

Python Basics

Python and R for Data Science

Data Science and Management

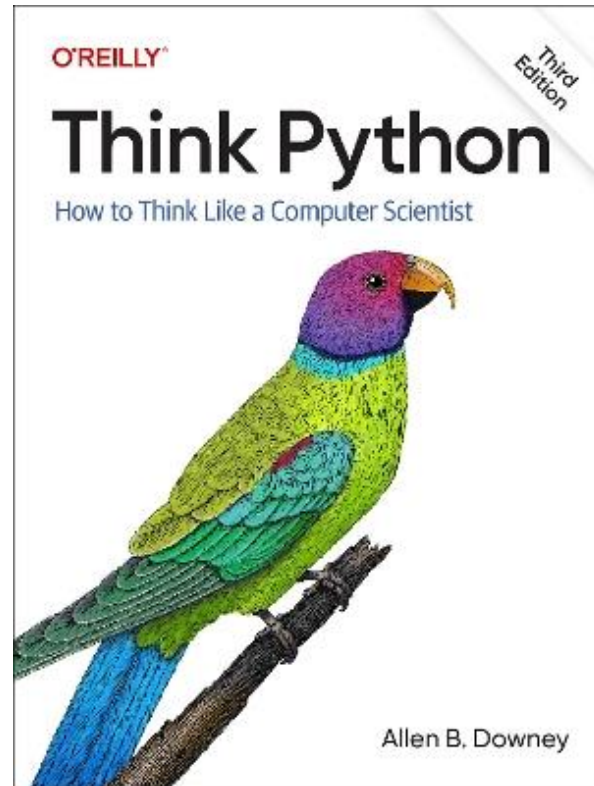


Python: from zero to hero



Preliminaries

Reference Book: Think Python (3rd ed.)



Think Python is an introduction to Python for people who have never programmed before – or for people who have tried and had a hard time.

Freely available: <https://alendowney.github.io/ThinkPython/>

Programs manipulate data

Given input data, compute output data.

Data can be anything:

- text file
- something on the web
- click of a mouse (input)
- text on the screen (output)

Two main categories of data

Data types can be either:

- scalar: a single value
- non scalar: a collection of values

Scalar data types in Python

Type	Python type	Examples
Number > integers	<code>int</code>	<code>5</code> , <code>-2</code>
Number > floating point	<code>float</code>	<code>5.0</code> , <code>-2.0</code>
boolean	<code>bool</code>	<code>True</code> , <code>False</code>
invalid value	<code>NoneType</code>	<code>None</code>

Non-scalar data types in Python

Type	Python type	Examples
textual	string	"Hello, World!", 'Bazinga123'
list	List	[1, 2, 3]
tuple	Tuple	("a", 2, 3.0)
set	Set	set("a", 1, 3.0)
dictionary	Dict	{"name": "Francesco", "surname" : "Totti"}

Printing data

We can print on the screen (output) using `print(<data>)`:

```
In [21]: print(5)
          print(5.0)
          print(True)
          print("Hello, Word!")
          print(None)
```

```
5
5.0
True
Hello, Word!
None
```

Variables

Programs manipulate non *constant* data

Real-world programs take the data from the external world (e.g., data from the web):

- no assumptions on data *value*
- assumptions on data *type*

Hence, programs must be correct, i.e., compute the correct result, regardless of the data value for which they were initially written and tested.

Notice that:

- if a program only works on constant data then it is enough to run it once and then use its result(s), without the need to run it again.
- if the program works on non-constant data then we have it from scratch every time the data has changed.

Non-constant data: **variables**

To represent data for which we cannot determine its constant value, we use variables:

```
In [22]: # let us suppose the current temperature is 35  
# we can now store this value into a variable  
current_temp = 35 # we call it current_temp  
print(current_temp) # we now print the variable _current_ value  
  
# after a bit of time, we measure the temperature again  
# and we find that the temperature has increased to 40  
current_temp = 40 # we change the value of the variable  
print(current_temp) # we now print the variable _current_ value  
  
# regardless of the current value, our code has not changed  
# i.e., the instruction performing the print is the same
```

35
40

Variables

- a variable is defined by using the assignment operator `=` :
`<name> = <data value>`
- `<name>` can be a mix of letters, numbers, and underscore (`_`)
- `<name>` cannot start with a number
- `<name>` cannot be equal to a Python keyword, i.e., reserved words (we will see them while introducing the language)
- `<name>` should be something meaningful for the human: Python does not care about the variable name as long as it is valid and unique.
- [PEP-8](#), which defines several style guidelines, suggests to use lowercase descriptive words separated by one underscore

Variables can... vary

- A variable will have a specific data type, e.g., `int` or `bool`
- We cannot make assumptions on its value when writing the code but we have to know its data type to write code that can handle it.
- Python allows us to re-assign the same variable name:

```
In [23]: current_temp = 35 # initial value  
current_temp = 40 # new value!
```

When re-assigning a variable, the previous value is lost.

- Python allows us to change the data type when doing a re-assignment:

```
In [24]: current_temp = 35 # here, the variable is an int  
current_temp = "IT IS (TOO) HOT" # now, it is a string
```

How to retrieve the data type?

Given a piece of data, we can retrieve its data type using `type(<data>)`:

```
In [25]: print(type(5))
         print(type(5.0))
         print(type(True))
         print(type("Hello, Word!"))
         print(type(None))
```

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

```
In [26]: current_temp = 35 # here, the variable is an int
         print(type(current_temp))
         current_temp = "IT IS (TOO) HOT" # now, it is a string
         print(type(current_temp))
```

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

None

None: informal definition

The `NoneType` has only the value `None`.

We use it when we do not know yet our data value.

For instance:

```
In [27]: temp = None # we do not know the temperature  
         print(temp)
```

None

None: operators

Given a variable, we can check whether it is (not) assigned to `None` using the `is` (`is not`) operator:

```
In [28]: temp = None  
print(temp is None)  
print(temp is not None)
```

```
True  
False
```

```
In [29]: temp = 35  
print(temp is None)  
print(temp is not None)
```

```
False  
True
```

Integers (`int`)

Integers (`int`): informal definition

Integer: sequence of numbers with no periods nor commas.

For instance:

```
In [30]: x = 10  
         y = -20  
         print(x)  
         print(y)
```

```
10  
-20
```

Integers (`int`): operators

Operator	Semantics	Example	Example Result	Result Type
<code>-</code> (unary)	Negation	<code>-5</code>	<code>-5</code>	<code>int</code>
<code>+</code>	Addition	<code>5 + -2</code>	<code>3</code>	<code>int</code>
<code>-</code> (binary)	Subtraction	<code>5 - -2</code>	<code>7</code>	<code>int</code>
<code>*</code>	Multiplication	<code>5 * -2</code>	<code>-10</code>	<code>int</code>
<code>/</code>	Division	<code>5 / -2</code>	<code>-2.5</code>	<code>float</code>
<code>//</code>	Floor Division	<code>5 // -2</code>	<code>-3</code>	<code>int</code>
<code>%</code>	Remainder (modulo)	<code>5 % 2</code>	<code>3</code>	<code>int</code>
<code>**</code>	Exponentiation	<code>5**2</code>	<code>25</code>	<code>int</code>

Integers (int): try!

```
In [9]: x = 5  
y = -2  
print(-x)  
print(x + y)  
print(x - y)  
print(x * y)  
print(x / y)  
print(5 // y)  
print(x % y)  
print(x ** -y)
```

```
-5  
3  
7  
-10  
-2.5  
-3  
-1  
25
```

Integers (`int`): operator order

Operators have a similar priority rules as in mathematics.
Use parentheses `()` to force the order.

```
In [32]: x = 5
          y = 2
          z = 3
          print(x + 2 * z) # same as the next one due to priority
          print(x + (y * z))
          print((x + y) * z) # force a specific operator order
```

```
11
11
21
```

Floating point (float)

Floating point (float): informal definition

Floating point: sequence of numbers containing one . .

For instance:

```
In [33]: pi = 3.14159  
x = -5.0  
print(pi)  
print(x)
```

```
3.14159  
-5.0
```

Floating point (float): operators

Operator	Semantics	Example	Example Result	Result Type
- (unary)	Negation	-5.0	-5.0	float
+	Addition	5.0 + -2.0	3.0	float
- (binary)	Subtraction	5.0 - -2.0	7	float
*	Multiplication	5.0 * -2.0	-10.0	float
/	Division	5.0 / -2.0	-2.5	float
//	Floor Division	5.0 // 2.0	2	float
%	Remainder (modulo)	5.0 % -2.0	-1.0	float
**	Exponentiation	5.0**2.0	25.0	float

Floating point (float): try!

```
In [7]: print("Negation:", -5.0)
print("Addition:", 5.0 + -2.0)
print("Subtraction:", 5.0 - -2.0)
print("Multiplication:", 5.0 * -2.0)
print("Division:", 5.0 / -2.0)
print("Floor Division:", 5.0 // 2.0)
print("Reminder:", 5.0 % -2.0)
print("Exponentiation:", 5.0 ** 2.0)
print(5.0 + 2.0 * 3.0) # same as the next one due to priority
print(5.0 + (2.0 * 3.0))
print((5.0 + 2.0) * 3.0) # force a specific operator order
```

```
Negation: -5.0
Addition: 3.0
Subtraction: 7.0
Multiplication: -10.0
Division: -2.5
Floor Division: 2.0
Reminder: -1.0
Exponentiation: 25.0
11.0
11.0
21.0
```

`int` and `float` aspects

What if we mix `ints` and `floats`?

If one of the operands is a `float`, then the result will be a `float`

```
In [35]: print(5 + 2.0)  
         print(5.0 - 2)
```

7.0

3.0

How to convert from `int` to `float` (or viceversa)?

Use `int(<data>)` and `float(<data>)`:

```
In [36]: print(int(3.6))  
         print(float(5))
```

```
3  
5.0
```

Notice that:

- converting a `float` to an `int` may lead to a loss of information
- `int(<data>)` truncates the number to its decimal part

If want to round a `float` to the nearest integer, then we can use `round(<data>)`:

```
In [37]: print(round(3.6))
```

```
4
```

Python still obliges to math rules!

Invalid math operations

```
In [38]: 5 / 0 # this cannot be done... it will raise an exception
```

```
-----  
--  
ZeroDivisionError                                Traceback (most recent call las  
t)  
Cell In[38], line 1  
----> 1 5 / 0 # this cannot be done... it will raise an exception  
  
ZeroDivisionError: division by zero
```

In general, an invalid operation in Python raises an `Exception`.
We will learn how to deal with `Exceptions` later on.

Python has a fixed amount of resources!

Fixed precision

```
In [ ]: print(1 / 3) # A computer keep tracks of an approximation  
         # of the number. Hence, there are no periodic  
         # numbers in Python.  
  
print((1 / 3) * 3) # still, in Python, in "easy" cases, we get back  
                  # what we expect.  
  
print(3.14 + 1) # however, in other cases, we only get an apx  
               # Here, we get something slightly bigger than 4.14!  
               # This is due to the way computers store numbers.
```

0.3333333333333333
1.0
4.1400000000000001

Mixing assignment and arithmetic operators

Often, we write:

```
In [ ]: x = 2  
        y = 4  
        x = x + y  
        x = x * y  
        x = x / y
```

These arithmetic+assignment operations can be written as:

```
In [ ]: x += y  
        x *= y  
        x /= y  
        x
```

Booleans (`bool`)

Booleans (`bool`): informal definition

Boolean: either `True` or `False`.

For instance:

```
In [ ]: is_raining = False
         is_too_hot = True
         print(is_raining)
         print(is_too_hot)
```

False

True

Booleans (`bool`): operator `not`

`not` performs the negation of the operand:

- the result is `True` when the operand is `False`
- the result is `False` when the operand is `True`

A	Expression: <code>not</code> A	Result
<code>False</code>	<code>not False</code>	<code>True</code>
<code>True</code>	<code>not True</code>	<code>False</code>

Booleans (`bool`): operator `or`

`or` performs the disjunction between two boolean operands:
the result is `True` when at least one of the operands is `True`.

A	B	Expression: A <code>or</code> B	Result
<code>True</code>	<code>True</code>	<code>True or True</code>	<code>True</code>
<code>True</code>	<code>False</code>	<code>True or False</code>	<code>True</code>
<code>False</code>	<code>True</code>	<code>False or True</code>	<code>True</code>
<code>False</code>	<code>False</code>	<code>False or False</code>	<code>False</code>

Booleans (bool): operator and

and performs the conjunction between two boolean operands:
the result is True when both operands are True .

A	B	Expression: A or B	Result
True	True	True or True	True
True	False	True or False	False
False	True	False or True	False
False	False	False or False	False

Booleans (bool): operator xor

\wedge performs the exclusive or between two boolean operands:
the result is True when exactly one of the two operands is True.

A	B	Expression: A \wedge B	Result
True	True	True \wedge True	False
True	False	True \wedge False	True
False	True	False \wedge True	True
False	False	False \wedge False	False

Booleans (`bool`): recap binary operators

A	B	A or B	A and B	A ^ B
False	False	False	False	False
True	False	True	False	True
False	True	True	False	True
True	True	True	True	False

Booleans (bool): try!

```
In [ ]: print(not True)
        print(True or False)
        print(True and False)
        print(True ^ False) # this is the XOR operator!
```

```
False
True
False
True
```

Why booleans are useful?

1. A `bool` will be generated when, e.g., comparing other types:

- Is `5` smaller than `2`? `False`

2. We often want to perform a task:

- when a condition is not true: `not`
- when one of the conditions is true: `or`
- when all conditions must be true: `and`

3. We need to enable or disable a feature

Data comparison

Comparison operators

Both `int`s and `float`s can be compared using a comparison operator and the result will be a `bool`:

Operator	Semantics	Example	Example Result
<code><</code>	Less than	<code>1 < 3</code>	<code>True</code>
<code><=</code>	Less or equal than	<code>5 <= 6</code>	<code>False</code>
<code>></code>	Greater than	<code>1 > 6</code>	<code>False</code>
<code>>=</code>	Greater or equal than	<code>1 <= 6</code>	<code>True</code>
<code>==</code>	Equal to	<code>5 == 6</code>	<code>False</code>
<code>!=</code>	Not equal to	<code>5 != 6</code>	<code>True</code>

Strings (`string`)

Strings: informal definition

String: sequence of any *kind* of characters enclosed in single (`'`) or double (`"`) quotes. For instance:

```
In [ ]: university_short_name = "LUISS"  
university_long_name = 'Libera Università Int. degli Studi Sociali'  
print(university_short_name)  
print(university_long_name)
```

```
LUISS  
Libera Università Int. degli Studi Sociali
```

```
In [ ]: university_short_name = "LUISS' # we need to be consistent  
                                             # with the use of quotes!
```

```
File "/tmp/ipykernel_794386/1682044638.py", line 1  
    university_short_name = "LUISS' # we need to be consistent  
                             ^
```

```
SyntaxError: unterminated string literal (detected at line 1)
```

How to have a string containing a quote?

If we want to have a quote *within* a string, we have several strategies:

```
In [ ]: s1 = 'Hello "world"'      # we can use double quotes
        # inside single quotes
s2 = "Hello 'world'"           # we can use single quotes
        # inside double quotes
s3 = "Hello \"world\""         # we can use the escape \ to
        # make the double quotes
        # part of the string
s4 = '''Hello 'world' '''      # we can use the triple quotes
```

String: escape characters

Some characters, when escaped, will be treated in a specific way by `print`:

```
In [ ]: s1 = "Hello\nWorld!" # \n is the newline  
print(s1)
```

```
Hello  
World!
```

```
In [ ]: s2 = "Hello\tWorld!" # \t is the tabular  
print(s2)
```

```
Hello  World!
```

```
In [ ]: s3 = "Hello\\World!" # \\ prints \  
print(s3)
```

```
Hello\\World!
```


String: operators

Operator	Semantics	Example	Example Result
<code>+</code>	Concatenation	<code>"a1" + "b2"</code>	<code>"a1b2"</code>
<code>*</code>	Repetition	<code>"a1" * 3</code>	<code>"a1a1a1"</code>
<code>in</code>	Membership	<code>"LU" in "LUISS"</code>	<code>True</code>
<code>not in</code>	Membership	<code>"LU" not in "LUISS"</code>	<code>False</code>
<code>==</code>	Equality	<code>"LUISS" == "luiss"</code>	<code>False</code>
<code>!=</code>	Inequality	<code>"LUISS" != "luiss"</code>	<code>True</code>

NOTE: strings are case sensitive in Python!

String: indexing

Using indexing, we can access a specific character of a string:

```
In [ ]: s1 = "Hello\nWorld!"  
print(s1[0]) # we access the first character  
print(s1[4]) # we access the fifth character  
  
i = 2  
print(s1[i]) # we access the (i+1)-th character
```

```
H  
o  
l
```

NOTE: indexes start their count from zero (not from one)!

String: negative indexing?

Quite strangely, Python supports negative indexes:

```
In [ ]: s1 = "Hello\nWorld!"  
print(s1[-1]) # we access the last character  
print(s1[-2]) # we access the second last character  
  
!  
d
```

Strings are immutable

Even if a `string` is a non-scalar data type, it is immutable, i.e., we cannot change its internal data. Hence, we cannot change an existing string:

```
In [ ]: s1 = "Hello\nWorld!"  
s1[5] = ", " # this will raise an exception
```

```
-----  
--  
TypeError                                Traceback (most recent call las  
t)  
/tmp/ipykernel_1885124/1214696963.py in <module>  
      1 s1 = "Hello\nWorld!"  
----> 2 s1[5] = ", " # this will raise an exception  
  
TypeError: 'str' object does not support item assignment
```

Notheless, we can always create a new string:

```
In [ ]: s1 = "Hello\nWorld!"  
s2 = s1[:5] + ", " + s1[6:] # concatenation gives a new string  
print(s2)
```

Hello, World!

String: slicing

We can get a slice, i.e., substring, by specifying the starting and ending index:

```
In [ ]: s1 = "Hello\nWorld!"  
print(s1[0:5]) # we access the first five characters  
           # the last index is not included
```

Hello

When the starting or ending index is the first and last position, respectively, then we can omit them:

```
In [ ]: print(s1[:5]) # starting index is 0  
print(s1[6:]) # ending index is the last character
```

Hello
World!

We can even use negative indexes:

```
In [ ]: print(s1[-6:]) # we access the last three characters
```

World!

Strings: interpolation

We can build string by embedding the values of other variables:

- Interpolation with `f`-strings:

```
In [ ]: pi = 3.14159
msg = f"An apx of pi is {pi}" # notice the f before the string
print(msg)
msg = f"An apx of pi is {int(pi)}"
print(msg)
```

An apx of pi is 3.14159

An apx of pi is 3

Strings: interpolation (cont'd)

- Interpolation with modulo operator:

```
In [ ]: msg = "An apx of pi is %f" % (pi,) # %f means print as a float
        print(msg)
        msg = "An apx of pi is %d" % (pi,) # %d means print as an int
        print(msg)
        msg = "An apx of pi is %s" % (pi,) # %s means print as a string
        print(msg)
        msg = "An apx of pi is %f, or after truncation, %d" % (pi, pi)
        print(msg)
```

An apx of pi is 3.141590

An apx of pi is 3

An apx of pi is 3.14159

An apx of pi is 3.141590, or after truncation, 3

In the next weeks, we will see more advanced aspects of interpolation.

Multiple arguments in `print`

`print` can take more than one argument:

```
In [ ]: s1 = "Hello"
s2 = "World"
print(s1, s2) # the resulting output will have
               # a space between the two strings
```

Hello World

String operations

Given a string `s = "cia0"` and `s2 = "ia"`:

Operations	Semantics	Result
<code>len(s)</code>	length, i.e., number of chars	<code>4</code>
<code>s.lower()</code>	lowercased <code>s</code>	<code>ciao</code>
<code>s.upper()</code>	uppercased <code>s</code>	<code>CIA0</code>
<code>s.capitalize()</code>	titlecased <code>s</code>	<code>Ciao</code>
<code>s.count(s2)</code>	count occurrences of <code>s2</code> in <code>s1</code>	<code>1</code>
<code>s.find(s2)</code>	finds index of <code>s2</code> in <code>s</code> , or <code>-1</code>	<code>2</code>
<code>s.index(s2)</code>	finds index of <code>s2</code> in <code>s</code> , or raise exception	<code>2</code>

We will see more examples later on. This just to give a hint about what Python can give you for free.

String operations (cont'd)

Given a string `s = "cia0"` and `s2 = "ia"`:

Operations	Semantics	Result
<code>s.startswith(s2)</code>	check if <code>s</code> starts <code>s2</code>	False
<code>s.endswith(s2)</code>	check if <code>s</code> ends <code>s2</code>	False
<code>s.istitle()</code>	check if <code>s</code> is titlecased	False
<code>s.islower()</code>	check if <code>s</code> is lowercase	False
<code>s.isupper()</code>	check if <code>s</code> is uppercase	False
<code>s.isalpha()</code>	check if <code>s</code> contains only letters	True
<code>s.isdigit()</code>	check if <code>s</code> contains only digits	False
<code>s.isalnum()</code>	check if <code>s</code> contains only letters or digits	True

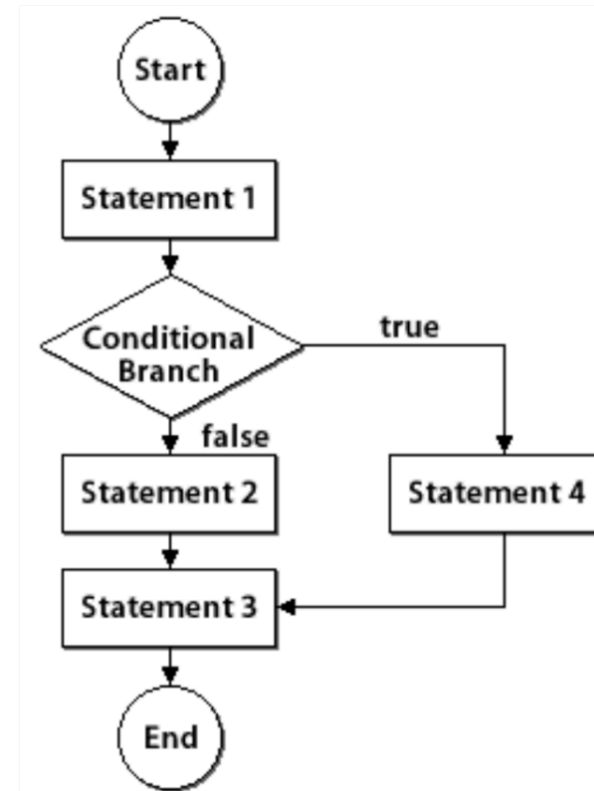
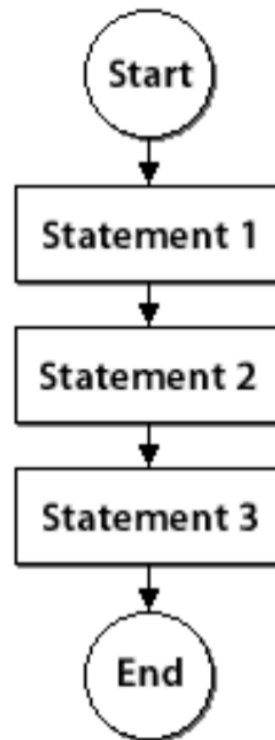
We will see more examples later on. This just to give a hint about what Python can give you for free.

Conditional execution

Conditionally execute something?

No branching

Branching



Statements 1, 2, and 3
are always executed

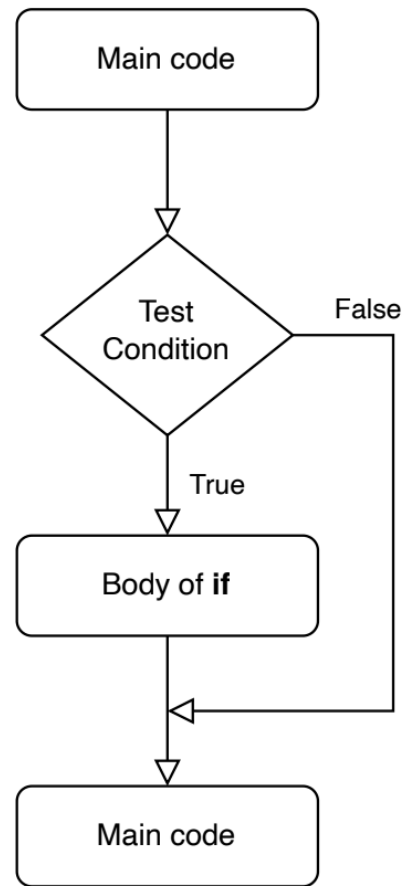
First, statement 1 is executed. If the condition is true, then statement 4 is executed, otherwise statement 2 is executed. Finally, statement 3 is executed.

Why to conditionally execute?

Most programs perform some tasks only when some conditions are met. E.g.:

- IF *the temperature is greater than 30* THEN *start the AC*
- IF *it is raining* THEN *take the umbrella* ELSE *leave the umbrella*

Conditional: **if** (flow diagram)



Conditional: `if`

The `if` statement allows us to conditionally execute a group of instructions (*body of the if*) based on a condition:

```
if <COND> :  
    <instruction #1>  
    <instruction #2>  
    [...]  
    <instruction #N>
```

Important remarks:

- `<COND>` must be a boolean. This could be yielded by the evaluation of different boolean condition through `not`, `and`, and `or` operator. Also, the condition may be generated by using a comparison operator.
- after `<COND>` we must have `:`
- the body of the `if` must be indented. Indentation is, by convention, four spaces or a tabular (tab key on your keyboard!).

```
In [ ]: current_temp = 50 # suppose we take it from a sensor
```

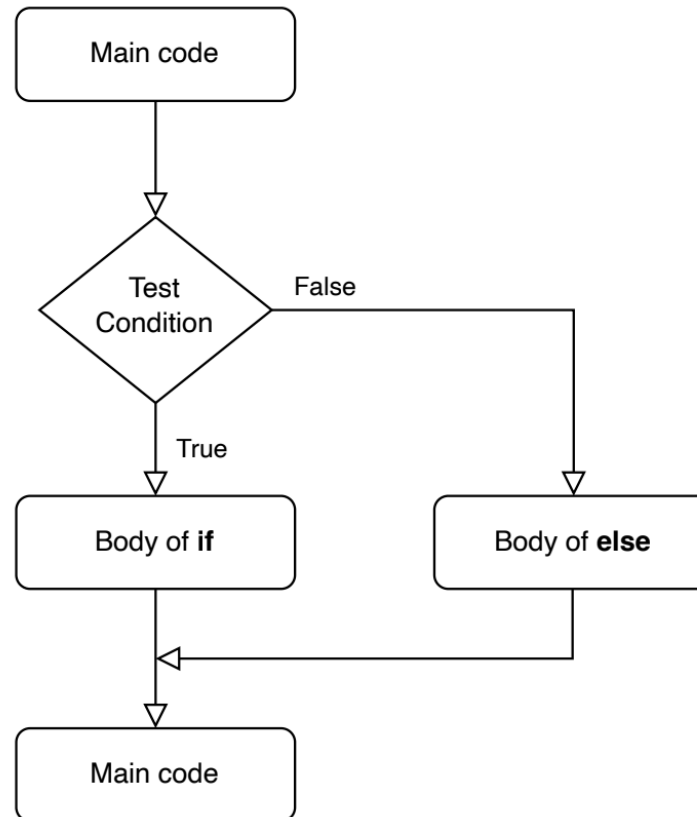
```
In [ ]: if current_temp > 30:  
        print("Need to start the AC!") # the body is one instr.  
print("---") # this is not part of the if body
```

```
Need to start the AC!  
---
```

```
In [ ]: if current_temp > 45:  
        print("Need to start the AC...") # body first instr.  
        print("or go to vacation")      # body second instr.  
print("---") # this is not part of the if body
```

```
Need to start the AC...  
or go to vacation  
---
```


Conditional: **if-else** (flow diagram)



Conditional: `if-else`

The `if` and `else` statements allows us to alternatively execute two groups of instructions (*body of the if* and *body of the else*, respectively) based on a condition:

```
if <COND> :  
    <instruction #A1>  
    <instruction #A2>  
    [...]  
    <instruction #AN>  
else:  
    <instruction #B1>  
    <instruction #B2>  
    [...]  
    <instruction #BN>
```

Important remarks:

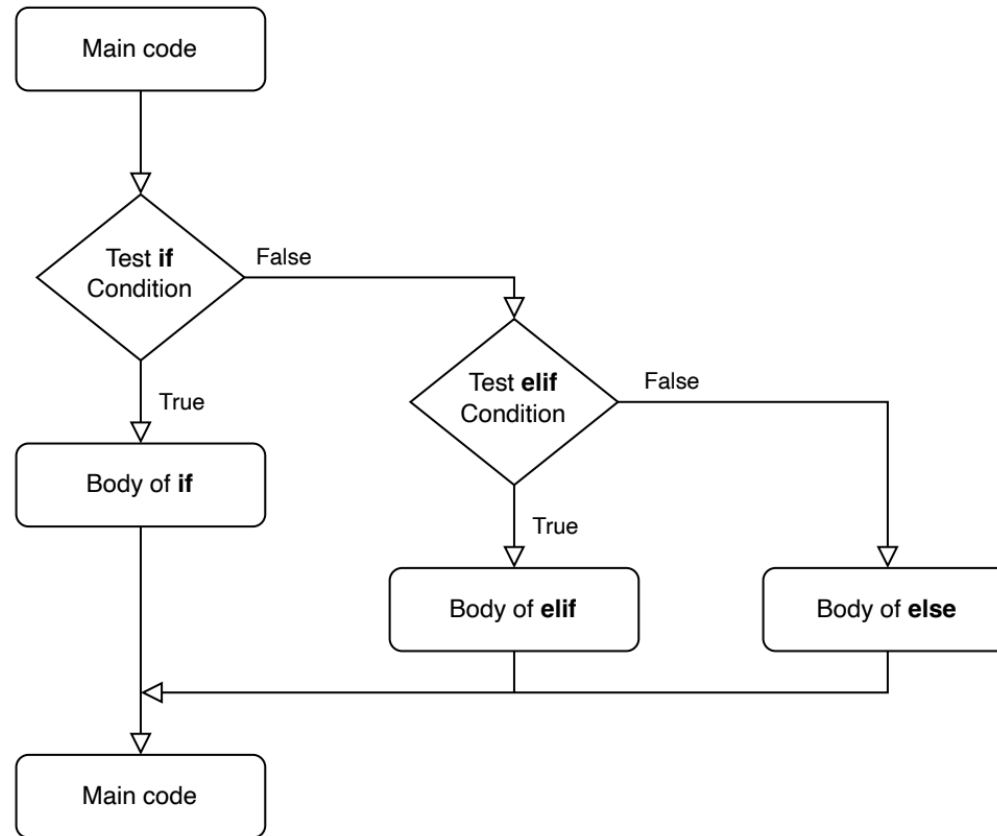
- after `else` we must have `:`
- the body of the `else` must be indented. Indentation is, by convention, four spaces or a tabular (tab key on your keyboard!). It must be consistent with the indentation of body of the `if`.

```
In [ ]: current_temp = 50 # suppose we take it from a sensor
```

```
In [ ]: if current_temp >= 20:  
        print("Warm") # the body of the if is one instr.  
    else:  
        print("Cold") # the body of the else is one instr.  
    print("weather") # this is not part of the if body
```

Warm
weather

Conditional: **if-elif-else** (flow diagram)



Conditional: `if-elif-else`

The `if`, `elif`, and `else` statements allows us to alternatively execute different groups of instructions:

```
if <COND-A> :  
    <instructions #A1-N>  
elif <COND-B> :  
    <instructions #B1-N>  
elif <COND-C> :  
    <instructions #C1-N>  
else:  
    <instructions #D1-N>
```

Important remarks:

- we can have an arbitrary number of `elif` statements
- each group of instructions must be indented
- the conditions are checked in order and the first one matching will lead to the related group of instructions (while other groups will be skipped)
- if no condition yields `True` then the body of the `else` is executed

```
In [ ]: current_temp = 50 # suppose we take it from a sensor
```

```
In [ ]: if current_temp >= 40:
        print("Hell")      # the body of the if is one instr.
    elif current_temp >= 30:
        print("Hot") # the body of the elif is one instr.
    elif current_temp >= 20:
        print("Warm") # the body of the elif is one instr.
    else:
        print("Cold") # the body of the else is one instr.
    print("weather") # this is not part of the if body
```

Hell
weather

Nested conditionals

We can nest one conditional within another conditional:

```
In [ ]: current_temp = 50 # suppose we take it from a sensor  
is_raining = True # suppose we take it from a sensor
```

```
In [ ]: if current_temp >= 0:  
    if is_raining:  
        print("Wet hot") # notice the double indentation  
    else:  
        print("Dry hot") # notice the double indentation  
else:  
    if is_raining:  
        print("Wet cold") # notice the double indentation  
    else:  
        print("Dry cold") # notice the double indentation
```

Wet hot

Nested conditionals (cont. d)

The previous code, it could be rewritten as:

```
In [ ]: if current_temp >= 0 and is_raining:
        print("Wet hot")
        elif current_temp >= 0 and not is_raining:
        print("Dry hot")
        elif current_temp < 0 and is_raining:
        print("Wet cold")
        else:
        print("Dry cold")
```

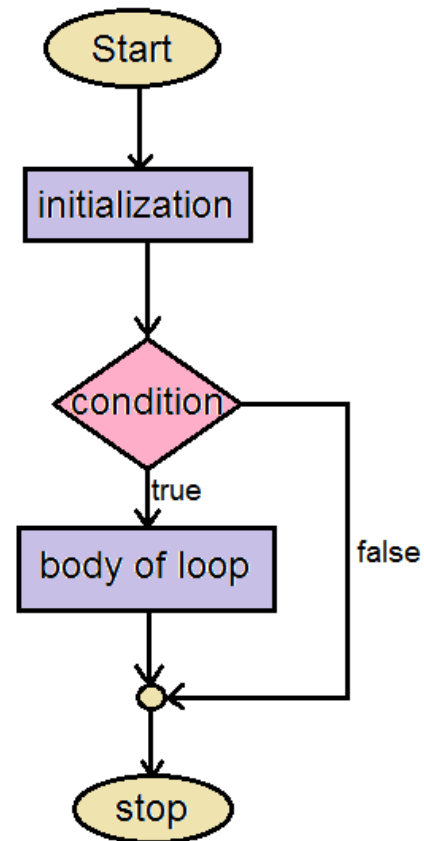
Wet hot

However, it would be slightly:

- less readable: less clear which are the common preconditions
- more error-prone: mistakes when rewriting the same condition more than once

Iteration

Iterate: repeated execution of something



Why we need to repeat the execution of something?

There are several cases where we need to repeatedly execute a group of instructions until a condition is met. For instance:

- read for the N times the current temperature and for each time sum it to an accumulator variable; finally compute the average.
- sum the first N positive numbers

Loop: `while`

The `while` statement allows us to repeat the execution of a group of instructions (body of the for) until a condition is met:

```
while <COND> :  
    <instruction #1>  
    <instruction #2>  
    [...]  
    <instruction #N>
```

Important remarks:

- `<COND>` must be a boolean
- after `<COND>` we must have `:`
- the body of the `while` must be indented.
- if `<COND>` is `True`, then the body of the `while` is executed once
- after the execution of the body, `<COND>` is evaluated again, deciding whether to execute again the body
- repeat until `<COND>` is `False`

```
In [ ]: n = 10 # change as you wish
k = 0
sum_temps = 0
while k < n:
    current_temp = 35 # suppose we take it from a sensor
                     # making something that changes
    sum_temps += current_temp
    k += 1
avg_temp = sum_temps / n
print("Average temperature is", avg_temp, "after", n, "measurements")
```

Average temperature is 35.0 after 10 measurements

```
In [ ]: n = 100 # change as you wish
        k = n
        sum_n = 0
        while k > 0:
            sum_n += k
            k -= 1
        print("Sum of the first", n, "numbers is", sum_n)
```

Sum of the first 100 numbers is 5050

Iteration N times?

Iterating for N times is quite common in most programs. For this reason, Python provides:

- `for` statement
- `range(<number>)`

Let us see how they play together.

Loop: `for`

The statement `for` performs an iteration (i.e., execution of its body) *for* each *<data item>* available in the *<dataset>* specified after `in`.

```
for <data item> in <dataset> :  
    <instruction #1>  
    <instruction #2>  
    [...]  
    <instruction #3>
```

Remarks:

- *<data item>* will be a new variable that we define to represent the current item. Making it a variable, allow us to access it within the body of the `for`.
- *<dataset>* can be any piece of data that can be *iterated*. As we will see later, non-scalar data types are iterable. For now, we consider the iterable dataset generated by `range(<number>)`.

Loop: `for` + `range(<n>)`

`range(<n>)` generates an iterable dataset containing integers from `0` to `n - 1`, where `n` is a positive integer. We will embed into our `for` loop in this way:

```
for <data item> in range(<n>):  
    <instruction #1>  
    <instruction #2>  
    [...]  
    <instruction #3>
```

The body of the `for` will be executed exactly `n` times.

```
In [ ]: n = 100 # change as you wish
        sum_n = 0
        for k in range(n + 1): # remember that the last number is not included!
            sum_n += k
        print("Sum of the first", n, "numbers is", sum_n)
```

Sum of the first 100 numbers is 5050

range is a bit more powerful

`range(<start>, <stop>, <step>)` returns a sequence of intergers:

- starting from `<start>`
- incremented by `<step>`
- ending at `<stop>-1`

Caveats:

- `<stop>` is mandatory
- `<step>` is optional: if not provided, it defaults to 1
- `<start>` is optional: if not provided, it defaults to 0

```
In [ ]: sum_n = 0
start = 10 # we assume it is an even number
end = 101
for k in range(start, end, 2):
    sum_n += k
print(f"Sum of the even numbers in interval [{start},{end-1}] is {sum_n}")
```

Sum of the even numbers [10,100) is 2530

Loops: `for` + `string`

A `string` is an iterable sequence. We can thus use it within a `for` :

```
for <data item> in <string> :  
    <instruction #1>  
    <instruction #2>  
    [...]  
    <instruction #3>
```

The body of the `for` will be executed for each character of the `<string>`.

```
In [ ]: s1 = "Hello"  
        for c in s1:  
            print(c)
```

```
H  
e  
l  
l  
o
```

break statement

Within a `for` or `while` body, we can force the termination of the entire loop using `break`. This is often used to break out from a loop when a specific condition is met.

```
In [ ]: sum_n = 0
        pivot = 0
        limit = 50
        for x in range(100):
            sum_n += x
            if sum_n > limit:
                pivot = x
                break          # break the loop
        print(f"After summing the first {pivot} numbers, the sum is greater than 50")
```

After summing the first 10 numbers, the sum is greater than 50

In []:

```
x = 0
sum_n = 0
pivot = 0
limit = 50
while x < 100:
    sum_n += x
    if sum_n > limit:
        pivot = x
        break          # break the loop
    x += 1
print(f"After summing the first {pivot} numbers, the sum is greater than 50")
```

After summing the first 10 numbers, the sum is greater than 50

continue statement

Within a `for` or `while` body, we can arbitrarily move to the next iteration using `continue`. As for `break`, this is often used when a specific condition is met.

```
In [ ]: sum_n = 0
        for x in range(20):
            sum_n += x
            if sum_n % 13 != 0:
                continue # skip the rest of the body
            print("Current sum is multiple of 13: ", sum_n)
```

```
Current sum is multiple of 13: 0
Current sum is multiple of 13: 78
Current sum is multiple of 13: 91
```

We can refactor our loop body, making it conditional, avoiding to use `continue`. However, `continue` may improve the readability of our code.

```
In [ ]: sum_n = 0
x = 0
while x < 20:
    sum_n += x
    x += 1
    if sum_n % 13 != 0:
        continue    # skip the rest of the body
    print("Current sum is multiple of 13: ", sum_n)
```

```
Current sum is multiple of 13: 0
Current sum is multiple of 13: 78
Current sum is multiple of 13: 91
```

Nesting

We can nest conditionals and loops:

- loops within conditionals
- conditionals within loops
- loops within loops
- conditionals within conditionals

For instance:

```
In [ ]: for x in range(3):  
        for y in range(3):      # nested loop  
            if x == y:          # nested if  
                print(f"({x},{y})")
```

(0,0)

(1,1)

(2,2)

Lists

Lists: informal definition

A list:

- is an *ordered* sequence of data elements
- can contain data elements of *heterogeneous data types*
- is *mutable*, i.e., we can update its content (e.g., add/remove/replace elements)
- is denoted by squared brackets (`[` at the begin and `]` at the end), with elements separated by a comma `,`

For instance:

```
In [ ]: l = [1, 2, 3]
        print(l)
```

```
[1, 2, 3]
```

Lists: construction

```
In [ ]: l0 = []          # empty list
        l0 = list()     # alternative way to create an empty list
        print(l0)

        l1 = [1, 2, 3]   # list with three
                        # homogeneous elements
        print(l1)

        l2 = [1, "two", 3.0] # list with three
                        # heterogeneous elements
        print(l2)

        l3 = [1, [2, 3], 4] # list with three
                        # elements, one of which is a list
        print(l3)
```

```
[]
[1, 2, 3]
[1, 'two', 3.0]
[1, [2, 3], 4]
```


Lists: indexing

Similarly to a string, we can perform *indexing*:

```
In [ ]: l = [1, 'two', 3.0]

print(l[0]) # access the first element
print(l[1]) # access the second element
print(l[-1]) # access the last element
print(l[-2]) # access the second last element

i = 2
print(l[i]) # access the element at index i
```

1
two
3.0
two
3.0

Lists: slicing

Similarly to a string, we can perform *slicing*:

```
In [ ]: l = [1, 'two', 3.0]

print(l[0:2])    # access the first two elements
print(l[:2])     # starting index is 0
print(l[1:])     # ending index is the last element
print(l[-2:])    # access the last two elements

[1, 'two']
[1, 'two']
['two', 3.0]
['two', 3.0]
```

Slicing generates new and independent lists.

Lists: mutability

We can mutate a list:

Hello World

In []:

```
In [20]: l = [1, 'two', 3.0]
print("Original list:", l)

l[0] = "one" # the original list is modified
print("Updated list:", l)
```

Original list: [1, 'two', 3.0]

Updated list: ['one', 'two', 3.0]

If we want a *copy* of a list:

```
In [ ]: l2 = l.copy() # we create a new list with the same elements
l2[0] = 1 # we modify the new list
print("Original list:", l)
print("Copied and updated list:", l2)
```

```
l3 = l[:] # we create a new list with the same elements  
print("Another copy of the original list:", l3)
```

Original list: ['one', 'two', 3.0]

Copied and updated list: [1, 'two', 3.0]

Another copy of the original list: ['one', 'two', 3.0]

Lists: operators and operations

Given a list `l = [1, 2, 3]` and `l2 = [4]`:

Operation	Semantics	Result	In-place?
<code>len(l)</code>	length, i.e., number of items	<code>3</code>	N/A
<code>l + l2</code>	concatenation	<code>[1, 2, 3, 4]</code>	New list
<code>l * 2</code>	replication	<code>[1, 2, 3, 1, 2, 3]</code>	New list
<code>l.extend(l2)</code>	concatenation	<code>[1, 2, 3, 4]</code>	Original list
<code>l.append(4)</code>	append element	<code>[1, 2, 3, 4]</code>	Original list
<code>del(l[0])</code>	remove element at a given index	<code>[2, 3]</code>	Original list
<code>l.pop()</code>	remove the last element	<code>[1, 2]</code>	Original list
<code>l.remove(2)</code>	remove the first occurrence of an element	<code>[1, 3]</code>	Original list

Lists: try!

```
In [ ]: l = [1, 2, 1, 1]
l.append(4) # we add an element at the end
print(l)
l.pop()    # we remove the last element
print(l)
del(l[0])  # we remove the first element
print(l)
l.remove(1) # we remove the first occurrence of 1
print(l)
l.remove(1) # we remove the second occurrence of 1
print(l)
l.extend([5, 6]) # we add two elements at the end
l2 = l + [7, 8] # this does not mutate list l
print(l)
print(l2)
```

```
[1, 2, 1, 1, 4]
[1, 2, 1, 1]
[2, 1, 1]
[2, 1]
[2]
[2, 5, 6]
[2, 5, 6, 7, 8]
```

Lists are iterable

Using the `for` statement and the `in` operator we can iterate over a list:

```
In [ ]: l = [1, 2, 3]
        for x in l:
            print(x)
```

```
1
2
3
```

Or, if you want to get the index:

```
In [ ]: for i in range(len(l)):
        print(f"Value at index {i}: {l[i]}")
```

```
Value at index 0: 1
Value at index 1: 2
Value at index 2: 3
```

Lists: use of `enumerate`

A more compact version of the last example is done by using `enumerate(<dataset>)`:

```
In [ ]: l = [1, 2, 3]
        for i, x in enumerate(l):
            print(f"Value at index {i}: {x}")
```

```
Value at index 0: 1
Value at index 1: 2
Value at index 2: 3
```

`enumerate` returns at each iteration the current index and the current element from the iterable dataset.

Lists: iteration vs mutability

Be aware that you should not mutate a list while iterating over it:

```
In [ ]: l = [1, 2, 3]
        for x in l:
            print(x)
            l.pop()
```

```
1
2
```

This code does not print the elements of the original list! Indeed, `pop()` is removing elements from the list, making hard to understand the iteration workflow.

Lists and strings

Common *conversions* between lists and strings:

```
In [ ]: s = "Hello"
l = list(s) # we convert the string into a list
print(l)

s = "Do you want a coffee?"
l2 = s.split(" ") # we split the string into a list
                        # using the space character
                        # as separator
s2 = " ".join(l2) # we join the list into a string
                        # using the space character
                        # as separator

print(l2)
print(s2)
```

```
['H', 'e', 'l', 'l', 'o']
['Do', 'you', 'want', 'a', 'coffee?']
Do you want a coffee?
```

List: sorting

Two ways to sort a list:

```
In [ ]: l = [2, 3, 1, 4]
l2 = sorted(l) # we create a new list with the elements sorted
print(l2)
l.sort()      # or... we sort the original list
print(l)
```

[1, 2, 3, 4]

[1, 2, 3, 4]

To mutate or not to mutate, that is the question

Lists: aliasing

Since copying groups of data could be expensive (e.g., think about a list with 100k elements), by default, Python does not make a copy of a list when we use the assignment operator (=). It makes an alias:

```
In [ ]: l = [1, 2, 3]
        l2 = l           # we create an alias to the original list
        l2.append(4)      # we modify the alias
        print(l)          # the original list is modified!

[1, 2, 3, 4]
```

We can check whether two variables are aliases to the same data using the `is` operator:

```
In [ ]: print(l2 is l)
        print(l is l2)

True
True
```

Lists: cloning

We you want to get a copy of a list:

```
In [ ]: l = [1, 2, 3]
l2 = l.copy()    # we create a new list with the same elements
l2.append(4)      # we modify the new list
l3 = l[:]         # alternative way to create a new list with the same elements
l3.append(5)      # we modify the new list
l4 = list(l)      # alternative way to create a new list with the same elements
l4.append(6)      # we modify the new list
print(l)
print(l2)
print(l3)
print(l4)
```

```
[1, 2, 3]
[1, 2, 3, 4]
[1, 2, 3, 5]
[1, 2, 3, 6]
```

Tuples

Tuples: informal definition

A tuple is:

- an immutable list
- is denoted by squared parantheses ((at the begin and) at the end), with elements seperated by a comma ,

For instance:

```
In [ ]: t1 = (1, 2, 3)           # tuple with three homogeneous elements
        t2 = (1, 'two', 3.0)    # tuple with three heterogeneous elements
        print(t1)
        print(t2)
```

```
(1, 2, 3)
(1, 'two', 3.0)
```

Tuples: construction

We can build a tuple in different ways:

```
In [ ]: t1 = ()          # empty tuple
        t2 = tuple()    # alternative way to create an empty tuple
        print(t1)
        print(t2)

        t3 = (1,)       # tuple with one element... notice the comma!
        i4 = (1)         # this is not a tuple, it is an int!
        t4 = tuple([1]) # tuple created from a iterable data structure
        print(t3)
        print("i4 is", i4, "of data type", type(i4))
        print(t4)

        t5 = (1, 'two', 3.0)
        t6 = tuple([1, 'two', 3.0]) # tuple created from a list
        print(t5)                  # same for t6
```

```
()
()
(1,)
i4 is 1 of data type <class 'int'>
(1,)
(1, 'two', 3.0)
```


Tuples are similar to lists

These two data structure have a lot in common:

```
In [ ]: t = (1, 'two', 3.0)

# indexing
print(t[0])      # access the first element

# slicing
print(t[:2])     # access the first two elements

# non in-place operations and operators
print(len(t))    # length of the tuple
print(t * 3)     # repeat the tuple three times
print(t + (5,))  # concatenate another tuple to the tuple
print("two" in t) # check if "two" is in the tuple
print(2 in t)    # check if "two" is in the tuple
```

1
(1, 'two')
3
(1, 'two', 3.0, 1, 'two', 3.0, 1, 'two', 3.0)
(1, 'two', 3.0, 5)
True
False

Tuples are different from lists

Differently from lists, tuples are immutable:

```
In [ ]: t = (1, 'two', 3.0)
t[0] = 2    # this will raise an exception
```

```
-----
--
TypeError                                Traceback (most recent call las
t)
/tmp/ipykernel_794386/3993153899.py in <module>
      1 t = (1, 'two', 3.0)
----> 2 t[0] = 2    # this will raise an exception

TypeError: 'tuple' object does not support item assignment
```

What if you *really* need to mutate a tuple?

The trick is to convert it to a list, mutate the list, and convert the list to a tuple:

```
In [ ]: t = (1, 'two', 3.0)
l = list(t)      # we convert the tuple into a list
l.append(4)      # we add an element at the end
t2 = tuple(l)    # we convert the list into a tuple
print(t2)
```

```
(1, 'two', 3.0, 4)
```

Tuples may contain mutable data!

The immutability property covers only the *container* and not its *content*.

```
In [ ]: t = (1, 2, [0, 0]) # the third element is a list
          # which is mutable data structure
print(t)
t[2][0] += 1 # this is allowed
             # because we are modifying the list
print(t)
```

```
(1, 2, [0, 0])
(1, 2, [1, 0])
```

Tuples are iterable

As lists, tuples are iterable

```
In [ ]: t = (1, 'two', 3.0)
        for x in t:
            print(x)
```

```
1
two
3.0
```

Sets

Sets: informal definition

A set:

- is an unordered sequence of data elements with no repetition
- can contain data elements of *heterogeneous data types*
- is *mutable*, i.e., we can update its content (e.g., add/remove elements)
- is denoted by curly brackets (`{` at the begin and `}` at the end), with elements separated by a comma `,`

For instance:

```
In [ ]: s = {4, 5.0, "six"}
print(s)  # the output does not respect the order
          # in which the elements were inserted
          # this is because a set is unordered

{'six', 4, 5.0}
```

Sets: construction

Different ways:

```
In [ ]: s1 = set()    # empty set
s2 = {}          # strangely, this is not a set!
print(s1)
print(s2)
print(type(s2))

s3 = {1, 2, 3}    # set with three elements
s4 = set({1, 'two', 3.0}) # set created from a list
print(s3)
print(s4)
```

```
set()
{}
<class 'dict'>
{1, 2, 3}
{1, 'two', 3.0}
```


Sets do not contain repetitions

```
In [ ]: sss = "Hello"
```

```
In [ ]: s = {1, 1, 1}
print(s)    # the output is {1} because a set
            # does not allow duplicates
```

```
{1}
```

```
In [ ]: s = set(["ciao", 2.0, "ciao", 2.0])
print(s)    # the output is {'ciao', 2.0} because a set
            # does not allow duplicates
```

```
{2.0, 'ciao'}
```

How does Python check for repetitions?

Python exploits data *hashing* to efficiently check the repetitions in a set. We do not talk in detail about *hashing* at this point of the course. What you need to know is that only immutable data is *hashable*. Hence, mutable data cannot be inserted into a set:

```
In [ ]: s1 = {1, 2.0, tuple([1, 2])}    # this is fine
        print(s1)                    # because a tuple is immutable

        s2 = {1, 2.0, [1, 2]}        # this is not allowed
                                         # because a list is mutable
```

```
{1, 2.0, (1, 2)}
```

```
-----
--
TypeError                                Traceback (most recent call las
t)
/tmp/ipykernel_794386/1262637465.py in <module>
      2 print(s1)
      3
----> 4 s2 = {1, 2.0, [1, 2]}    # this is not allowed
      5                        # because a list is mutable

TypeError: unhashable type: 'list'
```

Sets: operators and operations

Given `s1 = {1, 2, 3}` and `s2 = {3, 4}`:

Operation	Semantics	Result	In-place?
<code>len(s1)</code>	number of items	<code>3</code>	N/A
<code>1 in s1</code>	membership	<code>True</code>	N/A
<code>5 not in s1</code>	membership	<code>True</code>	N/A
<code>s1.add(4)</code>	add an element	<code>{1, 2, 3, 4}</code>	Original set
<code>s1.update([3, 4])</code>	union with any iterable data structures	<code>{1, 2, 3, 4}</code>	Original set
<code>s1.remove(3)</code>	remove an element (exception if missing)	<code>{1, 2}</code>	Original set
<code>s1.discard(3)</code>	remove an element (no exception if missing)	<code>{1, 2}</code>	Original set
<code>s1.pop()</code>	remove one (arbitrary) element and returns it	<code>{1}</code>	Original set
<code>s1.clear()</code>	remove all elements	<code>{}</code>	Original set

Sets: operators and operations

Given $s1 = \{1, 2, 3\}$ and $s2 = \{3, 4\}$:

Operation	Semantics	Result	In-place?
<code>s1.union(s2)</code>	union	$\{1, 2, 3, 4\}$	New set
<code>s1 \ s2</code>	union	$\{1, 2, 3, 4\}$	New set
<code>s1.intersection(s2)</code>	intersection	$\{3\}$	New set
<code>s1 & s2</code>	intersection	$\{3\}$	New set
<code>s1.difference(s2)</code>	difference	$\{1, 2\}$	New set
<code>s1 - s2</code>	difference	$\{1, 2\}$	New set

Sets: try!

```
In [152]: s1 = {1, 2, 3}
s2 = {3, 4, 5}
print(s1)
s1.add(4)
print(s1)
print(s1.union(s2))      # union of two sets
print(s1.intersection(s2)) # intersection of two sets
print(s1.difference(s2))  # difference of two sets
```

```
{1, 2, 3}
{1, 2, 3, 4}
{1, 2, 3, 4, 5}
{3, 4}
{1, 2}
```

Why do we need sets?

Since they are efficient at checking repetitions thanks to *hashing*, we use sets when we want to check whether we have already met a specific piece of data.

```
In [ ]: s = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10}
is_in_the_set = 5 in s
print(is_in_the_set)

l = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
is_in_the_list = 5 in l # this is way slower than the previous one
print(is_in_the_list)
```

True
True

Dictionaries

Dictionary: informal definition

Features of a dictionary:

- it maps a given `<key>` to a given `<value>`
- `<key>` can be any immutable value (actually, it must be a *hashable* value)
- `<value>` can be anything
- the (`<key` , `<value`) pairs are sorted without a specific ordering (it was implementation specific, while in the last releases it is ordered but you should not rely on it)
- it is *mutable*, i.e., we can update its content (e.g., add/remove elements)
- it is denoted by curly brackets (`{` at the begin and `}` at the end), with pairs separated by a comma `,` where each pair has `:` to separate the key from the value

For instance:

```
In [ ]: d = {1: 'one', 2: 'two', 3: 'three'} # dictionary with three key-value pairs
print(d)

{1: 'one', 2: 'two', 3: 'three'}
```


Dictionaries: construction

Different ways:

```
In [ ]: d1 = {}           # empty dictionary
        d2 = dict()      # alternative way to create an empty dictionary
        print(d1)
        print(d2)
```

```
{}
```

```
{}
```

```
In [ ]: d3 = {1: 'one', 'two': 2, 3.0: 'three'} # dictionary with:
        # - different data types for the keys
        # - different data types for the values
        print(d3)
```

```
{1: 'one', 'two': 2, 3.0: 'three'}
```

Dictionaries: add, update, or delete a pair

The assignment operator `=` adds/update a pair in the dictionary, while `del` remove an existing pair from it:

```
In [ ]: d = {1: 'one', 'two': 2}
print(d)
d[3.0] = "tree" # we add a new key-value pair
print(d)
d[1] = "uno"    # we update the value of an existing key
print(d)
del(d[1])      # we remove a key-value pair
print(d)
```

```
{1: 'one', 'two': 2}
{1: 'one', 'two': 2, 3.0: 'tree'}
{1: 'uno', 'two': 2, 3.0: 'tree'}
{'two': 2, 3.0: 'tree'}
```

Dictionary: a **single** value for each key

Given a key, a dictionary only keeps track of a single value for it:

```
In [ ]: d = dict()
d['a'] = 1
d['a'] = 2  # we update the value of an existing key
            # the previous value is lost!
print(d)

{'a': 2}
```

Dictionary: retrieve value associated to a key

Quite easy:

```
In [ ]: d = {1: 'one', 'two': 2}
print(d[1])      # access the value associated with the key 1
print(d['two'])  # access the value associated
                 # with the key 'two'

print(d[2])      # this will raise an exception
```

```
one
2
```

```
-----
--
KeyError                                Traceback (most recent call las
t)
/tmp/ipykernel_1885124/3592842802.py in <module>
      4             # with the key 'two'
      5
----> 6 print(d[2])    # this will raise an exception

KeyError: 2
```

Dictionaries: operators and operations

Given `d = {1: 'one', 'two': 2}`:

Operation	Semantics	Result
<code>len(d)</code>	number of key-value pairs	<code>2</code>
<code>1 in d</code>	membership: whether there is pair with a specific key	<code>True</code>
<code>2 not in d</code>	membership: whether there is not a pair with a specific key	<code>True</code>
<code>d.keys()</code>	(unsorted) list of keys	<code>[1, 'two']</code>
<code>d.values()</code>	(unsorted) list of values	<code>[2, 'one']</code>
<code>d.items()</code>	(unsorted) list of pairs	<code>[(1, 'one'), ('two', 2)]</code>

NOTE: `d.values()`, `d.values()`, and `d.items()` do not actually returns *standard* lists (due to optimization reasons) but you can treat their result as list.

Dictionaries: operators and operations (cont'd)

Given `d = {1: 'one', 'two': 2}`:

Operation	Semantics	Result	In-place?
<code>d.get(3)</code>	return the value associated with a key (if it exists) or <code>None</code> (without raising an exception)	<code>None</code>	N/A
<code>d.pop(1)</code>	remove the pair with the given key and return its value; if the key is not present, then exception	<code>'one'</code>	Yes
<code>d.clear()</code>	remove all pairs	<code>{}</code>	Yes
<code>d.update({'3': 3.0})</code>	add pairs from another dictionary	<code>{1: 'one', 'two': 2, '3': 3.0}</code>	Yes

Dictionaries are iterable

By default, iterating over a dictionary with a `for` gives us its keys:

```
In [ ]: d = {1: 'one', 2: 'two', 3: 'three'}  
for k in d:  
    print(f"Key: {k}, Value: {d[k]}")
```

```
Key: 1, Value: one  
Key: 2, Value: two  
Key: 3, Value: three
```

Which is equivalent to:

```
In [ ]: for k in d.keys():  
    print(f"Key: {k}, Value: {d[k]}")
```

```
Key: 1, Value: one  
Key: 2, Value: two  
Key: 3, Value: three
```

Dictionaries are iterable (cont'd)

If want to get both the key and value while iterating:

```
In [ ]: d = {1: 'one', 2: 'two', 3: 'three'}  
for k, v in enumerate(d):  
    print(f"Key: {k}, Value: {v}")
```

Key: 0, Value: 1

Key: 1, Value: 2

Key: 2, Value: 3

Which is equivalent to:

```
In [ ]: for k, v in d.items():  
        print(f"Key: {k}, Value: {v}")
```

Key: 1, Value: one

Key: 2, Value: two

Key: 3, Value: three

Dictionaries are iterable (cont'd)

If want only wants to iterate only over its values:

```
In [ ]: for v in d.values():  
        print(f"Value: {v}")
```

Value: one

Value: two

Value: three

Functions

Make the code reusable

It is not convenient in programming to copy&paste code around. Most of the time we want to perform similar tasks but in slightly different order.

In other words, we need to introduce the concept of a function!

Notice that unctions can come from three sources:

1. Our code: we write them!
2. Python: they are built-in from the language
3. External packages: we can use functions implemented by the Python community after installing the related external package(s).

Function: informal definition

A function:

- takes zero or more data inputs: *arguments* (or *parameters*)
- returns zero or more data outputs: *return values*
- performs a task, processing the *arguments* (if any), possibly producing side effects (e.g., writing a file) or/and returning some *values*
- has a *name*
- has a function *body*, i.e., the nested group of instructions defining its behavior
- is not executed unless we explicitly call it using its *name* (passing its *arguments* if required)

Function: definition

To define a new function, we use the `def` statement:

```
def <NAME> ( <PARAM1> , <PARAM2> , [...] ):  
    <instruction #1>  
    <instruction #2>  
    [...]  
    <instruction #N>  
    return <RET_VALUES>
```

Remarks:

- the *<NAME>* follows the same rules as for variable naming (and its conventions)
- the parameters are optional (e.g., *<PARAM1>*)
- when writing the function, we do not know the values of the parameters
- the body of the functions is indented to mark its begin and end
- if the function wants to return one or more values (*<RET_VALUES>*), the function can use the `return` statement

Function: invocation

To execute a function, we call or *invoke* it:

`<NAME> (<ARG1> , <ARG2> , [...])`

At this time, we will decide the values of the function parameters (e.g., `<PARAM1>`) by passing some arguments (e.g., `<ARG1>`).

Remarks:

- *local scope*: if a function defines a new variable, it generates a local variable which is not observable outside of the function.
- *global scope*: any variable defined outside functions are global variables

Function: examples

Let us consider a function with:

- no parameters
- no return values
- no side effects

```
In [ ]: def do_nothing_valuable(): # function definition
        a = 1 # local variable a
        a = a + 1 # new local variable a

do_nothing_valuable() # first invocation
do_nothing_valuable() # second invocation
# print(a) # this will raise an exception:
# we cannot access a local variable
# because we are in the global scope
```

This function is NOT useful since its effects are not observable.

Function: examples (cont'd)

- no parameters
- no return value
- some side effects (i.e., print something on the screen)

```
In [ ]: def say_hello():    # function definition
        s = "World"      # local variable s
        print("Hello,")  # side effect
        print(f"{s}!")   # side effect

say_hello() # first invocation
say_hello() # second invocation
# print(s)  # this will raise an exception:
            # we cannot access a local variable
            # because we are in the global scope
```

```
Hello,
World!
Hello,
World!
```

Since we called it twice, we get twice its side effects.

Function: examples (cont'd)

Function with:

- one parameter
- no return value
- no side effects

```
In [ ]: def increment(x): # function definition
        x = x + 1         # the assignment is creating a new variable x
                           # this new variable is local to the function
                           # and does not affect the variable x outside the function

x = 1                     # global variable x
increment(x) # first invocation
increment(x) # second invocation
print(f"global x: {x}")

global x: 1
```

Remarks:

- This function is NOT useful since its effects are not observable.
- Any defined variable within the function is local to it.

Function: examples (cont'd)

- one parameter
- one return value
- no side effects

```
In [ ]: def increment(x): # function definition
        x = x + 1      # local variable x
        return x       # we return the value of the local x

x = 1      # global variable x
y = 0      # global variable y
y += increment(x) + increment(x) # two invocations
print(f"global x: {x}")
print(f"global y: {y}")
```

```
global x: 1
global x: 4
```

Remarks:

- This function is valuable only if we use its return value!
- Outside of a function, we can access the *value* of a local variable only if the local variable is returned by the function.

Function: examples (cont'd)

Function with:

- one parameter
- one return value
- side effects

```
In [ ]: def increment_and_print(x): # function definition
        x = x + 1 # the assignment is creating a new variable x
        # this new variable is local to the function
        # and does not affect the variable x outside the function
        print(f"local x: {x}") # we print the new value of x
        return x # we return the new value of x

x = 1
y = 0
y += increment_and_print(x) + increment_and_print(x) # two invocations
print(f"global x: {x}")
print(f"global x: {y}")
```

```
local x: 2
local x: 2
global x: 1
global x: 4
```

Function parameters: required by default

Each parameter:

- is required by default:
 - the caller must pass it...
 - ...otherwise we get an exception

```
In [ ]: def add(a, b, c): # three required parameters
        return a + b + c

print(add(1, 2, 3)) # we must pass three arguments
```

6

```
In [ ]: print(add(1, 2)) # this will raise an exception
```

```
-----
--
TypeError                                Traceback (most recent call las
t)
/tmp/ipykernel_2327557/3570833341.py in <module>
----> 1 print(add(1, 2)) # this will raise an exception

TypeError: add() missing 1 required positional argument: 'c'
```

Function parameters: optional by choice

- can be made optional:
 - the functions defines a default value
 - the caller may omit to pass it
 - if the caller pass it, the caller's value is used in place of the default one

```
In [ ]: def add(a, b=2, c=3): # three required parameters
        return a + b + c

print(add(1))           # we must pass the first argument
                        # while the other two have default values
print(add(1, 4))        # we can pass the second argument
print(add(1, 4, 5))     # we can pass all the arguments
```

6
8
10

Function parameters: optional by choice (cont'd)

However, optional parameters must come after required parameters:

```
In [ ]: def add(a=0, b=2, c): # incorrect because required argument
        return a + b + c   # is after optional ones
```

```
File "/tmp/ipykernel_2327557/2774706269.py", line 1
```

```
def add(a=0, b=2, c): # incorrect because required argument
```

```
^
```

```
SyntaxError: non-default argument follows default argument
```

Function arguments ordering: positional by default

When invoking a function, we must pass the arguments consistently with the parameters ordering:

```
In [ ]: def say_hello(name, age): # first parameter is name, second is age
        print(f"{name} is {age} years old")

say_hello(45, "Francesco") # this does not work as expected!
                        # because we swapped the arguments!
```

45 is Francesco years old

We can use parameters names to get the correct mapping even when ignoring the expected ordering:

```
In [ ]: say_hello(age=45, name="Francesco") # this works as expected!
```

Francesco is 45 years old

Built-in functions

Python provides several built-in functions:

- type conversion: e.g., `int(x)`, `float(x)`, `str(x)`, `tuple(it)`, `list(it)`
- input and output: e.g., `print(s)`
- utils: e.g., `range(n)`, `enumerate(l)`, `sorted(l)`
- mathematical: see next slide(s)
- ...and many additional ones!

We will cover the most common ones over the weeks.

Built-in mathematical functions

Function	Semantics	Example	Example Result
<code>abs(x)</code>	Absolute integer value	<code>abs(-5)</code>	5
<code>round(x, ndigits=None)</code>	Return number rounded to <code>ndigits</code> precision after the decimal point. If <code>ndigits</code> is omitted, it returns the nearest integer to its input	<code>round(5.256, 1)</code>	5.3
<code>min(a, b)</code>	Min value between <code>a</code> and <code>b</code>	<code>min(1, 2)</code>	1
<code>min(L)</code>	Min value within an iterable	<code>min([1, 2, 3])</code>	1
<code>max(a, b)</code>	Max value between <code>a</code> and <code>b</code>	<code>min(1, 2)</code>	2
<code>max(L)</code>	Max value within an iterable	<code>min([1, 2, 3])</code>	3
<code>sum(L)</code>	Sum over an iterable	<code>min([1, 2, 3])</code>	6
<code>pow(a, b)</code>	Exponentiation. Same as <code>a**b</code> but more efficient	<code>pow(5, 2)</code>	25

Modules and Packages

Each Python file is a module

In Python, each `.py` file is called a *module* and we can `import` its functions from other modules. For instance, let suppose we have `my_math.py` containing:

```
def custom_add(a, b):  
    return a + b  
  
def custom_sub(a, b):  
    return a - b
```

Then, in other file, e.g., `test.py`, we can reuse the functions from `my_math.py`:

```
import my_math # import the entire module  
  
# we can call a function from an imported module using the dot notation:  
# <module_name>.<function_name>  
print(my_math.custom_add(10, 20))
```

Or:

```
from my_math import custom_add, custom_sub # import specific functions  
from my_math import *                     # import all functions  
  
# we can call the function(s) directly!  
print(custom_add(10, 20))
```

Notice that `import` will look for the imported module in the current directory (of the module importing it) or a few fixed locations on our filesystem (e.g., a few system directory related to Python).

Package: collection of modules

Often, we want to organize our functions in different modules but group them together into the same logical *container*. This brings the idea of a *package*.

For instance, let suppose we have in the `mypackage` :

- `my_math.py` : functions `custom_add` and `custom_sub`
- `my_utils.py` : functions `say_hello`

To make it a package, we need to add within the same directory a file called `__init__.py` :

```
from . import my_math
from . import my_utils
```

Then, any other module can import the package:

```
import mypackage # entire package
print(mypackage.my_math.custom_add(10, 20))
# or...
from mypackage import my_math # import module from package
print(my_math.custom_add(10, 20))
# or...
from mypackage import * # all modules from the package
print(my_math.custom_add(10, 20))
```

Again, `import` will look for our package in the current directory and specific system directories.

Python pre-installed packages

Python ships with many pre-installed packages:

- `string`: additional string functions
- `math`: additional mathematical functions
- `io`: utils to read and write files
- `random`: pseudo-random number generator
- `sys`: check execution environment aspects

We will cover their interesting bits when needed. For instance:

```
In [ ]: import math
print(f"Square root of 9 is {math.sqrt(9)}, while pi is {math.pi}")
```

Square root of 9 is 3.0, while pi is 3.141592653589793

However, there is way more in `math`: `acos`, `acosh`, `asin`, `asinh`, `atan`, `atan2`, `atanh`, `ceil`, `comb`, `copysign`, `cos`, `cosh`, `degrees`, `dist`, `e`, `erf`, `erfc`, `exp`, `expm1`, `fabs`, `factorial`, `floor`, `fmod`, `frexp`, `fsum`, `gamma`, `gcd`, `hypot`, `inf`, `isclose`, `isfinite`, `isinf`, `isnan`, `isqrt`, `lcm`, `ldexp`, `lgamma`, `log`, `log10`, `log1p`, `log2`, `modf`, `nan`, `nextafter`, `perm`, `pi`, `pow`, `prod`, `radians`, `remainder`, `sin`, `sinh`, `sqrt`, `tan`, `tanh`, `tau`, `trunc`, `ulp`.

Install third-party packages

We can easily install third-party packages from the community using `pip` from the terminal:

```
pip install <package>
```

