*), divide: (/)

```
>>> 2 + 5
7

>>> 4 - 1
3

>>> 5 * 3
15

>>> 5 / 2
2.5
```

Numbers: integers

Other useful operation with integers are:

- `//` : floored quotient
- `%` : remainder of the division
- `**` : exponent calculation

```
>>> 5//2
2
>>> 7%2
1
>>> 2**3
8
```

Numbers: integers

Python allows to use multiple operations in a single expression.

It is also possible to use **parentheses to specify the order** of operations.

```
>>> 3 + 5 * 2
13

>>> (3 + 5) * 2
16
```


Numbers: integers

In contrast to other languages that provide a fixed number of bits to represent integers, Python integers have **unlimited precision**.

This means that we can represent any **integer** value we want, independently from how large the number is.

Numbers: floating points numbers

In Python, any number that has a decimal points is called **float**.

Other languages differentiate in multiple types for decimal numbers, depending on the needed precision (float, double). Python floats are usually implemented using double in C (64 bits [IEEE 754 double precision](#))

```
>>> 0.1 + 0.1
0.2

>>> 1.6 * 4
6.4

>>> 5 * 0.1
0.5
```

Numbers: floating points numbers

It can happen that some time, you will get an unexpected number of decimals

```
>>> 0.2 + 0.1
0.30000000000000004

>>> 1.6 * 3
4.8000000000000001
```

This is something that happens in all programming languages and is due to the way floating point numbers are represented internally.

You can find more on this topic at the following link:
<https://docs.python.org/3/tutorial/floatingpoint.html#tut-fp-issues>

Numbers: additional operations

A lot of additional numeric operations are available in Python.

Please refer to the official documentation on numeric types:

<https://docs.python.org/3/library/stdtypes.html#numeric-types-int-float-complex>

See also the math and cmath modules documentation.

- math: <https://docs.python.org/3/library/math.html#module-math>
- cmath: <https://docs.python.org/3/library/cmath.html#module-cmath>

Comparison operators

To **compare two values** to establish an order we can use the **comparison operators**.

Operator	Description
==	equal
!=	different
>	greater
>=	greater or equal
<	lesser
<=	lesser or equal

The **result** of the evaluation of a comparison operator is a **boolean** value True or False.

Boolean

The boolean type is a data type that can assume uniquely two values: **True** and **False**

The boolean values can be obtained also from the evaluation of conditional expression such as:

```
result = rate > 0.5
```

The variable `result` will contain the value `True` if the variable `rate` has a value greater than `0.5`, otherwise if the value is equal or lesser than `0.5`, it will contain the value `False`

This data type is used in logical algebra, also known as **boolean algebra**. The name comes from George Boole, a British mathematician who, in the 19th century, was the first to define a logical algebraic system.

Boolean operators

Set of values: {False, True}

Operators:

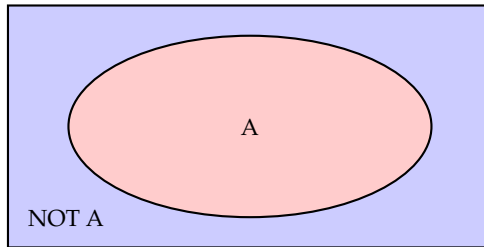
NOT logical negation, in Python: **not**

AND logical conjunction, in Python: **and**

OR logical disjunction, in Python: **or**

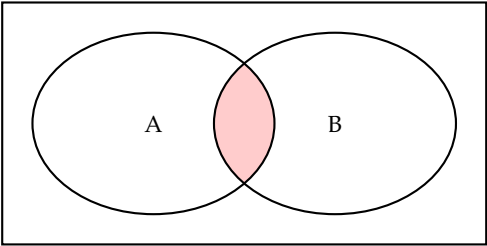
Boolean operators: NOT

A	not A
False	True
True	False



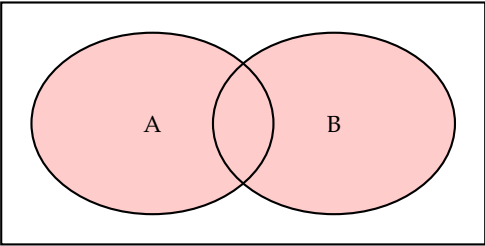
Boolean operators: AND

A	B	A and B
False	False	False
False	True	False
True	False	False
True	True	True



Boolean operators: OR

A	B	A or B
False	False	False
False	True	True
True	False	True
True	True	True



Boolean operators: examples

Some example of usage of the logical operators that can be applied on boolean data types.

```
b = 3 > 2 or 5 < 7
# b = True

b = 2 > 3 or 5 > 7
# b = False

b = 2 > 3 and 5 < 7
# b = False

b = not 3 > 2
# b = False

b = not 2 > 3
# b = True
```

Truthiness in Python

In Python it is possible to **evaluate any kind of value, not only booleans, to a boolean type**.

Basically the notion of true and false is **related to every object**.

For example, the number 0 is considered a False value.

All other numbers are instead considered True.

Other objects are considered False if empty, True otherwise.

We can try to convert a value to boolean using the **bool(object)** function.

```
>>> bool("a string")
True
```

Truthiness in Python

Some examples of how objects evaluate to booleans:

Value	Boolean
"a string"	True
""	False
1	True
0.0	False
[1, 2]	True
[]	False
{"a": "b"}	True
{}	False
None	False

None object

The **None** object is a special object that represents an empty placeholder.

The meaning of None is something like *no value present*.

It can be used for example to preallocate a list with n items for which we do not have the value.

Example:

```
l = [None] * 5
print(l)
[None, None, None, None, None]
```

It is similar to the null literal available in Java.

Getting the type of a variable

In Python, if we need to discover the type of a variable or a value in general, it is possible to use the `type()` function.

Here some usage examples:

```
>>> a = "some text"
>>> type(a)
<class 'str'>

>>> type(15)
<class 'int'>

>>> b = 14.67
>>> type(b)
<class 'float'>

>>> type(True)
<class 'bool'>
```

Conversion between data types

Some times it is possible that we want to **convert between one data type to another**.

Let's take an easy scenario. We want to create a the string "number: 5" by combining the string "number: " and the integer 5.

```
>>> "number: " + 5
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

The Python interpreter gives us an error since we are trying to combine different data types.

Converting to string

In order to solve the problem we need a way to convert our number to a string so that we can then concatenate the two.

To convert an object to its string version, we can use the `str(obj)` function.

Our example than becomes:

```
>>> "number: " + str(5)
'number5'
```

Now that the two data type are compatible we get the expected result.

Converting to int

Let's assume that this time we want to calculate the result of a simple sum but we have one number expressed as string and another one as int.

If we try to sum them we would end up with the same problem as seen before with string concatenation.

```
>>> 3 + "34"
Traceback (most recent call last):
  File "<input>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

Converting to int

Similar as before, to convert a compatible type to its integer version, we can use the `int(obj)` function.

Our example than becomes:

```
>>> 3 + int("34")
37
```

Converting to float

As you can imagine, also with float numbers the problem is exactly the same.

To convert a compatible object to its float version, we can use the `float(obj)` function.

```
>>> 23.67 + float("14.02")
37.69
```

String formatting

As we already have seen several time we often need to combine together strings with numbers in order for example to print out results in our programs.

So far we always used string concatenation in combination with conversion methods as in this example:

```
>>> "my result is " + str(42)
'my result is 42'
```

Starting from Python 3.6 we can use **f-Strings** (formatted string literals) to do the same in a simple way.

String formatting: f-Strings

The syntax to use f-Strings is as follows:

```
f"some string {expression} some other string":
```

Here an example of usage:

```
>>> f"my result is {42}"  
'my result is 42'
```

By adding curly braces { } where we want inside our f-String, we can for example add multiple expressions, that will resolve in a string.

In the *expression* slot, we can use arbitrary expressions like {2*34}, or pass in a variable {my_variable}, ...

Bibliography

- Learning Python 5th edition, Oreilly - Mark Lutz: Chapters 3, 4, 5, 7, 10
- Python Crash Course, no starch press - Eric Matthes: Chapters 1, 2
- Python Official Documentation: <https://docs.python.org/3/tutorial/>
- Python Wiki: <https://wiki.python.org/>
- Real Python: <https://realpython.com>
- LearnByExample: <https://www.learnbyexample.org/python/>