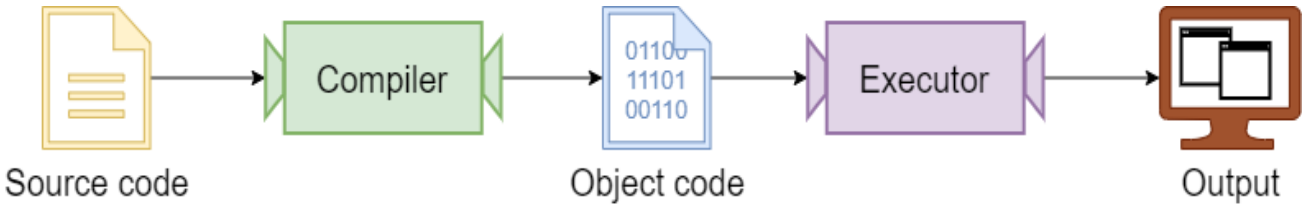
Chapter 1: Python introduction

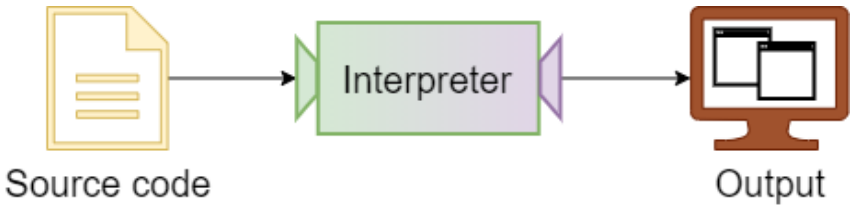
Compiled and interpreted languages

There are two major categories of programming languages: **compiled and interpreted**.

- A compiled language is a language that is turned by a compiler into direct machine code that runs upon the CPU. (Or it might run on a virtual machine stack like the JavaVM or the .NET runtime.)



- An interpreted language is a language that is read in its raw form and executed a statement at a time without being first compiled.



Python is an interpreted language.

How Jupyter Notebook works

A notebook contains cells, each cell is a logically separate and independent information.

There are 2 main type of cells:

- Markdown cells
  - Write documentation and comments
  - Can be formatted with *markdown* syntax
  - See this tutorial on how to use markdown (https://guides.github.com/features/mastering-markdown/)
- Code cells
  - Write code
  - Evaluate code on the fly

Note: you can find a *User Interface Tour* in the *Help* menu.

Let's get Python started! Hello world!

Literals

Literals are constants primitive values. They can be e.g. numbers or strings (texts). Strings are surrounded with quotation marks or apostrophes.

In [1]:

```
"Hello World"
```

Out[1]:

```
'Hello World'
```

In [2]:

```
'Hello Earth'
```

Out[2]:

```
'Hello Earth'
```

In [3]:

```
42
```

Out[3]:

```
42
```

Print

Syntax: `print` is a function, which has an argument. The argument is surrounded by parentheses.

- The argument can be a string literal between quotation marks
- Or a number literal
- Or a variable *(we will cover that later)*

In [4]:

```
print('Hello World')
print("Hello Earth")
print(4)
```

```
Hello World
Hello Earth
4
```

Task: what is the problem with the following codes?

In [5]:

```
print(Hello ELTE!)
```

```
  File "<ipython-input-5-10f8028f182a>", line 1
    print(Hello ELTE!)
                  ^
SyntaxError: invalid syntax
```

In [6]:

```
print "Hello Faculty of Informatics!"
```

```
  File "<ipython-input-6-015d2f017565>", line 1
    print "Hello Faculty of Informatics!"
          ^
SyntaxError: Missing parentheses in call to 'print'. Did you mean pr
int("Hello Faculty of Informatics!")?
```

Handling outer Python files

Viewing the content of an external file

Lines starting with the `%` symbol are not regular Python instructions, instead they are special commands for the Jupyter Notebook.

We can load the content of an external file with the special `%load` command:

In [7]:

```
# %load ../data/01_outerfile.py
print('Great! This line was print from an external file!')
```

```
Great! This line was print from an external file!
```

Note that after the loading the original `%load` command is commented out and instead the content of the external file is loaded.

Executing an external file

We can execute the content of an external file with the special `%run` command.

Specifying the `-i` flag is important, so the file is executed in the same environment with the Jupyter notebook. Therefore, if we e.g. declare a variable in the external script, it will be accessible in the code cells of the notebook. If the `-i` flag is not specified, the external Python file is evaluated in a separate environment.

In [8]:

```
%run -i ../data/01_outerfile.py
```

```
Great! This line was print from an external file!
```

Variables

Variables can be considered **containers**. You can put anything inside a container, **without specifying the size or type**, which would be needed in e.g. Java, C++ or C#. Note that Python is case-sensitive. Be careful about using letters in different cases.

When assigning values, we put the variable to be assigned to on the left-hand side (LHS), while the value to plug in on the right-hand side (RHS). LHS and RHS are connected by an equal sign (=), meaning assignment.

In [9]:

```
x = 3  # integer
y = 3.1 # floating point number
z = "Hello!" # strings
Z = "Wonderful!" # another string, stored in a variable upper-case z.
print(x)
print(y)
print(z)
print(Z)
```

```
3
3.1
Hello!
Wonderful!
```

You can do operations on numeric values as well as strings.

In [10]:

```
sum_ = x + y # int + float = float
print(sum_)
```

```
6.1
```

In [11]:

```
v = "World!"
sum_string = z + " " + v # concatenate strings
print(sum_string)
```

```
Hello! World!
```

Naming convention

There are two commonly used naming style in programming:

1. **camelCase**
2. **snake_case** or **lower_case_with_underscore**

All variable (function and class) names must start with a letter or underscore (_). You can include numbers.

In [12]:

```
myStringHere = 'my string'
myStringHere
```

Out[12]:

```
'my string'
```

In [13]:

```
x = 3 # valid
x_3 = "xyz" # valid
```

In [14]:

```
3_x = "456" # invalid. Numbers cannot be in the first position.
```

```
  File "<ipython-input-14-520aa7218b05>", line 1
    3_x = "456" # invalid. Numbers cannot be in the first position.
     ^
SyntaxError: invalid decimal literal
```

You can choose either camel case or snake case. Always make sure you use one convention consistently across one project.

Simple Data Structures

In this section, we go over some common primitive data types in Python. While the word *primitive* looks obscure, we can think of it as the most basic data type that cannot be further decomposed into simpler ones.

Numbers

For numbers without fractional parts, we say they are ***integer***. In Python, they are called `int` .

In [15]:

```
x = 3
print(type(x))
```

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

For numbers with fractional parts, they are floating point numbers. They are named `float` in Python.

In [16]:

```
y = 3.0
print(type(y))
```

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

We can apply arithmetic to these numbers. However, one thing we need to be careful about is ***type conversion***. See the example below.

In [17]:

```
z = 2 * x
print(type(z))
```

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

In [18]:

```
z = y + x
print(type(z))
```

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

Text/Characters/Strings

In Python, we use `str` type for storing letters, words, and any other characters.

To initialize a string variable, you can use either double or single quotes.

In [19]:

```
my_word = "see you"
print(type(my_word))
```

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

Unlike numbers, `str` is an iterable object, meaning that we can iterate through each individual character. You can think of strings as a sequence of characters (or a **list** of characters, which we will cover later). In this case, indices and bracket notations can be used to access specific ranges of characters.

In [20]:

```
print(my_word[1])   # character with index 1; Python starts with 0 NOT 1
print(my_word[2:6]) # [start, end), end is exclusive
print(my_word[-1])  # -1 means the last element
```

```
e
e yo
u
```

We can also use `+` to *concatenate* different strings

In [21]:

```
my_word + ' tomorrow'
```

Out[21]:

```
'see you tomorrow'
```

Formatted strings

Print with formatting with the `format()` function and using placeholders:

In [22]:

```
print("The sum of x and y is {0}".format(sum_))
```

```
The sum of x and y is 6.1
```

In [23]:

```
print("The string `sum_string` is '{0}'".format(sum_string))
```

```
The string `sum_string` is 'Hello! World!'
```

In [24]:

```
print("The sum of x and y is {0} and the string `sum_string` is '{1}''".format(s
um_, sum_string))
```

```
The sum of x and y is 6.1 and the string `sum_string` is 'Hello! Wor
ld!''
```

Boolean

Boolean type comes in handy when we need to check conditions. For example:

In [25]:

```
my_error = 1.6
compare_result = my_error < 0.1
print(compare_result)
```

```
False
```

In [26]:

```
print(type(compare_result))
```

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

There are two and only two valid Boolean values: `True` and `False`. We can also think of them as `1` and `0`, respectively.

In [27]:

```
print(my_error > 0)
```

```
True
```

When we use Boolean values for arithmetic operations, they will become `1 / 0` implicitly.

In [28]:

```
print((my_error > 0) + 2)
```

```
3
```

Type Conversion

Since variables in Python are dynamically typed, we need to be careful about type conversion.

When two variables share the same data type, there is not much to be worried about:

In [29]:

```
s1 = "no problem. "
s2 = "talk to you later"
s1 + s2
```

Out[29]:

```
'no problem. talk to you later'
```

But be careful when we are mixing variables up:

In [30]:

```
a = 3 # recall that this is an ____?
b = 2.7 # how about this?
c = a + b # what is the type of `c`?

print(c)
print(type(c))
```

```
5.7
<class 'float'>
```

To make things work between string and numbers, we can explicitly convert numbers into `str` :

In [31]:

```
print(s1 + 3)
```

```
---------------------------------------------------------------------
-------
TypeError                                 Traceback (most recent cal
l last)
~/Repos/Lecture/elte_teralg/data/01_outerfile.py in <module>
----> 1 print(s1 + 3)

TypeError: can only concatenate str (not "int") to str
```

In [32]:

```
print(s1 + str(3))
```

```
no problem. 3
```

We may also convert strings to numbers:

In [33]:

```
s3 = "42"
d = int(s3)
print(type(s3), type(d))
```

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

User input

User input can be easily requested in Python:

```
k = input('Question')
```

The question text is arbitrary. The user will see a console input prompt. The typed input will be stored in the `k` variable by Python. Example:

In [34]:

```
k = input('What is your name? ')
print('Hello ' + k)
```

```
Hello John
```

Question: what is the type of the value in variable `k` ?

In [35]:

```
print(type(k))
```

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

Exercise

Task: Query the height of the user.

Print afterward that *Your height is XXX centimeter.*

Print the type of the variable storing the height of the user.

In [36]:

```
height = input("What is your height? ")
print("Your height is {0} centimeters.".format(height))
print(type(height))
```

```
Your height is 186 centimeters.
<class 'str'>
```

Variables assigned by the **input** function will always contain strings. (We will cover error handling later.)

Summary exercise on Python variables, data types and user input

Task: Question the user for the amount of days he/she works a week and his/her gross salary per hour (in euro).

Calculate and print the monthly gross salary of the user.

Calculate and print the monthly net salary of the user, assuming that the net salary is 65% of the gross salary.

Assume that a month consists of 4 weeks and a working day consists of 8 working hours.

In [37]:

```
work_days = int(input("Number of work days per week? "))
salary_hour = int(input("Salary per hour? "))
salary_gross = 4 * work_days * 8 * salary_hour
salary_net = salary_gross * 0.65

print("Gross salary: {0} euros / month".format(salary_gross))
print("Net salary: {0} euros / month".format(salary_net))
```

```
Gross salary: 2560 euros / month
Net salary: 1664.0 euros / month
```