L1S1

Python Basics

PALISSON Antoine

TABLE OF CONTENTS

TABLE OF CONTENTS

Introduction

What is Python ?

Python is a very simple **Programming Language**, that has a very straightforward syntax. Python is **dynamically-typed** and **garbage-collected**.

It is the most popular Programming Language. However, most of the other languages are different.

What does it look like ?

```
var_1 = 1
var_2 = 10.0
var_sum = var_1 + var_2
```

```
if var_1 > var_2 :
  print("Hello World")

else :
  print("World Hello")
```

```
def function(x):
  return x**2
```

Applications

Usages

Python is a very general language that can be used for many purposes :

- Developing websites
- Building software
- Task automation
- Data analysis
- Data visualization
- Artificial Intelligence

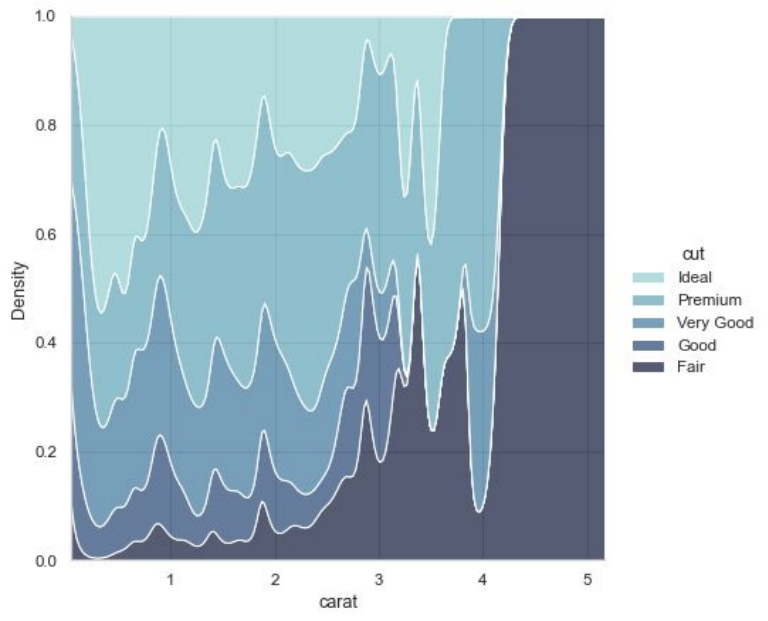
What is its main application ?

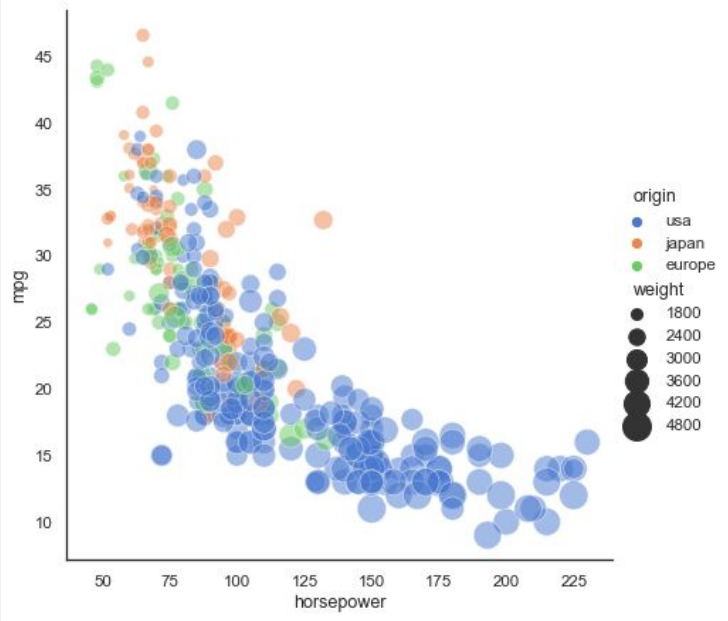
Data Analysis

	A	B	C	D	E	F	G	H	I	J
2000-01-03 00:00:00	-20.1863	-30.916	58.9187	99.0353	36.5655	-18.0928	-258.852	114.065	-145.543	83.2085
2000-01-04 00:00:00	50.7861	34.1686	-39.8916	-156.354	68.2068	97.8078	-14.9382	47.9822	-240.239	-95.4873
2000-01-05 00:00:00	189.422	-15.5873	40.6291	-27.5119	-269.28	37.311	-148.209	-104.216	-32.8902	-63.2067
2000-01-06 00:00:00	190.422	172.06	-190.506	-123.561	-46.0929	-10.5616	0.047801	0.846932	94.037	50.7201
2000-01-07 00:00:00	142.145	-122.872	132.62	15.8588	-379.276	4.57074	51.0181	42.3276	-61.3839	-156.779
2000-01-10 00:00:00	-35.3665	-121.451	133.642	-28.563	-41.1246	120.592	-1.0356	-39.1903	7.7991	140.851
2000-01-11 00:00:00	13.3999	-18.9973	32.5474	58.6557	22.2628	57.4611	18.2556	102.327	93.5969	125.019
2000-01-12 00:00:00	-146.557	115.329	-236.82	46.07	190.868	-47.7942	80.1292	-138.331	223.69	-49.9104
2000-01-13 00:00:00	58.4828	-157.913	30.8043	-34.0038	134.189	-118.555	18.7315	5.72604	29.5194	-47.3357
2000-01-14 00:00:00	190.293	-188.671	120.43	-96.7293	-36.7141	-47.8564	107.547	-47.1704	-105.001	-68.7717
2000-01-17 00:00:00	112.167	7.06374	85.8183	-44.1304	240.018	53.5899	-199.769	49.3378	39.1819	13.8417
2000-01-18 00:00:00	126.355	116.551	-60.125	27.4333	-77.5145	-85.0603	158.785	119.515	-49.4693	-10.4454
2000-01-19 00:00:00	-132.838	57.1512	-102.478	-106.242	4.6459	1.26142	242.231	-21.5105	159.697	30.9611
2000-01-20 00:00:00	-177.494	164.65	-25.3784	121.331	56.275	-190.621	20.294	39.5168	-136.713	137.953
2000-01-21 00:00:00	-32.618	-186.976	-146.837	-101.498	-103.488	-73.2039	-16.4683	-16.8418	244.662	219.604

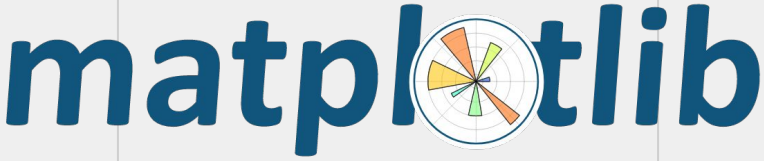
Data Analysis

Data Visualization

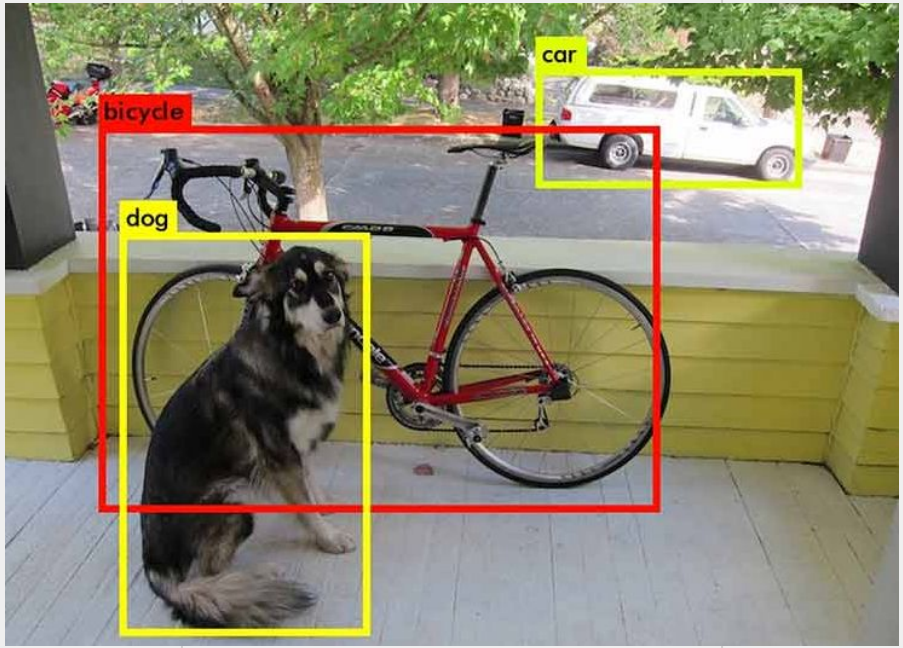




Data Visualization



Artificial Intelligence



Artificial Intelligence

TensorFlow

First Steps

How to use Python ?

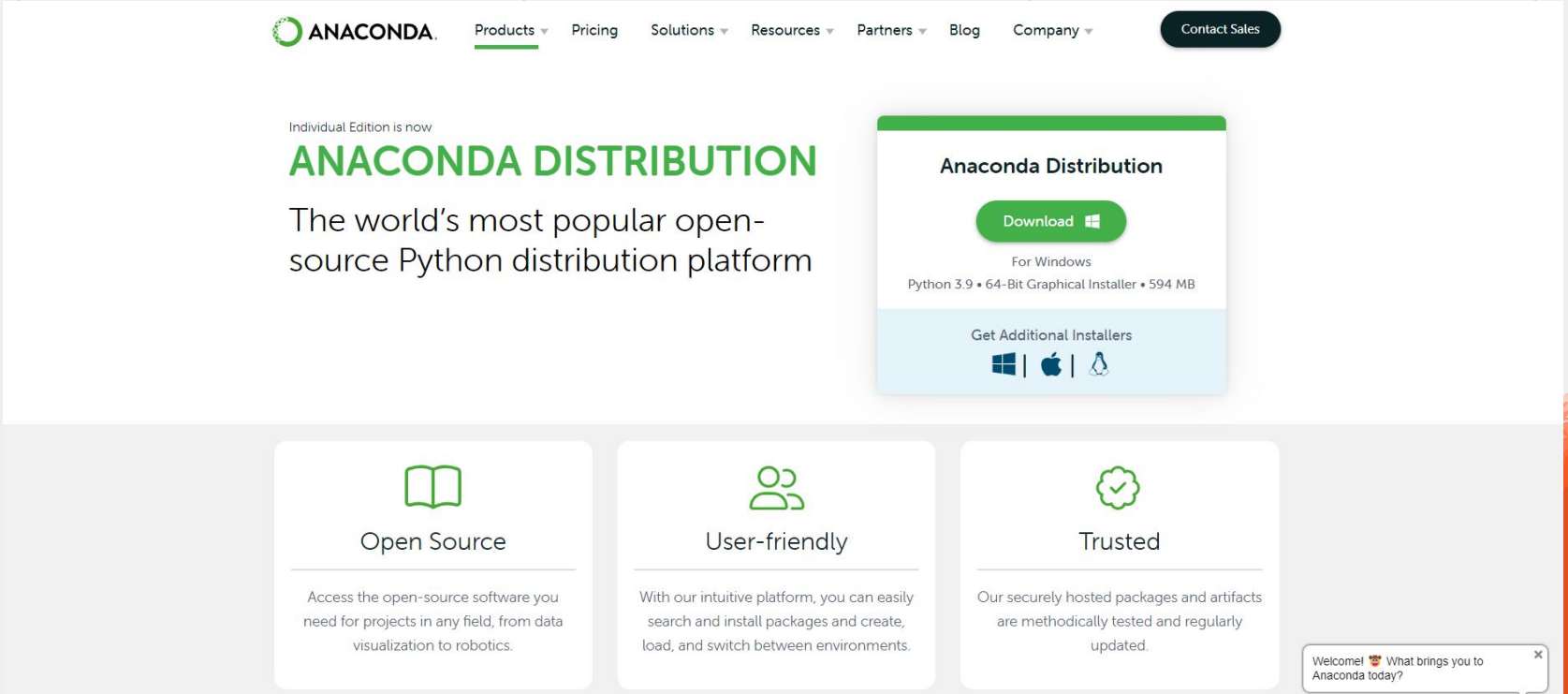
To use a Programming Language one should :

1. Install the **Programming Language**
2. Install an **Integrated Development Environment**
3. Start coding !

And , of course … you should also learn its syntax before starting coding !

Installation

https://www.anaconda.com/products/distribution



Python IDE

Some of the most known Python IDEs :

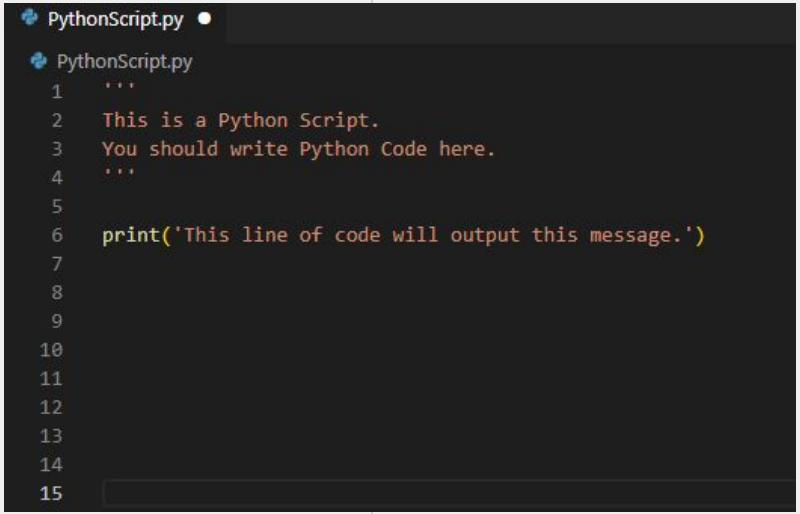
- PyCharm
- Visual Studio Code
- Spyder
- Jupyter

General Purpose Programming IDEs.
Python files are Scripts

Used for Data Science/Analysis
Python files are Notebooks

Python Scripts

Python scripts are files that are intended to be run directly. Each execution will go through **every line of code**.



PythonScript.**py**

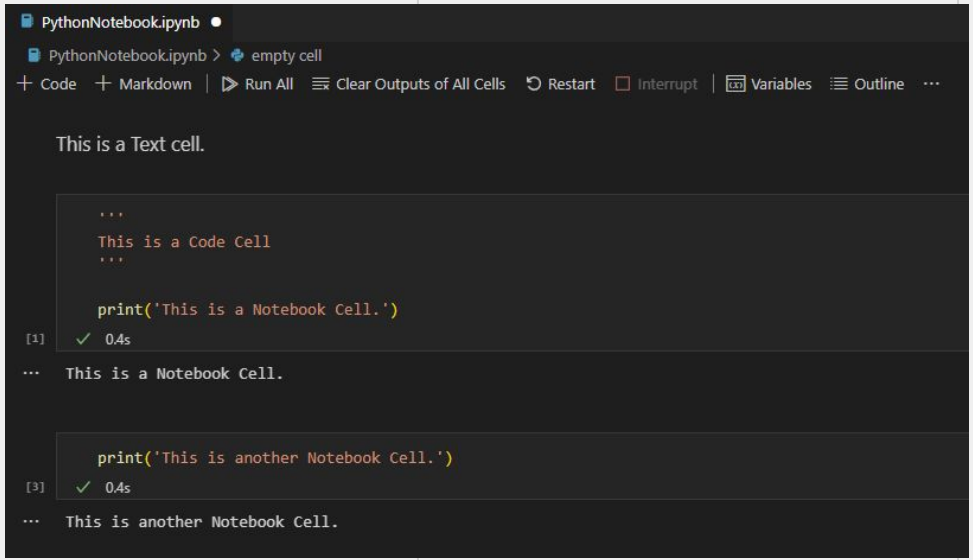
Python Notebook

Python Notebook are very similar to scripts but they are divided into multiple cells of code. **Each cell can be run independently.**

Cell n°1

Cell n°2

Cell n°3



PythonNotebook.**ipynb**

Variables & Types

Variables

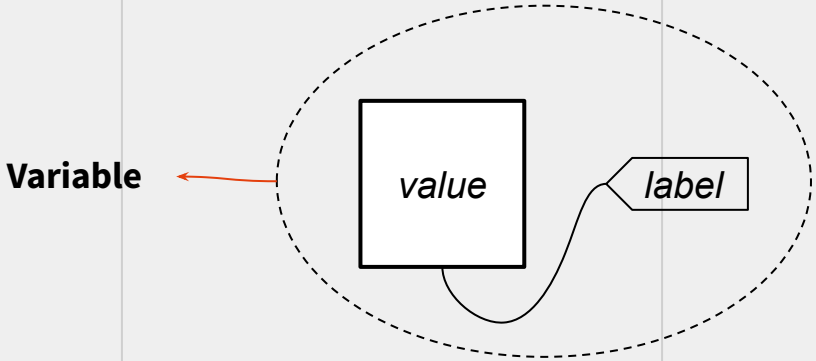
In programming, **a variable is where values are stored**.
These values can come from many sources :

- From a database
- From the user
- From the program

These variables can be of many types : numbers, texts …

Variables (2)

A variable is like a value in a box with a label.



Variables (3)

Before using a variable, the box and the label must be created.
This process is called :

Declaring a Variable

The label of the variable can be anything composed of numbers and texts but spaces and special characters must be avoided.

The box should contain the information relative to the memory needed for the value by specifying the type. However, this process is automatically handled in Python.

Declaring a variable

To declare a variable, a value should be assigned to it.

In Programming, variables can change over time between program states. Hence, in most programming language x = y means :

take the value stored at address y and assign it to the location with address x

In Mathematics, variables can change between similar function definitions and x = y means :

x equals y

Declaring a variable (2)

In Python, it is a bit different because it works with objects. In fact, Python is a highly object-oriented language and virtually every item of data in a Python program is an object of a specific type or class.

Thus, in Python x=y means:

take the object y and point it to the object where x is pointing

When the number of references to an object drops to zero, it is no longer accessible. At that point, its lifetime is over. Python will eventually notice that it is inaccessible and reclaim the allocated memory so it can be used for something else. This process is referred to as **garbage collection**.

Type of Variables

There are 4 main types of variable :

- Integers
- Floats
- Strings
- Booleans

```
variable = 1
```

```
variable = 3.14159
```

```
variable = "This is a string"
```

```
variable_1 = True
variable_2 = False
```

Integers & floats

Each variable value of type **integer** or **float** has a subtype that defines the range of values the memory can fit.

	Integers	Floats
16 bits	-32 768 to +32 767	4 digits precision
32 bits	-2 147 483 648 to +2 147 483 647	8 digits precision
64 bits	-9 223 372 036 854 775 808 to +9 223 372 036 854 775 807	17 digits precision

Strings

Each variable value of type string can contain **characters** (letters, numbers, special characters …)

Be careful to never :

- Get a **number** and a **number IN a string** mixed up.

  123 is different from “123”

- Mistake a **variable label/name** with a **variable value of type string**.

Booleans

Each variable value of type boolean is either **True** or **False**.

It highly improves the algorithm readability and allows to write conditional structures.

Python Variable

In Python, a variable value can be accessed and shown with the following function :

```
variable = 1
print(variable)
```

A function can also be used to access the type of a variable :

```
variable = 1
type(variable)
```

Python Variable (2)

Variable value types can be changed into other types :

```
variable = 1

# to string
str(variable)

# to float
float(variable)

# to integer
int(variable)
```

Variables & formula

Variable values can be the result of a formula.
Of course, variable values can also be used in formulas.

```
variable_1 = (5 + 10) / 10
variable_2 = variable_1 +10
```

Numerical Operators

Python can be used as a **calculator**.
Python contains **7 numerical operators** that can be used in formula :

Addition	+
Subtraction	-
Multiplication	*
Exponentiation	**
Division	/
Floor Division	//
Modulus	%

String Formatting

Python allows to format/customize strings in three different ways.

“Hello, John”

```
name = "John"

"Hello, %s" % name

"Hello, {}".format(name)

f"Hello, {name}"
```

String Formatting (2)

The `format()` method accepts multiple arguments. Everything that is not inside the curly braces is considered as literal (*i.e. as a string*).

```
name = "John"
age = 23

"{} is {} years old.".format(name, age)

"{name} is {age} years old.".format(name="John",
                                    age=23)
```

String Formatting (3)

Python formatting method has options to **Pad**, **Truncate** and **Transform** the strings.

`{:<10}`	Pad the string on the **right side** up to 10 characters
`{:>10}`	Pad the string on the **left side** up to 10 characters
`{:^10}`	Pad the string on **both sides** up to 10 characters
`{:.10}`	Truncate the string to 10 characters

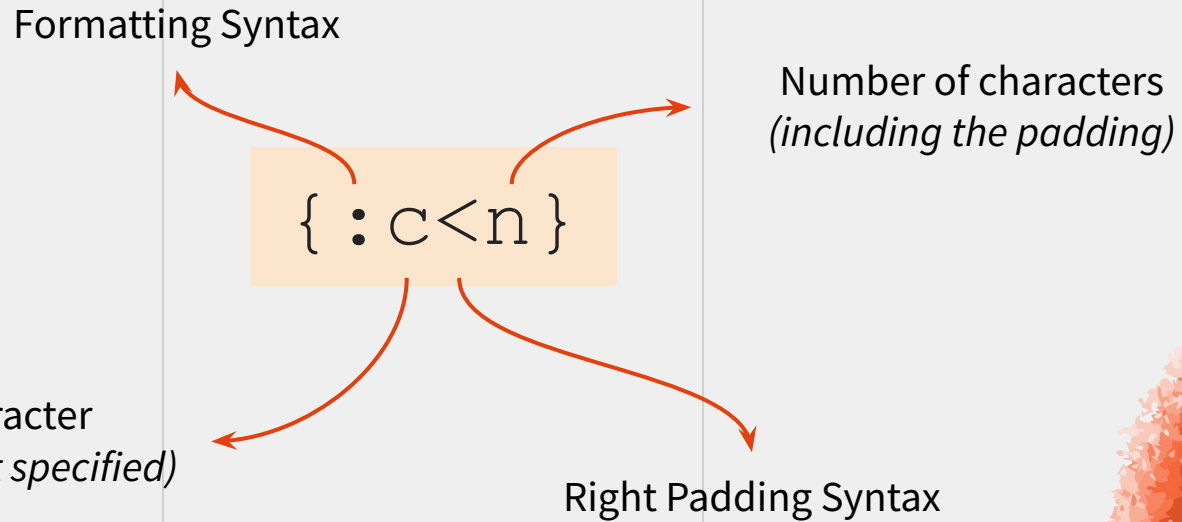
String Formatting (4)

```
'Hello, {:<10}!'.format('John')

'Hello, {:>10}!'.format('John')

'Hello, {:^10}!'.format('John')

'Hello, {:.2}!'.format('John')
```

```
"Hello, John      !"

"Hello,       John!"

"Hello,    John   !"

"Hello, Jo!"
```

String Formatting (5)

Most string formatting can be customized.



String Formatting (6)

Some “advanced” customization.

```
'Hello, {:_>10}!'.format('John')

'Hello, {:.^10.2}!'.format('John')
```

```
"Hello, ______John!"

"Hello, ....Jo....!"
```

String Formatting (7)

Numbers have special formatting options.

`{:_5}`	Pad the number to the **left** side with 5 underscores
`{:+}`	Add the **+ sign** before the number
`{:,}`	Thousand's separator
`{:=+.5}`	Pad with 5 dots **between** the number and its positive sign

`{:e}`	Exponentiation format *(3.14e-5)*
`{:%}`	Percentage format *(84.4%)*
`{:f}`	Float format *(3.14159)*
`{:.5f}`	Truncate the **float** to 5 digits

String Formatting (8)

```
'pi number is {:.5f}'.format(3.141592653589793)

'Padded number : {:04}'.format(42)

'Signed number : {:+}'.format(42)

'Signed Padded number : {:=+4}'.format(42)
```

```
'pi number is 3.14159'

'Padded number : 0042'

'Signed number : +42'

'Signed Padded number : + 42'
```

Exercises

Do the following exercise sheets :

- Variables & Types
- String Formatting

Conditional Structures

Definition

Condition

As its name suggests, a conditional structures is composed of a condition/test and a structure.

The **condition** is always composed of at least **two values** and a **comparator**.

```
value_A comparator value_B
```

The output of the condition is a boolean : **True** or **False**.

Definition (2)

Structure

The **structure** depends on the task.
There are three types :

- The **if** that do only something if the condition is met (True).
- The **if / or if** that do only something if one of the successive conditions is met (True). It stops as soon as one condition is met.
- The **if / or if / if not** that always do something even if the conditions aren't met (False).

Comparators

A lot of comparators exist.

==	True if the values are **equal**
!=	True if the values are **different**
>	True if the first value is **bigger** than the following one
<	True if the first value is **smaller** than the following one
>=	True if the first value is **bigger or equal** to the following one
<=	True if the first value is **smaller or equal** to the following one

Structure

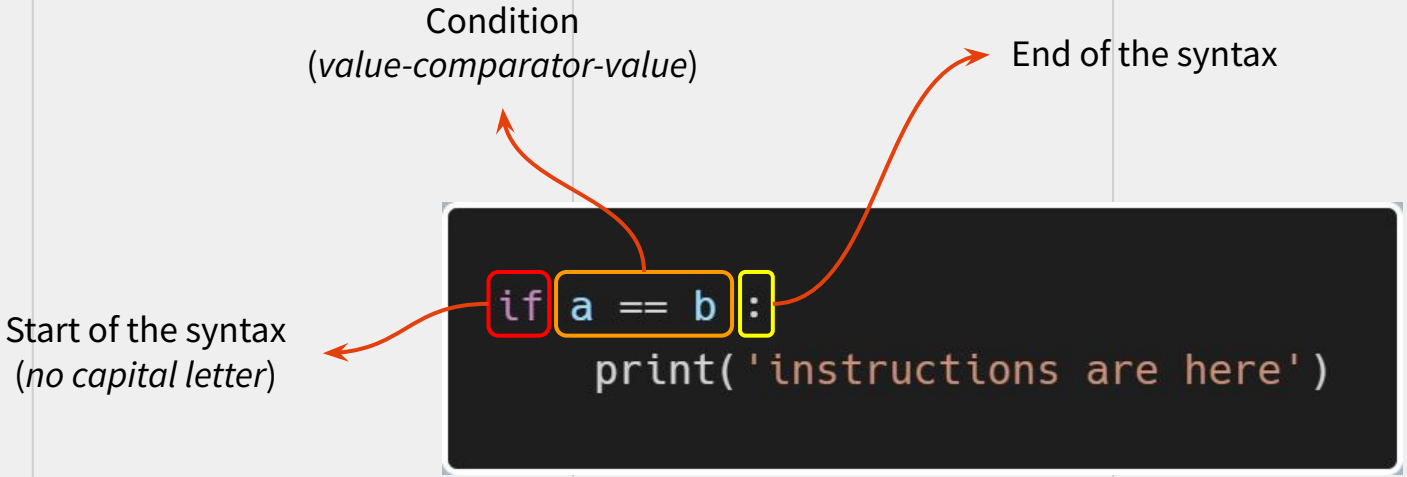
if

The **if** that only do something if the condition is met (True).
Otherwise, the instructions inside the conditional structure are not read.

```
if a == b :
    print('instructions are here')
```

Structure (2)

if



Indentation

In Python, a block of code is **always** delimited by **indentation** (spaces). The syntax is the following :

- It **starts** with a **colon :**
- Then, it is composed of code lines with **indentations** (spaces)
- It **ends** whenever a code line is **not indented**

Inside the if block

Outside of the if block

```
if a == b :
    print('this is an indented code line')
    print('and also that one')
print('this code line is not inside the if')
```

Structure (3)

if / or if

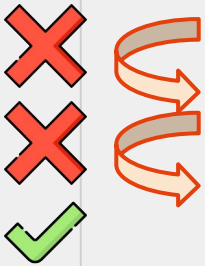
Similarly to the **if - then**, it only do something if **one of the conditions** is met (True). Otherwise, the instructions inside the conditional structure are not read.

If one of the condition is met, the following ones are **skipped**.

```
if a == b :
    print('instructions are here')
elif a > b :
    print('instructions are here')
```

Structure (4)

if / or if



```python
a = 2
b = 1

if a == b :
    print('a is equal to b')
elif a < b :
    print('a is smaller than b')
elif a > b :
    print('a is bigger than b')
```

```python
a = 1
b = 1

if a == b :
    print('a is equal to b')
elif a < b :
    print('a is smaller than b')
elif a > b :
    print('a is bigger than b')
```

The elif structures are skipped

Structures (5)

if / or if / if not

This structure **always do something**. If the `if` and the `elif` are not met, then the code lines inside the `else` are read.

```
if a == b :
    print('a is equal to b')
elif a < b :
    print('a is smaller than b')
else :
    print('a is bigger than b')
```

Multiple conditions

The if structure accept multiple conditions. The conditions are connected by a **logical operator**. The output of a multiple condition can only be True or False and depends on all the conditions.

```
value_A comparator value_B

AND

value_C comparator value D
```

Logical Operators

Four logical operators exist.

`and`	**All the conditions** are met to be True
`or`	**At least one condition** is met to be True
`xor`	**Only one condition** is met
`not`	**Reverse** the result of a condition

Logical Operators (2)

In the following examples, C1 and C2 are conditions such as `a == b` or `a >= b`.

`C1 and C2`	**C2 True**	**C2 False**
C1 True	True	False
C1 False	False	False

`C1 or C2`	**C2 True**	**C2 False**
C1 True	True	True
C1 False	True	False

Logical Operators (3)

In the following examples, C1 and C2 are conditions such as `a == b` or `a >= b`.

`C1 ^ C2`	**C2 True**	**C2 False**
C1 True	False	True
C1 False	True	False

`not C1`	
C1 True	False
C1 False	True

Multiple conditions (2)

or condition

xor condition

```
if a == b or a < 5 :
    print('a == b or a < 5')
elif a >= b and not(b < 5) :
    print('a >= b and b < 5')
elif (a != b) ^ (b == 5) :
    print('a >= b and b < 5')
else :
    print('else')
```

and & **not** condition

else

Nested Structures

A condition can hide another one

Conditional structures can be used inside other conditional structures.
Beware of indentation !

```
if a == b :
    if a > 5 :
        print('a == b and a > 5')
    else :
        print('a == b and a <= 5')
else :
    print('a != b')
```

Side Notes

Multiple conditions or nested structures ?

Generally, using nested structures is sometimes better than multiple conditions as they tend to clean unnecessary duplicates :

- Use **multiple conditions** whenever it does not create any condition duplicate or whenever it reduces the total number of code lines.
- Use **nested structures** whenever it removes condition duplicates

Conditional structures should not be overused. Their usage should be limited to the bare minimum as most task can be solved using other Python tools.

Side Notes (2)

Compare strings

ASCII table

Decimal	Hex	Char	Decimal	Hex	Char	Decimal	Hex	Char
32	20	[SPACE]	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(	72	48	H	104	68	h
41	29	)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[	123	7B	{
60	3C	<	92	5C	\	124	7C	|
61	3D	=	93	5D	]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	[DEL]

```
'a' > 'b'
```

↓

```
unicode(a) > unicode(b)
```

↓

```
U+0061 > U+0062
```

↓

```
False
```

Side Notes (3)

Compare strings

Python has two functions for the unicode format :

- One that converts a character into its unicode number : `ord()`
- One that converts a number into a its unicode character : `chr()`

It is not possible to convert more than one character with `ord()`

```
hex_number = ord('a')
print(hex_number)
>   97

mystring = chr(hex_number)
print(mystring)
>   'a'
```

Side Notes (4)

if value is not None

In Python, the existence of a variable value can be checked with the if condition.
Beware, in Python the variable can exist without a “proper” value : the `None` value.

```
a = 1

if a :
    print('a has a value')
else :
    print('a is None')
```

```
a = None

if a :
    print('a has a value')
else :
    print('a is None')
```

Side Notes (5)

if value is not None

In fact, Python is using the `is` **identity operator** to check if the value exist.
This operator can also be used to check the type of the variable.

```python
a = None

if a :
    print('a has a value')
else :
    print('a is None')
```

```python
a = None

if a is not(None) :
    print('a has a value')
else :
    print('a is None')
```

Exercises

Do the following exercise sheets :

- Conditional Structures p.I

Once the aforementioned exercises have been completed :

- Conditional Structures p.II
  - The Date Checker
  - The Car Insurance

Data Structure

Table of Content

The following section is divided into three parts :

Definition

A data structure (also known as a container) is a **data organization** and **storage format**. More precisely, it is a collection of values.

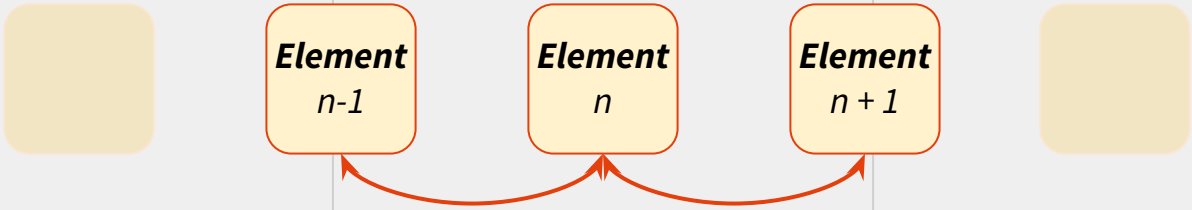
Data structures can be of two types :

- **Linear** - elements are arranged sequentially or linearly
  - ***Dynamic*** - has a variable memory size
  - ***Static*** - has a fixed memory size
- **Non-linear** - elements are not placed sequentially or linearly

Definition (2)

Linear Data Structures

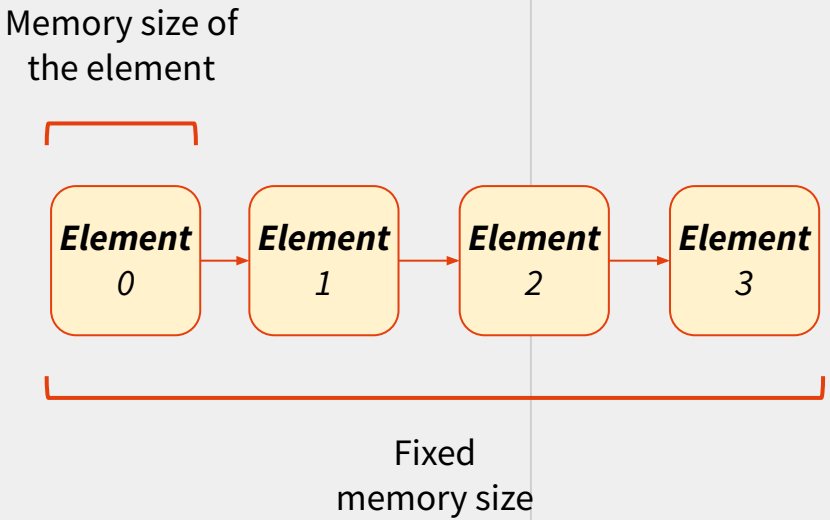
Linear **data structure** elements are linked to each other. These structures can be assimilated to a sequence and all the elements can be traversed in a single run.

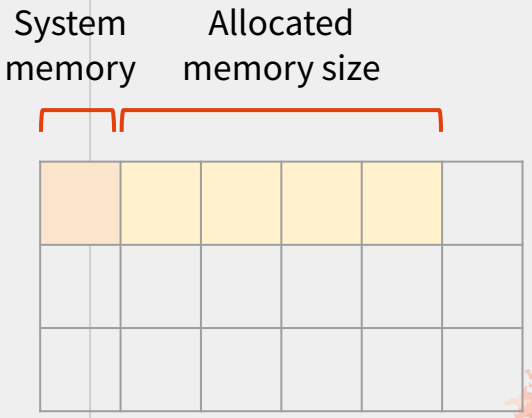


Definition (3)

Static Linear Data Structures

Static linear data structures have a fixed memory size upon their creation (it is not possible to add new elements). The allocated memory must be contiguous.

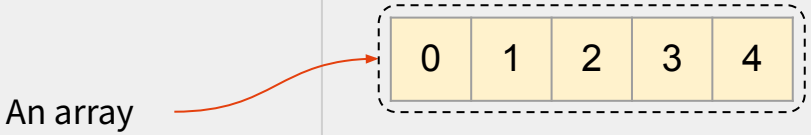




Definition (4)

Static Linear Data Structures

The most known **static linear** data structures are the **arrays**. They are very simple as they cannot be modified after their creation and they can accept only one variable type. It is a box that contain ordered variables.



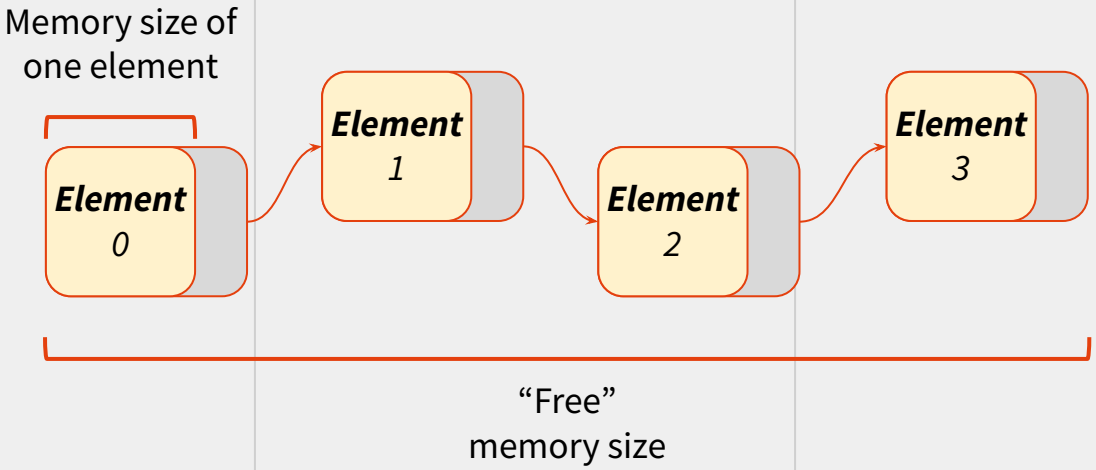
Most programming languages have arrays … but not Python (*not in-built*) !

Instead Python has a structure called a **Tuple** which is a bit different than the array because it accepts multiple element types.

Definition (5)

Dynamic Linear Data Structures

Dynamic linear data structures allow to modify the memory size (*ex : adding new elements*) after their creation. The allocated memory is not forced to be contiguous.



Definition (6)

Dynamic Data Structures

The most known **dynamic linear** data structures is the **Linked List**. It is a sequence of nodes that contain data and a link to the next node (see figure of the previous slide).

It can be derived into multiple data structures types such as :

- the **queues** ;
- the **stacks** ;
- **non-linear** data structures.

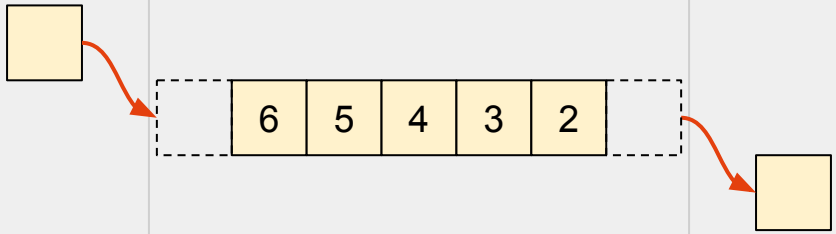
Python default data structure is the **list** which is not a Linked List but more like a dynamic array.

Definition (7)

Queues

Queues in programming are very similar to queues at the grocery store.

They are a collection of ordered elements that can be modified by an addition at one end of the sequence and a deletion at the other end of the sequence (**First In, First Out**).

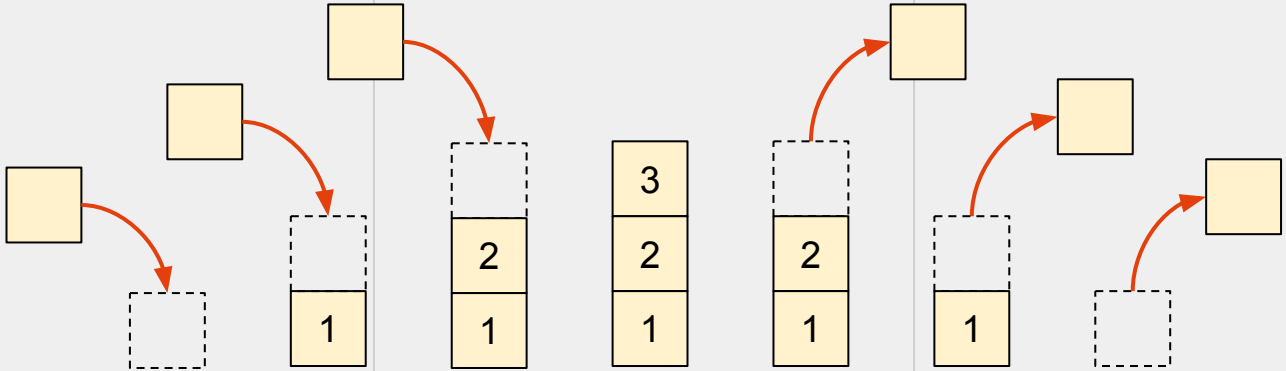


Definition (8)

Stacks

Stacks in programming are very similar to a pile of plates in a cafeteria : the cleaned plates are added to the top of the pile, pushing down the previous ones.

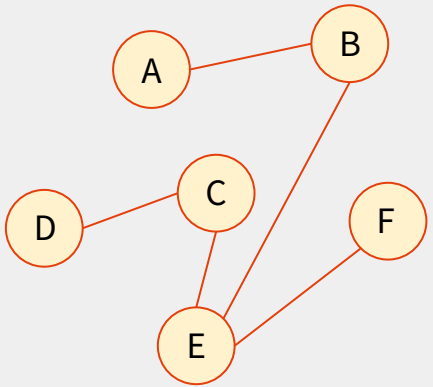
They are a collection of ordered elements that can be modified by an addition at one end of the sequence and a deletion at the **same** end of the sequence (**Last In, First Out**).

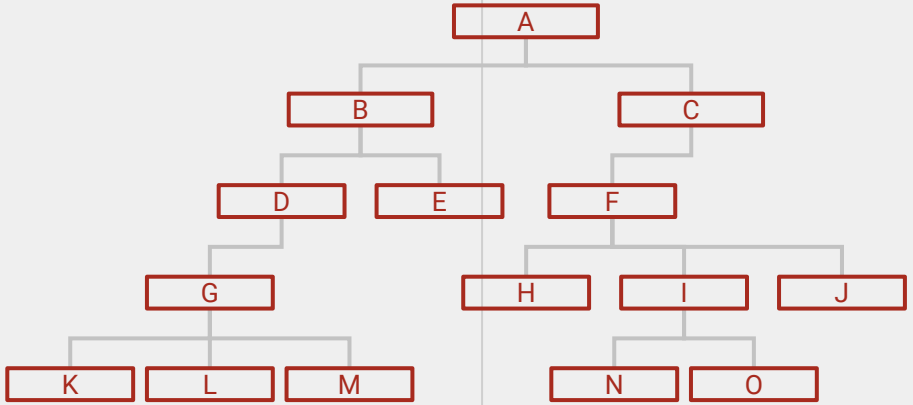


Definition (9)

Non-Linear Data Structures

Non-linear data structure elements are not ordered into a sequence.

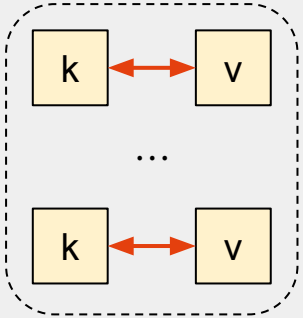




Definition (10)

Associative Arrays

Associative Arrays are also known as **maps** or **dictionaries**. They store a collection of (key, value) pairs. Each possible key appears at most once in the collection. Values can be other data types.



Python has an in-built **dictionary** data structure type.

Python Structures

In Python, data structures are classified in two main categories :

- **Ordered** or not ;
- **Mutable** or not ;

And two minor categories :

- Allow **duplicates** or not ;
- Allow multiple **element types** ;

Python Structures (2)

In Python, a lot of data structures exists (non-exhaustive list) :

- List
- Tuple
- Dictionary
- Set
- String
- *Array*
- *Stack*
- *Queue*
- *Graph*

In-built Python data structures

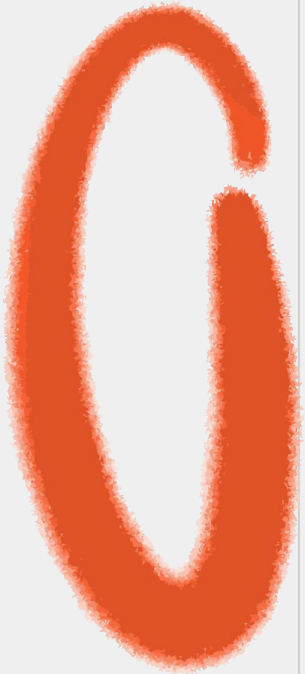
External Python data structures

Python Structures (3)

In-built structures

Python	Data Structure	Mutable	Ordered	Duplicates	Element types
List	Dynamic	Yes	Yes	Allowed	Multiple
Tuple	Static	No	Yes	Allowed	Multiple
Set	Non-linear	Yes/No*	No	Not allowed	Multiple
Dictionary	Non-linear	Yes	Yes**	Not allowed	Multiple
String	Static	No	Yes	Allowed	One***

* *Set elements are not mutable but the set itself is mutable*
** *Dictionaries are ordered since Python version 3.7*
*** *Strings are made of characters*

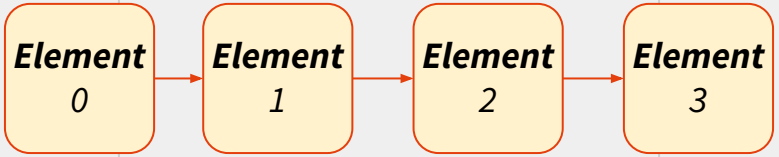


Part I

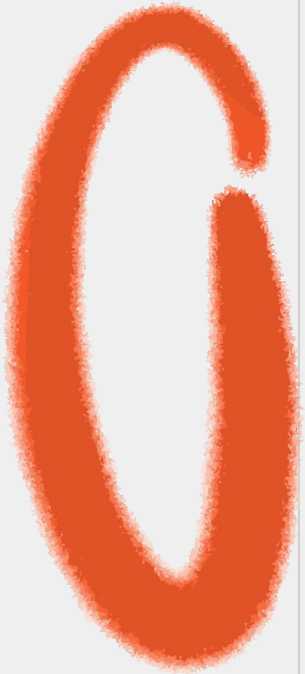
Sequences

Sequence

Sequences are a chain of ordered elements.



Strings, **Lists** and **tuples** are sequences of ordered elements.



Lists

Lists

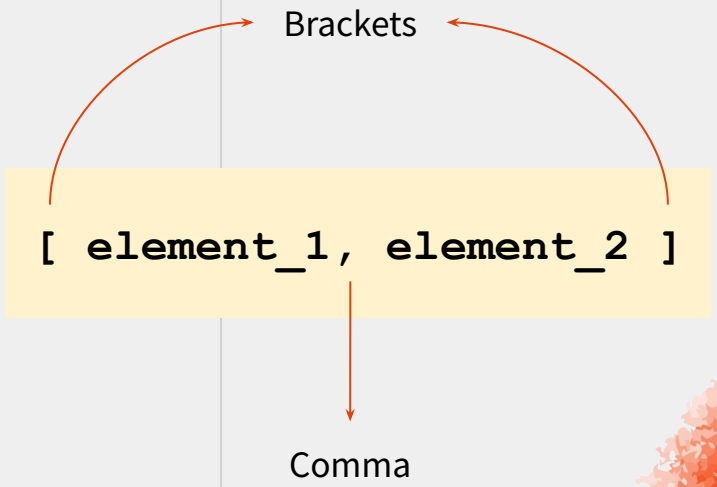
Lists are the simplest data structure of Python :

- Lists accept all types of elements (integers, floats, strings, booleans …)
- Lists accept any **element type combination**
- List elements are **ordered** and allow **duplicate values**
- List elements are **changeable (mutable)**
- List elements can be accessed by their **index**

Lists (2)

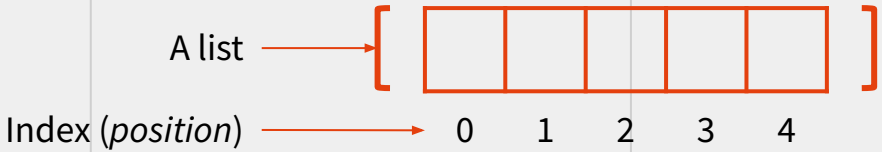
Basics

```
# one element
mylist = [0]

# multiple elements
mylist = [1.1, 1.4, 2.2, 3.2]

# multiple element types
mylist = ['abc', 1, 3.14, True]
```



Lists (3)

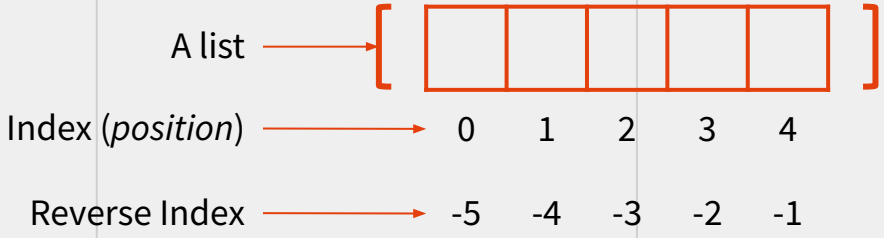
Lists index




```
mylist = ['a','b','c','d']

print(mylist[0])
>    'a'

print(mylist[3])
>    'd'
```

Lists (4)

Reverse indexing




```
mylist = ['a', 'b', 'c', 'd']

print(mylist[-1])
>   'd'

print(mylist[-4])
>   'a'
```

Lists (5)

Slicing

`mylist[b:e]`	Get all the list elements **from the b-th one to the e-th one excluded**
`mylist[-b:-e]`	Same as above in **reverse indexing**
`mylist[b:e:s]`	Same as the first one but with a **step of s**
`mylist[b:e:-s]`	Same as above but with a **reverse step of s**
`mylist[b:]`	Get all the list elements **from the m-th one to the end**
`mylist[:e]`	Get all the list elements **from the start to the n-th one excluded**
`mylist[::-1]`	**Reverse all the list elements**

Lists (6)

Slicing Examples

```
mylist = ['a','b','c','d','e']

print(mylist[0:2])
>   ['a', 'b']

print(mylist[2:])
>   ['c', 'd', 'e']

print(mylist[:3])
>   ['a', 'b', 'c']

print(mylist[0:5:2])
>   ['a', 'c', 'e']

print(mylist[::-1])
>   ['e','d','c','b','a']
```

Lists (7)

List operations

Adding an element

```
mylist = [0, 1, 2, 3, 4]

mylist = mylist + [5]
print(mylist)
>   [0, 1, 2, 3, 4, 5]
```

`list + [new_element]`

Changing an element

```
mylist = [0, 1, 2, 3, 4]

mylist[0] = 'a'
print(mylist)
>   ['a', 1, 2, 3, 4]
```

`list[n] = new_element`

Removing an element

```
mylist = [0, 1, 2, 3, 4]

del mylist[2]
print(mylist)
>   [0, 1, 3, 4]
```

`del list[n]`

Lists (8)

Other addition methods

New elements can be **added** to a list in two other ways.

.append()

At the end of a list

```
mylist = [0, 1, 2, 3, 4]

mylist.append('a')
print(mylist)
>   [0, 1, 2, 3, 4, 'a']
```

`mylist.append(new_el)`

.insert()

At a specific index

```
mylist = [0, 1, 2, 3, 4]

mylist.insert(2, 'a')
print(mylist)
>   [0, 1, 'a', 2, 3, 4]
```

`mylist.insert(n, new_el)`

Lists (9)

Other removal methods

Elements can be **removed** in three other ways.

.remove()

First matching element

```
mylist = [0, 1, 2, 3, 4]

mylist.remove(0)
print(mylist)
>   [1, 2, 3, 4]
```

`mylist.remove(element)`

.pop()

Extract the element at index

```
mylist = ['a','b','c','d']

el = mylist.pop(2)
print(mylist)
>   ['a', 'b', 'd']
```

`mylist.pop(index)`

.clear()

All the elements

```
mylist = [0, 1, 2, 3, 4]

mylist.clear()
print(mylist)
>   []
```

`mylist.clear()`

Lists (10)

Addition of two lists

Math operator

```
mylist_1 = [0, 1, 2, 3, 4]
mylist_2 = [5, 6, 7, 8, 9]

mylist_3 = mylist_1 + mylist_2
print(mylist_3)
>   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`mylist + mylist`

.extend()

Multiple elements at the end of a list

```
mylist_1 = [0, 1, 2, 3, 4]
mylist_2 = [5, 6, 7, 8, 9]

mylist_1.extend(mylist_2)
print(mylist_1)
>   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

`mylist.extend(iterable)`

Lists (11)

List duplication

Lists can be duplicated by using the multiplication operator. It is a good way to create a list with one number repeated n times.

```
mylist = [0] * 5

print(mylist)
>   [0, 0, 0, 0, 0]
```

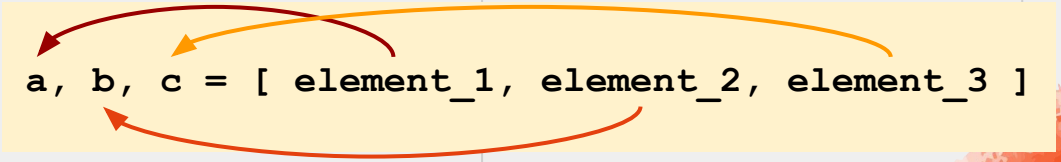
```
mylist = [1, 2, 3] * 2

print(mylist)
>   [1, 2, 3, 1, 2, 3]
```

Lists (12)

List unpacking

Lists can be unpacked *i.e.* list elements can be stored in independant variables.

```
mylist= [1, 2, 3]

one, two, three = mylist
print(one)
>   1

print(two)
>   2

print(three)
>   3
```



Lists (13)

Other special methods

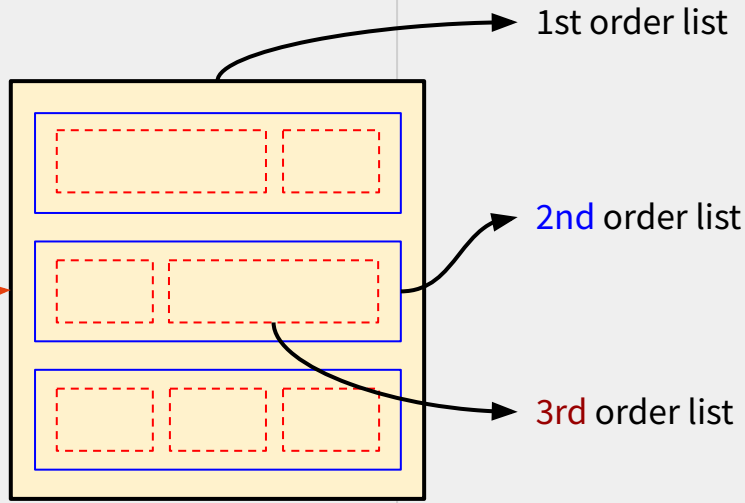
Python list objects have other **methods**.

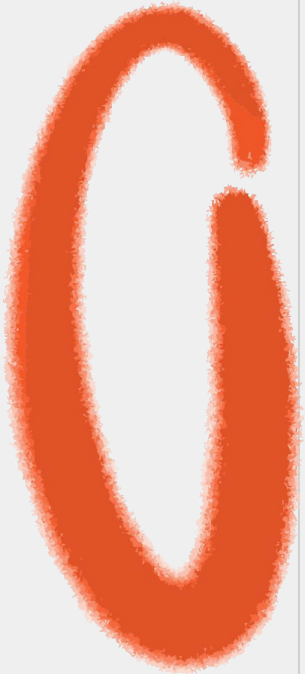
`mylist.copy()`	Make a **copy** of the list
`mylist.sort(reverse=False)`	**Sort** the list (*default is ascending*)
`mylist.count(element)`	**Count** the number of matching elements
`mylist.index(element)`	Return the **index** of the first matching element

List-ception

Lists of any length can be stored inside of a list. Thus, a list of lists of lists of lists could be created.
Every methods and functions can be used on list of lists.

```
# a list of lists
mylist = [[0,1,2],
          [3,4,5],
          [6,7,8]]

# a list of lists of lists
mylist = [[[0,1],[2]],
          [[3],[4,5]],
          [[6],[7],[8]]]
```





Tuples

Tuples

Python **tuples** are static data structures :

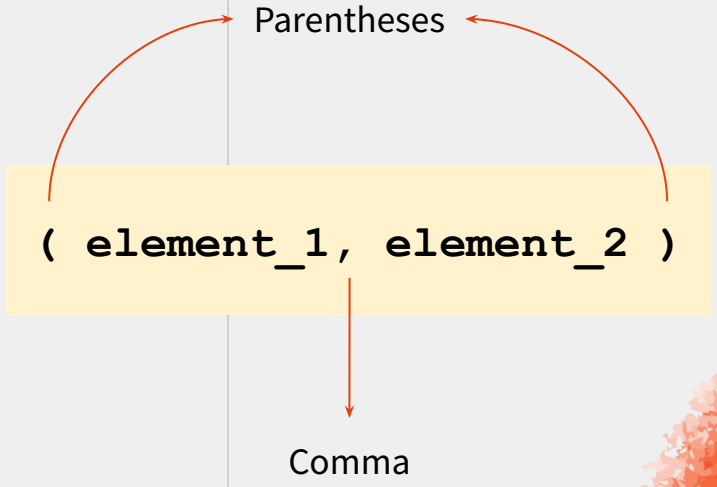
- Tuples accept **all types of elements** and **any element type combination**
- Tuples elements are **ordered** and allow **duplicate values**
- Tuples elements are **unchangeable (immutable)**
- Tuples elements can be accessed by their **index**

Tuples are **faster** than lists.

Tuples (2)

Basics

```python
# one element
mytuple = (0,)

# multiple elements
mytuple = (1.1, 2.2, 3.3, 4.4)

# element combinaison
mytuple = (0, 'a', 3.14, True)
```



Tuples (3)

Tuple index

Tuples indexing is exactly the same as for the lists :

- It starts at `0` and ends at `n-1`
- It supports **negative indexing**
- It supports **slicing**

```
mytuple = (0, 'a', 3.14, True)

print(mytuple[0])
>   0

print(mytuple[-1])
>   True

print(mytuple[1:3])
>   ('a', 3.14)
```

Tuples (4)

Tuple operations

An **element can be added** to a tuple. However, this will create a new tuple instead of extending the original tuple. This operation is not memory & time efficient compare to lists.

Tuples **support unpacking** and **duplication**.

Elements of a tuple **cannot be changed** or **removed**. A tuple can still be deleted with the `del mytuple` command.

Consequently, a tuple should only be preferred over a list when it is certain that it will never need to be modified.

Tuples (5)

Addition, unpacking & duplication

Math operator

```
mytuple_1 = (1, 2, 3)
mytuple_2 = (4, 5, 6)

mytuple_3 = mytuple_1 + mytuple_2
print(mytuple_3)
>   (1, 2, 3, 4, 5, 6)
```

Unpacking

```
mytuple = (1, 2, 3)

one, two, three = mytuple
print(one)
>   1
print(two)
>   2
print(three)
>   3
```

Duplication

```
mytuple = (1, 2, 3) * 2

print(mytuple)
>   (1, 2, 3, 1, 2, 3)
```

Tuples (6)

Other special methods & functions

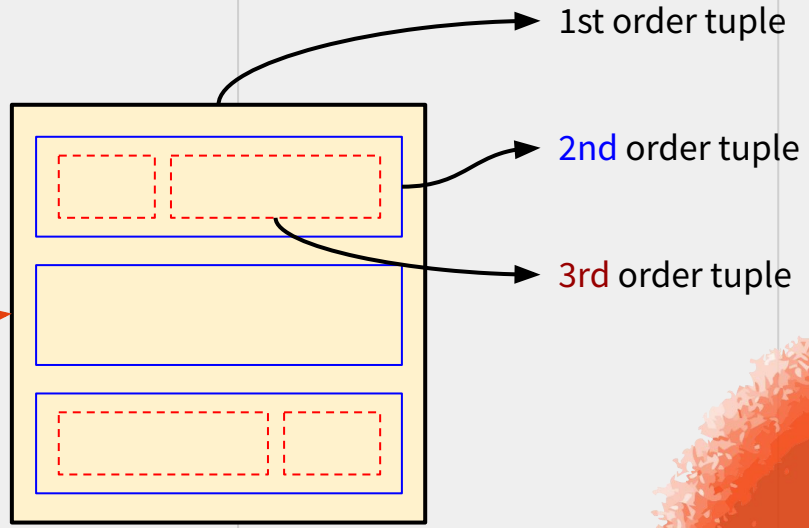
Python tuple objects have other **methods**.
It is not possible to copy or to sort a tuple with methods.

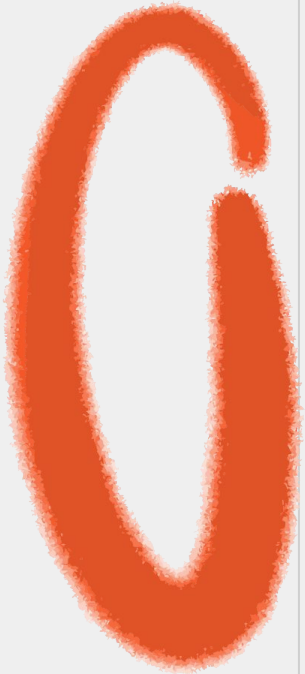
`mytuple.count(element)`	**Count** the number of matching elements
`mytuple.index(element)`	Return the **index** of the first matching element

Nested Tuples

Tuples can also be **nested**.

```
# nested tuple
mytuple = ((1,2,3),
           (4,5,6),
           (7,8,9))

# multiple nesting
mytuple = (((1,), (2,3)),
           (4,5,6),
           ((7,8), (9,)))
```





Strings

Strings

Python **strings** can be seen as containers :

- Strings are composed of **characters**
- Strings elements are **ordered** and can be **duplicated**
- Strings elements are **not changeable (immutable)**
- Strings elements can be accessed by their **index**

Strings (2)

Basics

```
mystring = 'abcdefgh'

mystring = "abcdefgh"

mystring = """abcdefgh"""

mystring = '''abcdefgh'''
```

Triple quotes	‘‘‘ **characters** ’’’
Double quotes	“ **characters** ”
Single quotes	‘ **characters** ’

Strings (3)

Encodings

String characters are not universal and must be encoded.

By default, most strings are encoded into the **UTF** format (an alias for *Unicode Transformation Format*).

Even if more than 98% of the web pages are encoded into the **UTF-8** format, some encodings might be useful. Python supports :

- **ASCII** (*American Standard Code for Information Interchange*)
- **CP** (*Code Page*)
- **ISO** / **latin** (an extension of the Unicode format)

Strings (4)

Bytes

Sometimes, strings are encoded as **bytes**. A byte is composed of 8 bits which are the smallest unit of information in computers.

These strings will appear with the letter b before the quotes. Python has tools to decode byte-strings into any of the available encodings.

```
mybytes = b'abcdefgh'

mybytes.decode('utf-8')
```

Strings (5)

Most used methods

`mystring.lower()`	Turn the string **lowercase**
`mystring.upper()`	Turn the string **uppercase**
`mystring.count(character)`	**Count** the occurrences of a character
`mystring.find(string)`	True if the string **is in** the other string, else False
`mystring.endswith(suffix)`	True if the string **ends with** the suffix, else False
`mystring.startswith(prefix)`	True if the string **starts with** the suffix, else False
`mystring.split()`	Cut the string into pieces given a character (*default is space*)
`mystring.join(iterable)`	**Concatenate all the strings** inside the iterable

Strings (6)

Most used methods p.II

`mystring.encode(encoding)`	**Encode** the string into an encoding
`mystring.strip(characters)`	**Remove** the strings parts that match the characters (start/end)
`mystring.replace(old, new)`	**Replace** the “old” string by the “new” one

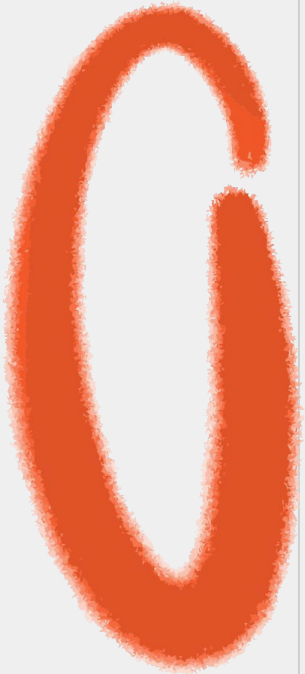
String methods also include the `.islower()`, `.isdigit()` and so on, that check what is inside the string.

More methods exists : *Documentation*

Nested strings

Strings can also be **nested** using a combination of the double and the single quotes or by using the triple quote.

```
# nested double quote string
mystring = "Is the color 'green' ?"


# nested triple quotes
mystring = """It is a nested '"triple"' and "'double'" quote"""
```



Side Notes

Side Notes

Advanced Unpacking

In Python, an **asterisk** can be used to **unpack an ordered container**, such as a list or a tuple, to **return lists**.

```
mylist = [0, 1, 2, 3, 4]

a, *b = mylist
print(a)
>   0

print(b)
>   [1, 2, 3, 4]
```

```
mytuple = ('a', 'b', 'c', 'd')

a, *b = mytuple
print(a)
>   'a'

print(b)
>   ['b', 'c', 'd']
```

Side Notes (2)

In-built functions

Some Python **in-built functions** can be used on sequences (list, tuple & strings).

`min(iterable)`	Find the **min** of the sequence
`max(iterable)`	Find the **max** of the sequence
`sum(iterable)`	**Sum** the elements of the sequence
`len(iterable)`	Find the **number of elements** inside a sequence
`sorted(iterable)`	**Sort** any data structure (**returns a list**)
`list(iterable)`	**Convert** any data structure **into a list**
`tuple(iterable)`	**Convert** any data structure **into a tuple**
`str()`	**Convert** any variable **into a tuple**

Side Notes (3)

In-built functions

```
mytuple = (4, 2, 3, 1)

# min/max
print(min(mytuple))
>   1

# len
print(len(mytuple))
>   4

# sorted
print(sorted(mytuple))
>   [1, 2, 3, 4]
```

```
mylist = [1, 2, 3]

mytuple = tuple(mylist)
print(mytuple)
>   (1, 2, 3)

mystring = str(mytuple)
print(mystring)
>   '(1, 2, 3)'

mylist = list(mystring)
print(mylist)
>   ['(', '1', ',', ' ', '2', ',', ' ', '3', ')']
```

Side Notes (4)

Conditional Structures

Sequences can be used as conditions in conditional structures using the **is** or the **in** operators :

- The **in** is used to check if a variable is inside an iterable.
- The **is** is used to check the type of the iterable.

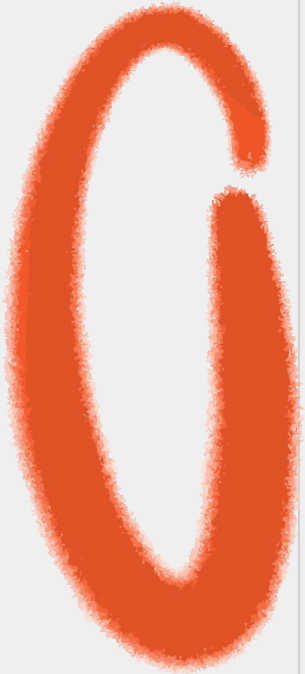
```
mylist = [0,1,2]

if type(mylist) is list :
    print('It is a list')
```

```
mylist = [0,1,2]

if 2 in mylist :
    print('3 is in the list')
```

Exercises

Do the following exercise sheet :

- Data Structures p.I

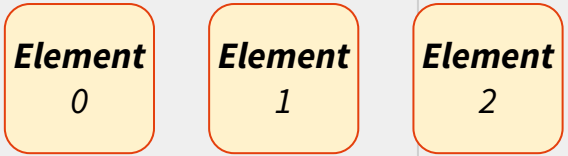


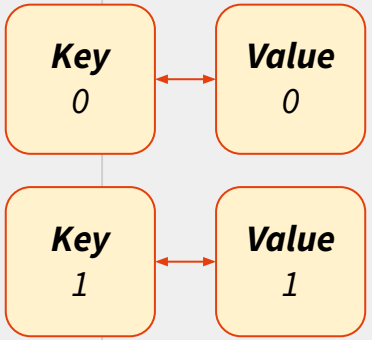
Part II

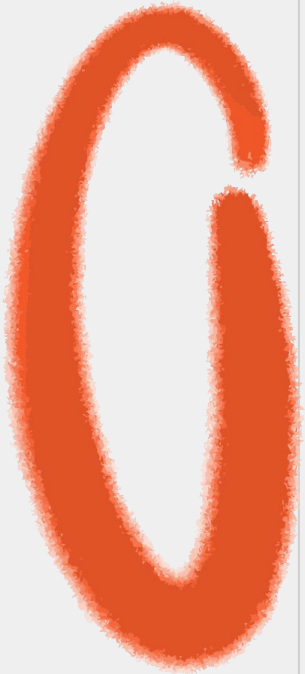
Dictionary & Set

Dictionary and Set

Dictionaries and **sets** are a collection of unordered elements. They are very similar excepted that dictionaries are a collection of element pairs (a key and its value) whereas sets are a collection of single elements.







Dictionary

Dictionary

Python **dictionaries** are non-linear data structures :

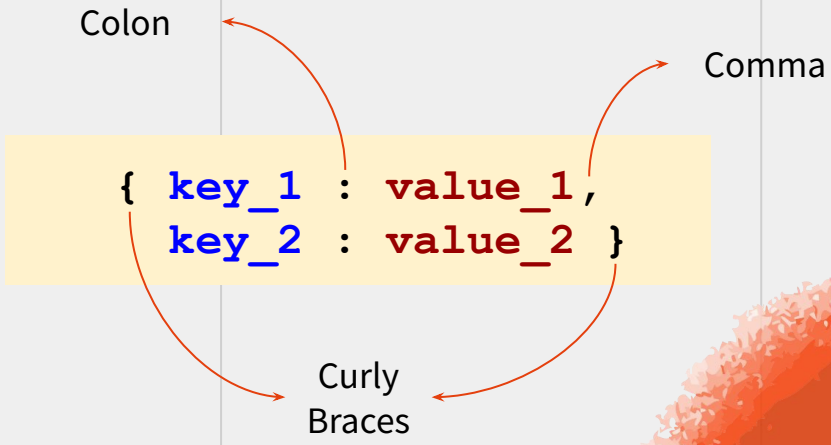
- Dictionaries are composed of **pairs of elements**
- Dictionaries elements are **ordered*** and cannot be **duplicated**
- Dictionaries elements are **changeable (mutable)** & **dynamic**
- Dictionary **values** can be accessed by their **keys**

** Dictionaries are ordered since the Python 3.7 version which means the order of insertion in the dictionary matters. However, most dictionaries in the programming world remain unordered. Thus I would advise to still consider dictionaries as unordered.*

Dictionary (2)

Basics

```
# One element dictionary
mydict = {'one':1}

# Multiple elements
mydict = {'one':1, 'two':2, 'three':3}

# Multiple element types
mydict = {'one':'one', 'two':True, 'three':3.14}
```



Dictionary (3)

Basics

Dictionaries can also be easily created using the `dict()` function.

```
# with a list of tuples
mydict = dict([('One', 1),
               ('Two', 2),
               ('Three', 3)])

# with the equal symbol
mydict = dict(One = 1,
              Two = 2,
              Three = 3)
```

Dictionary (4)

Accessing a value

Dictionary elements cannot be accessed by their index. They can only be accessed thanks to their **keys**. As a consequence, **a dictionary can't be sliced**.

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

print(mydict['one'])
>   1

print(mydict.get('two'))
>   2
```

Dictionary (5)

Index vs Keys

Never get indexes and keys mixed up when dealing with dictionaries. A dictionary can have integers as keys which may spawn confusion for the users.

```
mydict = {1:'a', 0:'b', 4:'c', 3:'d'}
mylist = ['a', 'b', 'c', 'd']

print(mydict[0])
>    'b'

print(mylist[0])
>    'a'
```

Dictionary (6)

Get Keys & Values

The **keys** and the **values** of a dictionary can be returned with `.keys()` and `.values()` respectively. It will return a **dictionary view object** that can be transformed into a list with the `list()` function.

```
mydict = {'One':1, 'Two':2, 'Three':3}

keys = mydict.keys()
print(list(keys))
>   ['One', 'Two', 'Three']

values = mydict.values()
print(list(values))
>   [1,2,3]
```

Dictionary (7)

Get Keys & Values

The **keys** and the **values** of a dictionary can be returned simultaneously with `.items()`. The result is also a dictionary view object.

```
mydict = {'One':1, 'Two':2, 'Three':3}

items = mydict.items()
print(list(items))
>   [('One',1), ('Two',2), ('Three',3)]
```

Dictionary (8)

Change values

The **values** of a dictionary can be changed by calling the dictionary with the associated key. It is also possible to update multiple values with the `.update()` method.

```
mydict = {'a':0, 'b':1, 'c':2}

mydict['a'] = 10
print(mydict)
>   {'a':10, 'b':1, 'c':2}

mydict.update({'b':20, 'c':30})
print(mydict)
>   {'a':10, 'b':20, 'c':30}
```

It can also be a list of tuples (key, value)

Dictionary (9)

Add key-value pairs

New key-value pairs can be added to dictionary at any time with the exact same syntax as when a value is changed.

```
mydict = {'a':0, 'b':1, 'c':2}

mydict['d'] = 3
print(mydict)
>   {'a':0, 'b':1, 'c':2, 'd':3}

mydict.update({'e':4, 'f':5})
print(mydict)
>   {'a':0, 'b':1, 'c':2, 'd':3, 'e':4, 'f':5}
```

Dictionary (10)

Remove key-value pairs

Two methods exist to **remove key-value pairs** :

- `pop(key)` – it extracts a **specific** key-value pair
- `popitem()` – it extracts the **last** key-value pair added.

Two methods exist to **empty/delete a dictionary** :

- `clear()` – it empties the entire dictionary.
- `del` – it deletes the entire dictionary or a remove a key-value pair

Dictionary (11)

Remove key-value pairs

```
mydict = {'a':0, 'b':1, 'c':2}

a = mydict.pop('a')
print(mydict)
>   {'b':1, 'c':2}

c = mydict.popitem()
print(mydict)
>   {'b':1}

del mydict['b']
print(mydict)
>   {}
```

Merging Dictionary

Dictionaries can be merged with the `dict_1 | dict_2` operation (create a new one), the `dict_1 |= dict_2` (update the first one) or with the `.update()` function.

```
mydict_1 = {'a':0, 'b':1}
mydict_2 = {'c':2, 'd':3}

mydict_3 = mydict_1 | mydict_2
print(mydict_3)
>   {'a':0, 'b':1, 'c':2, 'd':3}

mydict_1 |= mydict_2
print(mydict_1)
>   {'a':0, 'b':1, 'c':2, 'd':3}
```

Conditional Structure

It is possible to check if a key is present in the dictionary using the **in** operator in the condition of a test.

```
mydict = {'a':0, 'b':1, 'c':2}

if 'c' in mydict :
    print('The c key is in the dictionary')
else :
    print('The c key is not in the dictionary')
```

Nested Dictionary

Dictionary can also be **nested** with other dictionary or data structures such as lists or tuples.

```
mydict = {'colors': {'yellow':0, 'blue':1, 'red':2},
          'sizes':  {'big':0, 'medium':1, 'small':2}
          'shape':  {'square':False, 'triangle':True}}

mydict = {'tuple' : (1,2,3,4,5),
          'list': [6,7,8,9,10],
          'dict_of_list':{'a' : [True, False, True],
                          'b' : [False, True, False]}}
```

Key Restriction

Dictionary keys have one restriction : to be an **hashable** object. Such object can be an integer, a float, a string, a boolean and even a tuple !

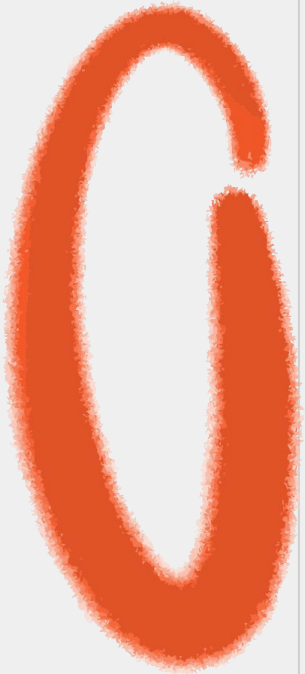
```
mydict = {(1,2) : 'a', (2,3) : 'b'}

keys = list(mydict.keys())
print(keys)
>   [(1,2), (2,3)]
```

Hash Value

An object is **hashable** when it can be passed to a hash function. A **hash function** takes data of arbitrary size and maps it to a “simpler” fixed-size value, called a **hash value**, which is used for table lookup and comparison.

To verify if an object is hashable, you can try to use the `hash()` function on it.

```
mytuple = (1,2,3)

print(hash(mytuple))
>   5293440672954974 51
```



Sets

Set

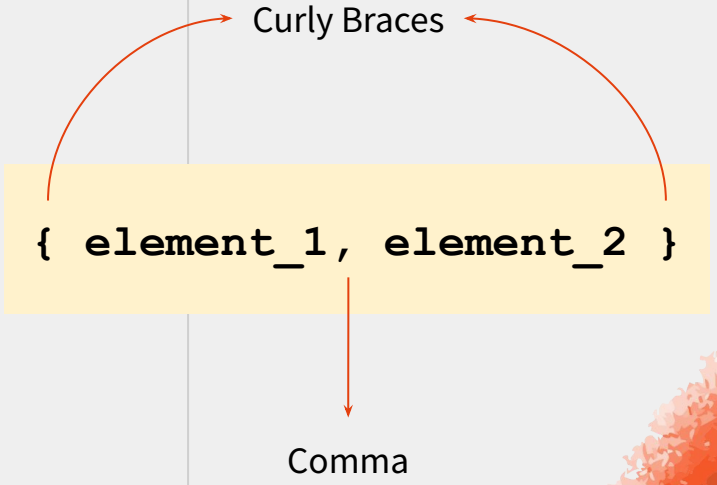
Python **sets** are non-linear data structures :

- Sets are composed of **hashable elements**
- Sets elements are **unordered** and cannot be **duplicated**
- Sets elements **can’t be accessed**
- Sets **elements** are **immutable** but **sets** are **mutable**

Set (2)

Basics

```
# one element
myset = {'a'}

# multiple elements
myset = {0, 1, 2, 3, 4}

# multiple types
myset = {0, 3.14, True, 'a'}
```



Set (3)

Basics

Sets can also be easily created using the `set()` function around any data structure.

```
myset = set(['a', 'b', 'c', 'd'])

myset = set(('a', 'b', 'c', 'd'))

myset = set({'a':0, 'b':1, 'c':2, 'd':3})

myset = set('abcd')

print(myset)
>   {'a', 'b', 'c', 'd'}
```

Set (4)

Adding elements

Even if set elements are not mutable, **elements can still be added to a set** using the `.add()` (for one element) and the `.update()` (for multiple elements) methods.

```
myset = {0,1,2,3,4}

myset.add(5)
print(myset)
>    {0,1,2,3,4,5}

myset.update({6,7,8})
print(myset)
>    {0,1,2,3,4,5,6,7,8}
```

Lists, tuples & dictionaries can also be added to the set

Set (5)

Removing elements

Similarly, **elements can be removed from a set** using three different methods :

- `.remove(element)`
- `.pop(element)` - Extract the element and delete it from the set
- `.discard(element)` - Do not produce an error if the element doesn't exist in the set

Additionally, a set can be **deleted/emptied** using :

- `del set`
- `.clear()`

Set (6)

Removing elements

```
myset = {'a', 'b', 'c'}

myset.pop('a')
print(myset)
>   {'b', 'c'}

myset.delete('b')
print(myset)
>   {'c'}

myset.discard('c')
print(myset)
>   {}
```

Set Operations

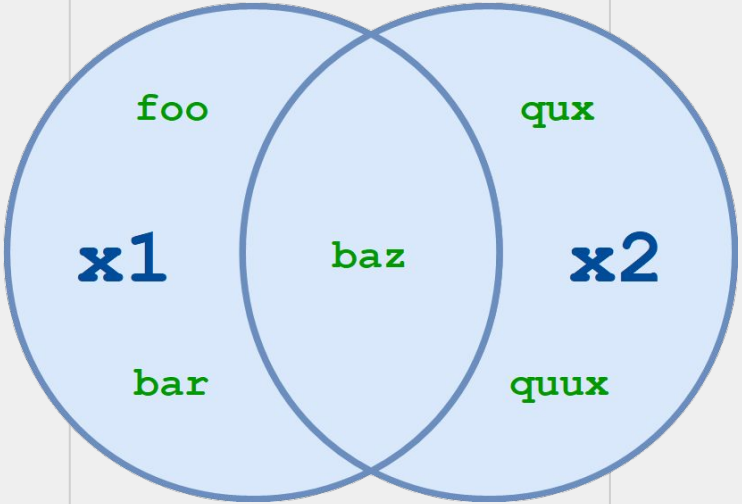
Sets can be used in **four operations** :

- **Union** - Merge the first & the second sets together
- **Intersection** - Keep the common elements of the two sets
- **Difference** - Remove the first set element that are in the second set.
- **XOR** - Keep the first set elements that are not in the second one and keep the second set elements that are not in the first one.

Set Operations (2)

Union

Sets can be **merged** using the `.union(set)` method or the `|` operator.

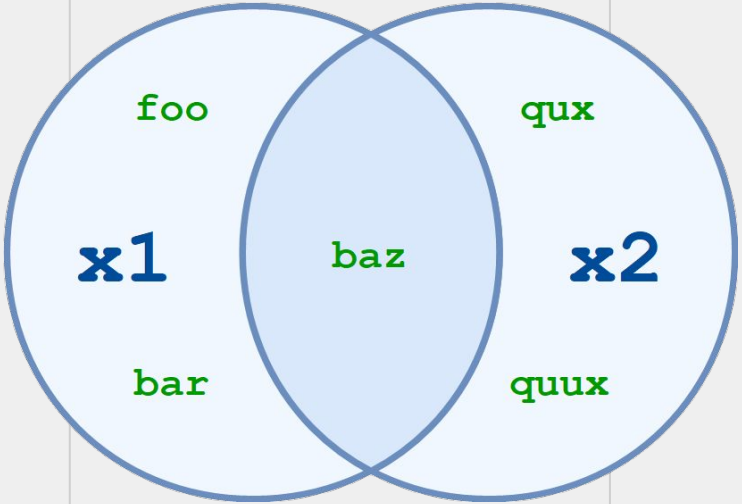



```
x1 = {'foo', 'bar', 'baz'}
x2 = {'qux', 'quy', 'baz'}

x3 = x1.union(x2)
print(x3)
>   {'quy', 'bar', 'baz', 'qux', 'foo'}
```

Set Operations (3)

Intersection

Sets can be **intersected** using the `.intersection(set)` method or the `&` operator.

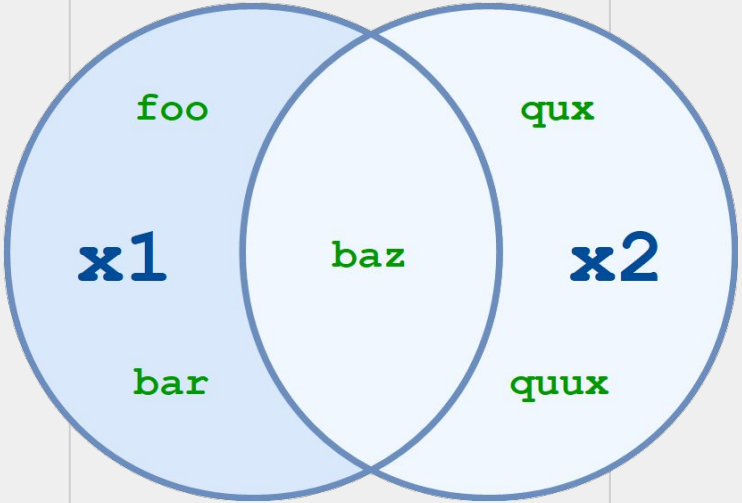



```
x1 = {'foo', 'bar', 'baz'}
x2 = {'qux', 'quy', 'baz'}

x3 = x1.intersection(x2)
print(x3)
>   {'baz'}
```

Set Operations (4)

Difference

The **difference** of two sets can be done using `.difference(set)` method or the – operator.

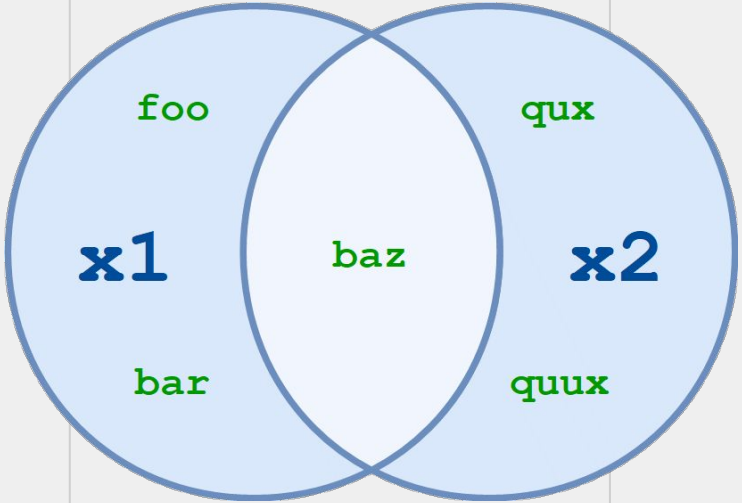



```
x1 = {'foo', 'bar', 'baz'}
x2 = {'qux', 'quy', 'baz'}

x3 = x1.difference(x2)
print(x3)
>   {'foo', 'bar'}
```

Set Operations (5)

XOR

The **XOR** can be used on sets thanks to the `.symmetric_difference(set)` method or the `^` operator.




```
x1 = {'foo', 'bar', 'baz'}
x2 = {'qux', 'quy', 'baz'}

x3 = x1.symmetric_difference(x2)
print(x3)
>   {'qux', 'foo', 'quy', 'bar'}
```

Set Operations (6)

Updated operations

The Union, Intersection, Difference and the XOR functions have a **variant that updates** the set instead of returning a new one :

- `set.update(set)` or `|=`
- `set.intersection_update(set)` or `&=`
- `set.difference_update(set)` or `-=`
- `set.symmetric_difference_update(set)` or `^=`

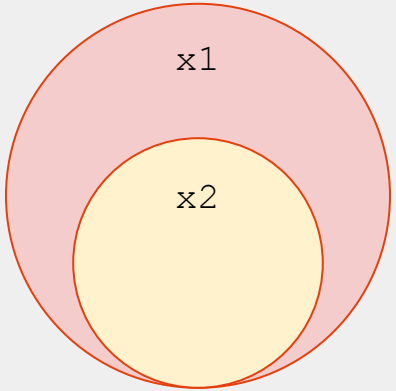
```
x1 = {0, 1, 2}
x2 = {2, 3, 4}

x1.intersection_update(x2)
print(x1)
>   {2}
```

Subsets

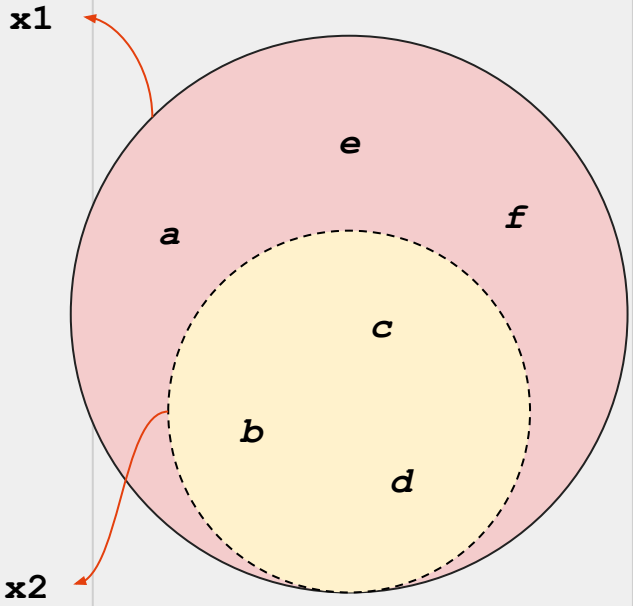
Sets can be **subsets** of other sets :

- Use `x1.isdisjoint(x2)` to check if `x1` & `x2` have no common elements ;
- Use `x1.issubset(x2)` or >= to check if `x1` is included in `x2` ;
- Use `x1.issuperset(x2)` or <= to check if `x2` is included in `x1`.

The operator > and < can also be used to compare sets. They return False if the sets are the same.



Subsets (2)




```
x1 = {'a', 'b', 'c', 'd', 'e', 'f'}
x2 = {'b', 'c', 'd'}

print(x1.isdisjoint(x2))
>   False

print(x1.issubset(x2))
>   False

print(x1.issuperset(x2))
>   True
```

Frozen Set

A **frozen set** is a completely **immutable** set i.e. elements cannot be removed or added to a frozen set. Thus, the **update** methods will not work on it.

```
# On a set
x1 = frozenset({0, 1, 2})
print(x1)
>   frozenset({0, 1, 2})

# On a list
x2 = frozenset([2, 3, 4])
print(x2)
>   frozenset({2, 3, 4})
```

Exercises

Do the following exercise sheets :

- Data Structures p.II

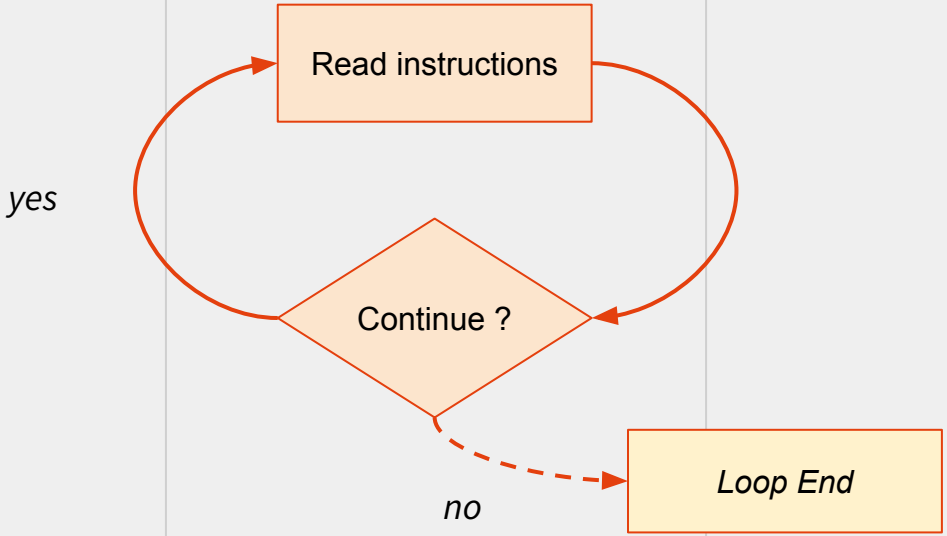
Loops

Table of Content

The following section is divided into four parts :

Definition

Iterations, also called **iterative structures**, are structures used to repetitively read a **loop** that contains instructions until completeness.



Definition (2)

In programming, there are two types of iteration :

- **Indefinite iteration** - the number of times the loop is executed isn't specified explicitly in advance. Rather, the designated block is executed repeatedly as long as some **condition** is met.
- **Definite iteration** - the number of times the designated block will be executed is specified explicitly at the time the loop starts.

Definition (3)

More specifically, the loops will be executed :

- a specified **number of times**

  Ex : the loop will be executed ten times.

- according to a **collection** of items

  Ex : the loop will be executed for each of the 10 elements of the list

- according to some **conditions** at the beginning, the middle or the end

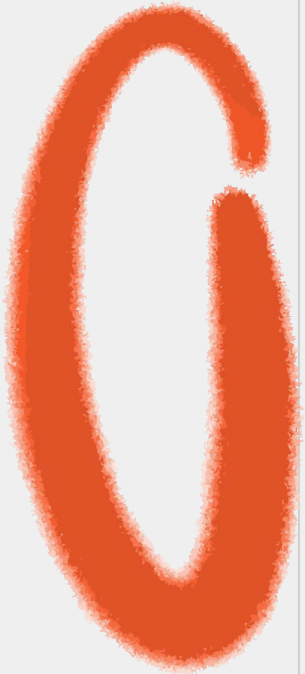
  Ex : the loop will be executed until a condition is met

Types of loops

Many types of **loop** exist, the most common ones are :

- The **while** loop
- The **do while** loop
- The **for** loop
- The **for each** loop

A lots of loop variants exist such as the **infinite loop**, the **continuation loop** that can skip some iterations or the **exit early loop** that checks if the loop has been completely executed or not.



while

& variants

while

The **while loop** is based on a **condition** *(see the conditional structure part)*.

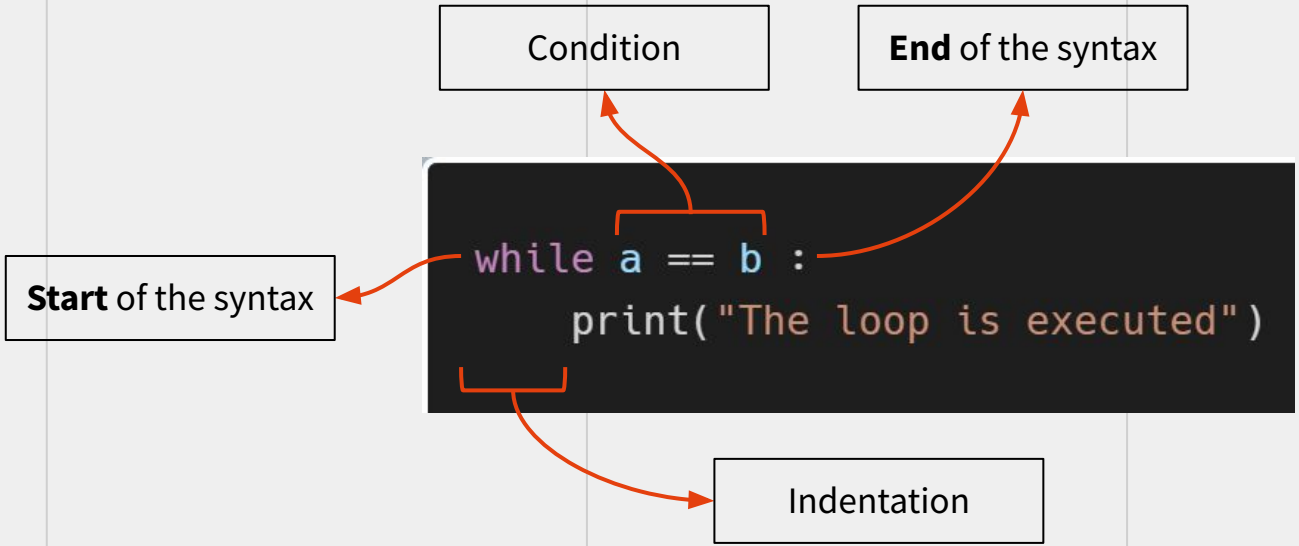
Instructions are executed given a condition at the start of the loop :

- To **start** the iteration, the **loop condition must be met** ;
- To **end** the iteration, the **loop condition must not be met**

Python has an in-built while loop.

```
while a == b :
    print("The loop is executed")
```

while (2)

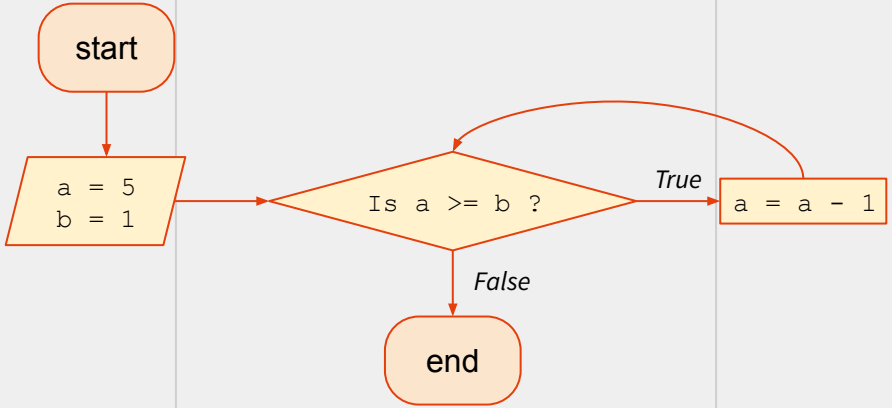



```
while condition :
```

while (3)

```
a = 5
b = 1
while a >= b :
    a -= 1
```

The “first” condition is met, thus the while loop is executed until the condition is not met anymore



Assignments

Python, as well as many other programming languages, has in-built operators that combines the = with another operator.

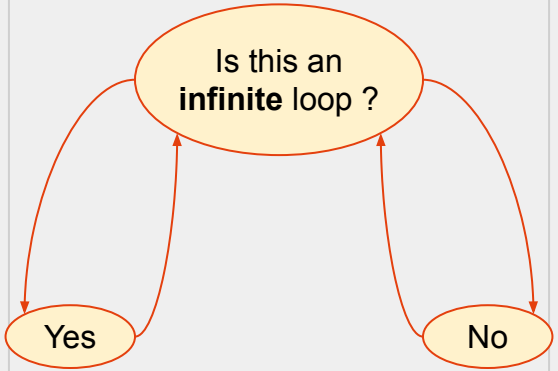
a += b	a = a + b
a -= b	a = a - b
a /= b	a = a / b
a //= b	a = a // b
a *= b	a = a*b
a **= b	a = a**b
a %= b	a = a%b

Booleans operations

Or	**a |= b**	a = a | b
And	**a &= b**	a = a & b
Xor	**a ^= b**	a = a ^ b

Infinite Loops

Beware of the **infinite loops** when using the `while` : an ill-defined condition can make the loop to run indefinitely !



do while

The **do while loop** is also based on a **condition** *(see the conditional structure part)*.

Contrary to the regular while loop, instructions are executed given a condition at the **end of the loop** :

- The first loop **starts automatically** ;
- The iteration is stopped when the **end condition is not met anymore**

Python do not have an in-built do while loop.

do while (2)

The **do while loop** can still be implemented in Python using by combining an infinite loop and the `break` **statement** that stops a loop whenever it is read by the program.

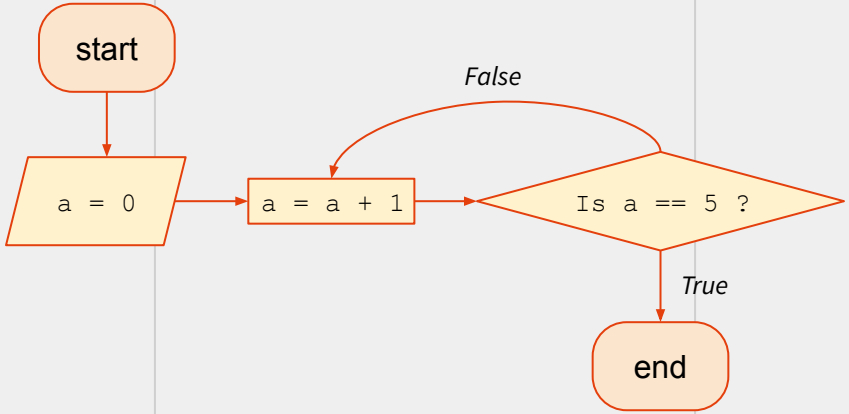
`while True` is an infinite loop - the loop is always executed at least once.

```
a = 0
while True :
    a += 1
    if a == 5 :
        break
```

A condition at the end of the loop will trigger the exit of the loop using the `break`

do while (3)

```
a = 0
while True :
    a += 1
    if a == 5 :
        break
```



continue & pass

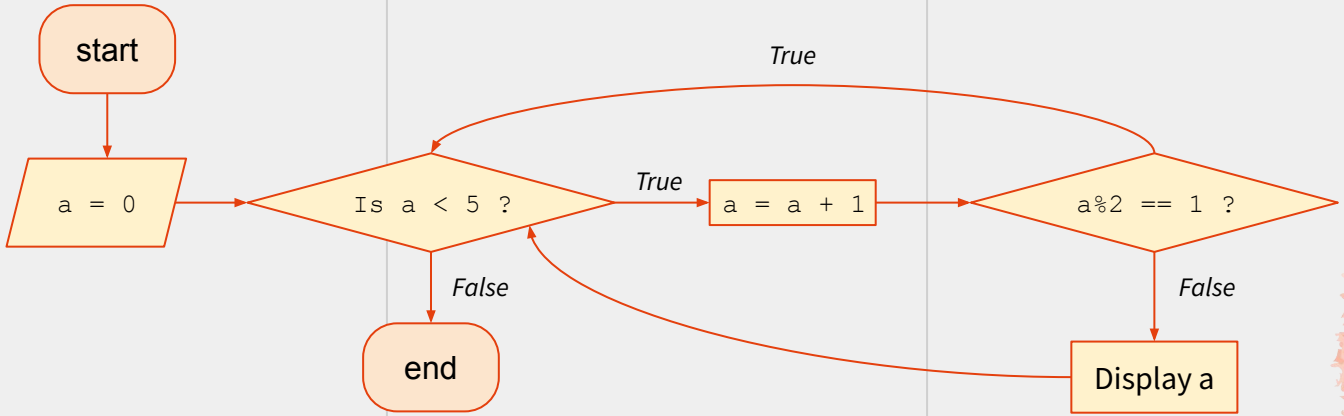
Two other Python **statements** can be used in loops :

- `continue` - it skips the current loop to the next one (**continuation loop**)
- `pass` - it does nothing

```
a = 0
while a < 5 :
    a += 1
    if a % 2 == 1 :
        continue
    print(a)

> 2
> 4
```

continue & pass (2)

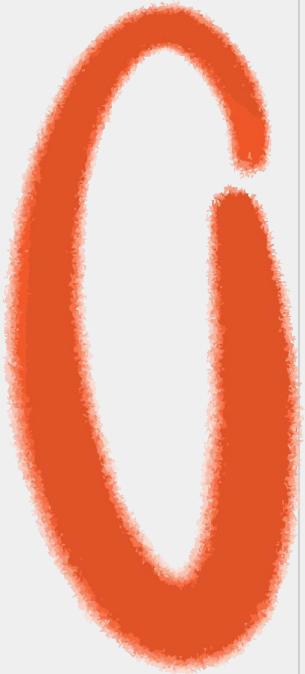
```python
a = 0
while a < 5 :
    a += 1
    if a % 2 == 1 :
        continue
    print(a)
```



Exercises

Do the following exercise sheets :

- Loops p.I



for

Types of for loops

The **for loop** is not based on a condition. Instead, the for loop is executed a **number of time** that is known in advance.

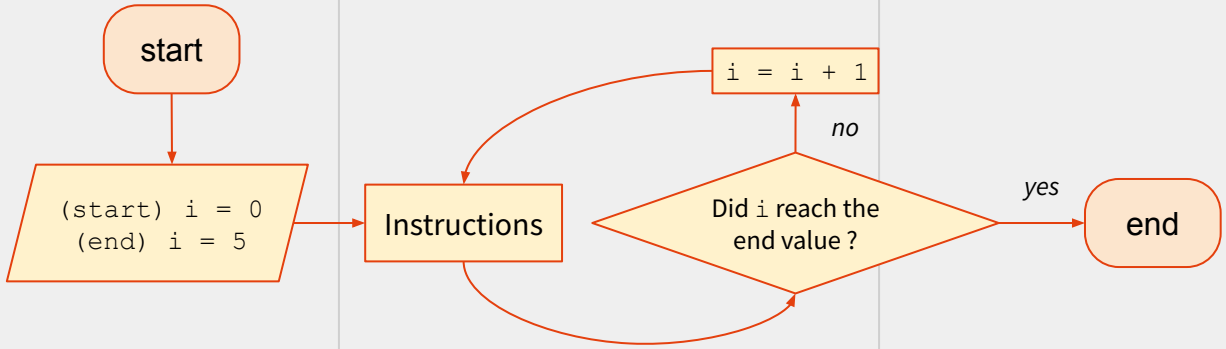
For loops have three types :

- **Numeric Range** Loop
- **Three-Expression** Loop
- **Iterator-Based** Loop

Numeric Range

Numeric Range loops are starting from a number and ending to another number.

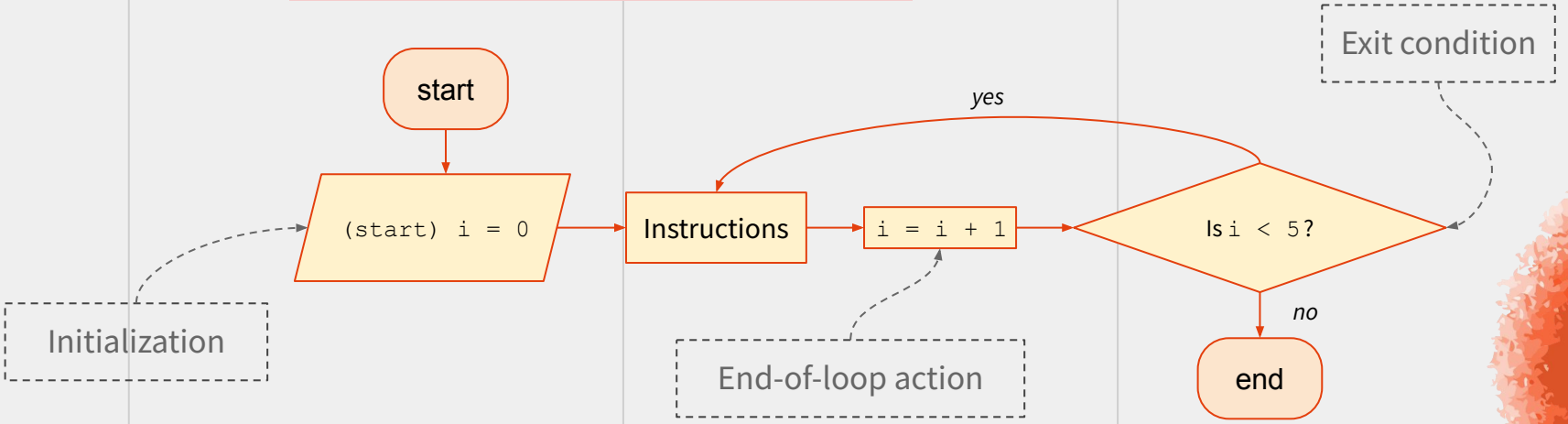
There is no in-built numeric range for loop in Python.



Three-Expression

Three-Expression loops are very flexible and are very popular (C, C++, Java …). They have an **initialization** (generally a number but not always), **an exit condition** and and an **end-of-the-loop action**.

There is no in-built three-expression for loop in Python.



Iterator-Based

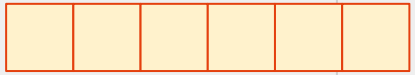
Iterator-Based loops iterates over a **collection of objects** (i.e. a container such as a string, a list, a tuple, a dict, a set, or a frozenset), rather than specifying numeric values or conditions.

Fortunately, there is a built-in Iterator-Based loop in Python and **Python iterators are so much versatile** that it can also be used to perform numeric range and three-expression loops !

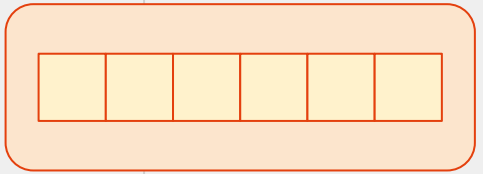
Iterator & Iterable

An **iterable** is a synonym for a **collection of objects** such as a list, a tuple, a dictionary or a set.

An **iterator** is always associated to an iterable object. In short, it is a value producer that **yields successive values** from its associated iterable object. The iterator will return the object elements one after another.

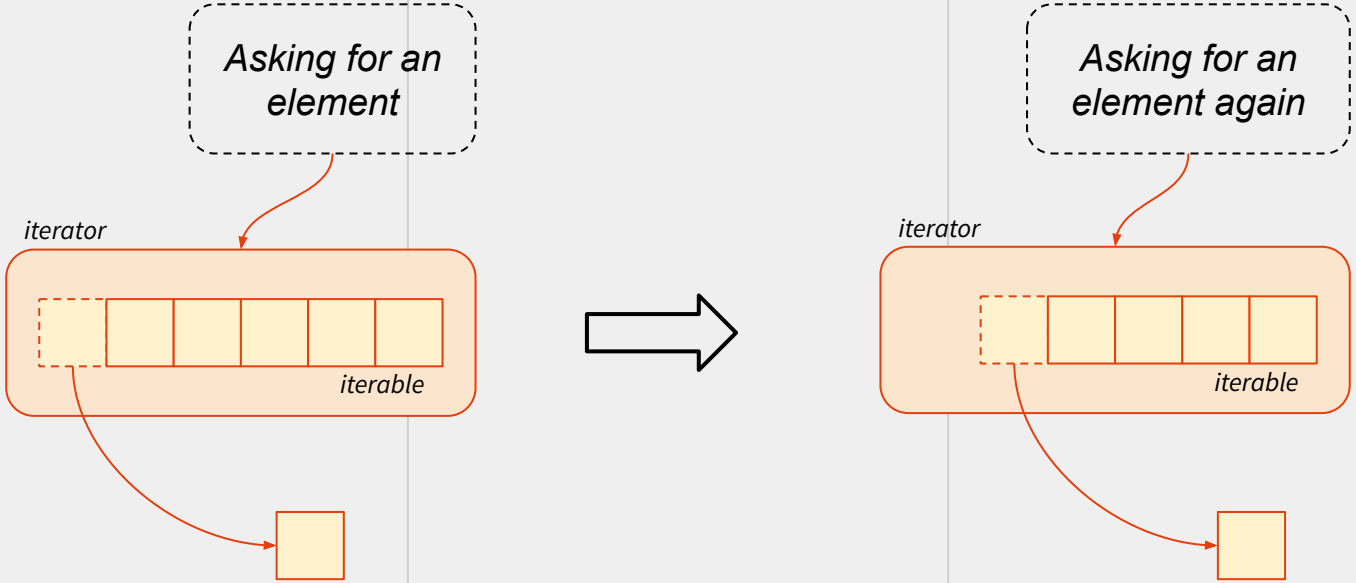


An iterable
(ex : list, tuple …)



An iterator

Iterator & Iterable (2)



Iterator & Iterable (3)

Every single collection of objects can be placed inside an **iterator** using the `iter()` function of Python.

```
mylist = [0,1,2,3,4,5,6]
iterator = iter(mylist)
print(iterator)
>   <list_iterator object at 0x000001A3A62B62F0>

mytuple = (0,1,2,3,4,5,6)
iterator = iter(mytuple)
print(iterator)
>   <tuple_iterator object at 0x000001D488056CB0>
```

Iterator & Iterable (4)

To use the iterator, Python has a `next()` function that asks the iterator to return the next element of the iterable.

```
mylist = [0,1,2]
iterator = iter(mylist)

print(next(iterator))
>   0

print(next(iterator))
>   1

print(next(iterator))
>   2
```

Iterator & Iterable (5)

When the iterator reaches the end of the iterable and the `next()` function continues to ask for an element, it returns a **StopIteration error**.

```
Traceback (most recent call last):
  File "....", line ..., in <module>
    print(next(iterator))
StopIteration
```

Iterator & Iterable (6)

The iterator can be **converted back to an container** using a function such as `list()` or `tuple()`. These functions are forcing the iterator to return all of its iterable elements at the same time : thus, the iterator **is consumed**.

```
mylist = [0,1,2,3]
iterator = iter(mylist)

# Convert the iterator to a list
print(list(iterator))
>   [0,1,2,3]

# The iterator has been consumed
print(list(iterator))
>   []
```

Iterator & Iterable (7)

Python treats looping over all iterables in exactly this way, and in Python, iterables and iterators abound:

- Many **built-in** and **external objects** (library objects) are **iterable**.
- There is a **Standard Library module*** named **itertools** containing many functions that return iterables.
- Python features a construct called a **generator** that allows you to create your own iterator.

** a Library module is a set of tools that can be installed and used in Python*

for each

In a Python for loop, you can use :

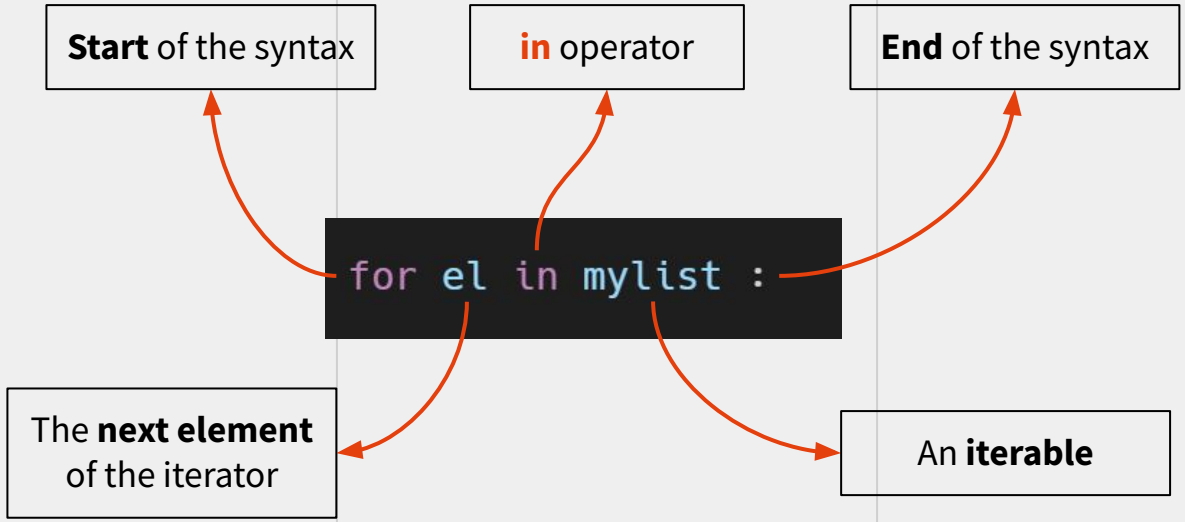
- An **iterator** - Python will repeatedly call `next()` to get the elements until exhaustion (StopIteration)
- An **iterable** - Python will first convert the iterable into an iterator and then proceed as above

```
mylist = [0, 1, 2]
for el in mylist :
    print(f"mylist contain the {el} element")
```

for each (2)

iterable

Don't forget the indentation after the colon !

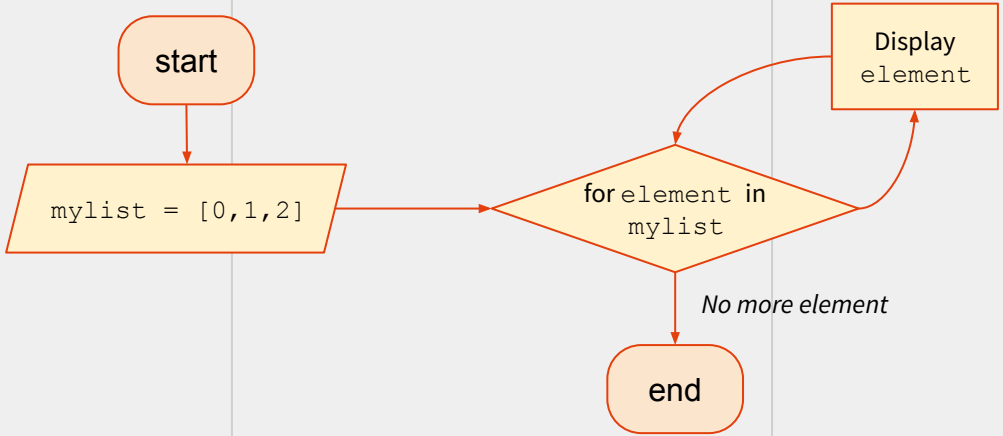



```
for element in iterable/iterator :
```

for each (3)

iterable

```
mylist = [0, 1, 2]
for el in mylist :
    print(el)
```



for each & for loops

iterator

Similarly, Python for loop works with any iterator.

```
iterator = iter([0, 1, 2])
for el in iterator :
    print(el)
```

Python has a special in-built iterator named `range()` that returns integers.

```
for i in range(10) :
    print(i)
```

range

Python `range()` iterable can be used in three different ways :

- `range(n)` - integers range from `0` to `n-1` included
- `range(k,n)` - integers range from `k` to `n-1` included
- `range(k,n,s)` - integers range from `k` to `n-1` with a step of `s`

Be careful ! **The last integer will never be given by the range iterator**.

Ex : range(10) will give the elements from 0 to 9 included.

Range (2)

The default **minimum** integer is `0` and the default **step** is `1`.

```
for i in range(5) :
    print(i)

>   0
>   1
>   2
>   3
>   4
```

```
for i in range(1, 5) :
    print(i)

>   1
>   2
>   3
>   4
```

```
for i in range(0, 5, 2) :
    print(i)

>   0
>   2
>   4
```

Range (3)

The **minimum** and the **maximum** integers can be **negative**.

If the maximum integer is lower than the minimum, then the **step** must also be **negative**.

```
for i in range(-2, 2) :
    print(i)

>   -2
>   -1
>   0
>   1
```

```
for i in range(5, 0, -1) :
    print(i)

>   5
>   4
>   3
>   2
>   1
```

Range (4)

Range can also be used to get an iterable of integers.

```
mylist = list(range(10))
print(mylist)
>   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

mytuple = tuple(range(10))
print(mytuple)
>   (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

myset = set(range(10))
print(myset)
>   {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
```

break, continue & pass

The **break**, the **continue** and the **pass** statements can also be implemented in a Python **for loop**.

```
for i in range(10) :
    if i == 4 :
        break
    print(i)

>   0
>   1
>   2
>   3
```

```
for i in range(5) :
    if i % 2 == 1 :
        continue
    print(i)

>   0
>   2
>   4
```

```
for i in range(5) :
    if i % 2 == 1 :
        pass
    print(i)

>   0
>   1
>   2
>   3
>   4
```

Enumerate

Enumerate is a python function, `.enumerate()`, that allows to iterate over a range iterator and an iterable/iterator **at the same time**.

```
mylist = ['a', 'b', 'c']
for i, element in enumerate(mylist):
    print(i, element)

>   0 a
>   1 b
>   2 c
```

Zip

Zip is a python function, `.zip()`, that allows to iterate over **multiple iterables (or iterators) at the same time**.

```
mylist = ['a', 'b', 'c']
mytuple = (10, 20, 30)
for list_el, tuple_el in zip(mylist, mytuple):
    print(list_el, tuple_el)

>   a 10
>   b 20
>   c 30
```

Dictionary

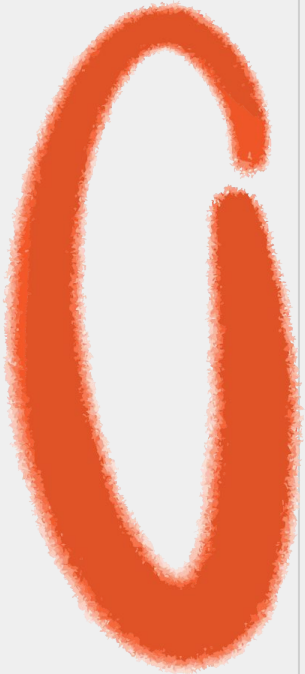
Dictionaries can be used inside for loops. By default, the for loop iterates over the keys. The `.values()` should be used to get the values. The `.items()` can be used to get both the keys and the values.

```
mydict = {'a':0, 'b':1, 'c':2}
for key,value in mydict.items():
    print(key,value)

>   a 0
>   b 1
>   c 2
```

Exercises

Do the following exercise sheets :

- Loops p.II



Side Notes

Nested for loops

It is possible to create **nested for loops**. The elements of the first loops can be used inside the instructions of the deeper loops. The indentation should always be preserved !

```
for i in range(2):
    for j in range(2):
        print(f"i = {i} - j = {j}")

>   i = 0 - j = 0
>   i = 0 - j = 1
>   i = 1 - j = 0
>   i = 1 - j = 1
```

Nested for loops (2)

Instructions can be written on every nested loop.

```
for i in range(2):
    print(f"i = {i}")
    for j in range(2):
        print(f"j = {j}")

>   i = 0
>   j = 0
>   j = 1
>   i = 1
>   j = 0
>   j = 1
```

Nested while loops

It is also possible to create **nested while loops**. Nested while loops are less frequent.

```
a,b = 0,0
while a < 3 :
    a += 1
    while b < 3 :
        b += 1
        print(f"a = {a} - b = {b}")

>   a = 1
>   b = 1
>   b = 2
>   b = 3
>   a = 2
>   a = 3
```

Loop combination

It is possible to **combine** different type of loops.

```
a = 0
while a < 10 :
    for i in range(3):
        a += 1
    print(a)

>   3
>   6
>   9
>   12
```

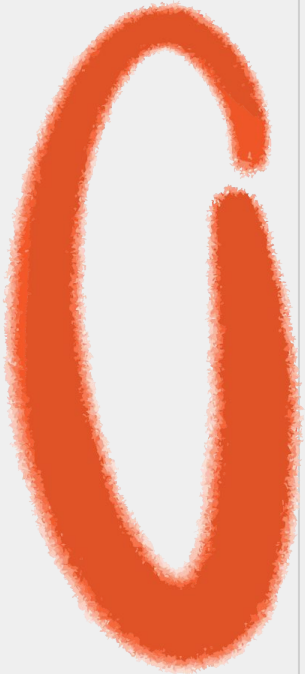
while vs for

While loops should be used whenever the number of loops is **unknown** in advance.

For loops should be used whenever the number of loops is **known** in advance.

While loops are significantly slower than for loops.

Thus, they should only used whenever they are needed.



Comprehension

Definition

Comprehensions is a way to **build containers** such as lists, sets or dictionaries.
Python supports the following 4 types of comprehensions:

- **List** Comprehensions
- **Dictionary** Comprehensions
- **Set** Comprehensions
- **Generator** Comprehensions

Advantages

Comprehensions is advantageous for many reasons:

- It's a single tool that can be **used in many different situations** : container creation, mapping, filtering …
- Comprehensions are also **more declarative than loops** i.e. they're easier to read and understand.

Basic Usage

Regular for loop
to create a list

```
new_list = []

for i in range(10):
    new_list.append(i)
```

List comprehension
to create a list

```
new_list = [i for i in range(10)]
```

```
[element for element in iterable]
```

Conditions

Filtering

Regular for loop
to create a list

```
new_list = []

for i in range(10):
    if i%2 == 0 :
        new_list.append(i)
```

List comprehension
to create a list

```
new_list = [i for i in range(10) if i%2 == 0]
```

```
[element for element in iterable if condition]
```

Conditions (2)

Mapping

Regular for loop
to create a list

```
new_list = []

for i in range(10):
    if i > 4 :
        new_list.append(0)
    else :
        new_list.append(1)
```

List comprehension
to create a list

```
new_list = [0 if i > 4 else 1 for i in range(10)]
```

```
[a if condition else b for element in iterable]
```

Set & Dictionary

Set & dictionary comprehensions work exactly the same as list comprehension.

Set comprehension

```
new_set = {i for i in range(10)}
```

Dict comprehension

```
new_dict = {i:i**2 for i in range(10)}
```

Exercises

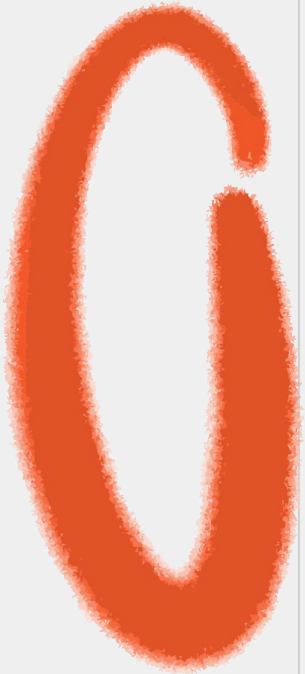
Do the following exercise sheets :

- Loops p.III

Once the aforementioned exercises have been completed :

- Loops p.IV
  - The Rock Paper Scissors
  - The Merchant

Functions



Basics

Definition

In computer programming, a **function is a sequence of instructions** that performs a specific task. This sequence of instruction can then be called wherever that particular task should be performed.

In other words, a **function is a relationship or mapping between one or more inputs and a set of outputs.**

Depending on the programming language, a function can be called a routine, a subprogram, a subroutine, a method, or a procedure. Be careful as these term can describe very different objects.

Definition (2)

In math, a function can be defined by the following:

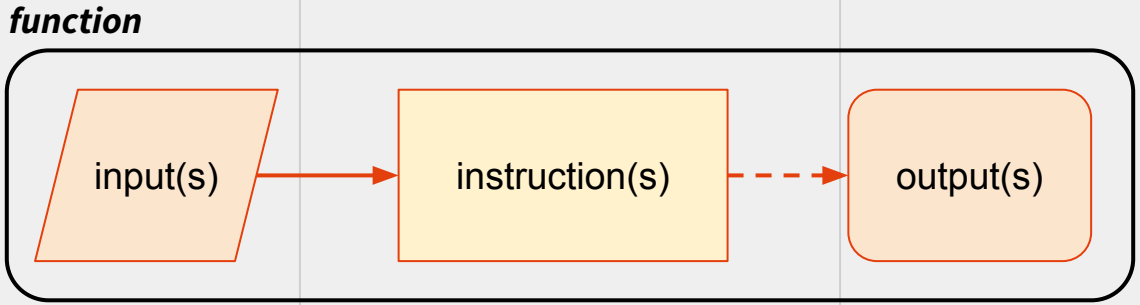
$$f(x) = ax + b$$

Where `x` is the input and `ax+b` is the output.

However, programming functions are much more generalized and versatile than this mathematical definition.

Definition (3)

A programming function can be seen as the following:



A function may or may not include an output. If the function do not output anything, this function is called a **subroutine**.

Advantages

Reusability

Suppose you write some code that does something useful.

As you continue development, you find that the task performed by that code is one you need often, in many different locations within your application.

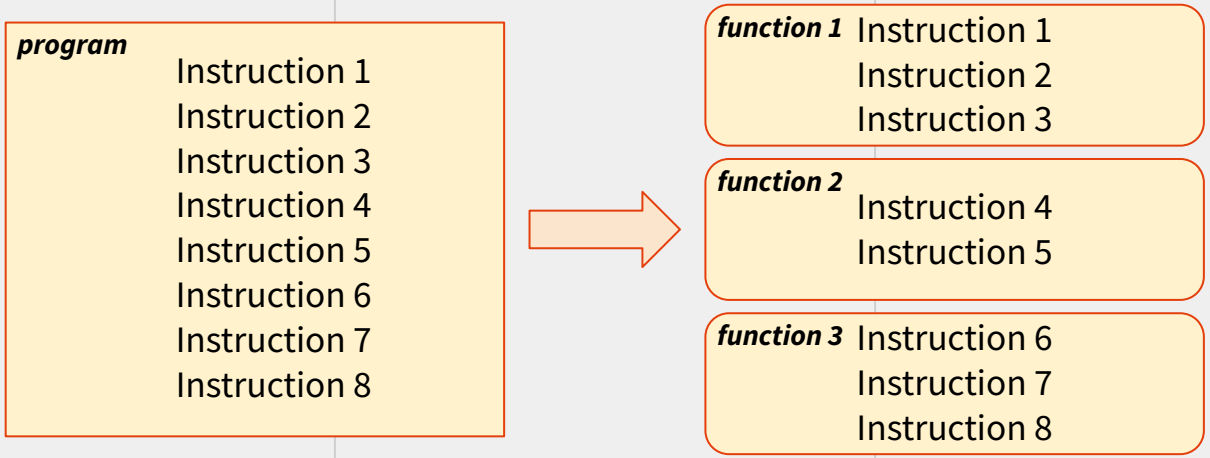
What should you do?

- You could just copy-and-paste the code where you need it. This way of programming, i.e. using a top-down approach, is called **procedural programming**.
- Or you could define a function that use the code and then call the function whenever you need it. That way you would only write the code once ! This way of programming is called **functional programming**.

Advantages (2)

Modularity

Functions allow complex processes to be broken up into smaller blocks. In other words, instead of all the code being strung together, it's broken out into separate functions, each of which focuses on a specific task.



Advantages (3)

Modularity

In real life, you do this sort of thing all the time, even if you don’t explicitly think of it that way.

For example, if you want to clean the shelves of your apartment the best way is to divide the task into smaller ones, this process is called **modularity**:

- **Take** all the stuff off one shelf.
- **Remove** dust from the stuff.
- **Remove** dust and other piece of dirt from the shelf.
- **Clean** the shelf with a towel. Let it dry.
- **Put** the stuff back on the shelf.

If you have many shelves, this process can be used on each of them using these “functions”.

Advantages (4)

Namespace Separation

A **namespace** is a region of a program in which identifiers have meaning. As you'll see later on this course, when a Python function is called, a new namespace is created for that function, one that is distinct from all other namespaces that already exist.

Thus, **variables can be defined and used within a Python function** even if they have the same name as variables defined in other functions or in the main program.

Advantages (4)

Namespace Separation

A **namespace** is a region of a program in which identifiers have meaning. As you'll see later on this course, when a Python function is called, a new namespace is created for that function, one that is distinct from all other namespaces that already exist.

Thus, **variables can be defined and used within a Python function** even if they have the same name as variables defined in other functions or in the main program.

Function types

There are two types of function in Python programming:

- **Standard library functions** - ready-to-use functions in Python.
- **User-defined functions** - custom functions based on our requirements

You already know some built-in functions: `print()`, `list()`, `dict()`, `len()`, `enumerate()`, `zip()`, `input()`, `format()` and so on are all Python built-in functions.

As you may have noticed, all of these functions are using a typical **parenthesis** syntax.

Custom Function

In Python, you can **create a function** using the following syntax:

```
def <function_name>(<parameters>):
    <statement(s)>
```

- **def** is the keyword that informs Python that a function is being defined ;
- **<function_name>** is a valid Python identifier that names the function ;
- **<parameters>** is an optional, comma-separated “list” of parameters ;
- **:** is the end of the function syntax ;
- **<statement(s)>** is a block of Python instructions.

Custom Function (2)

In Python, you can **call a function** using the following syntax:

```
<function_name>(<argument(s)>)
```

Where **<argument(s)>** are the values passed to parameter of the function, these values must be passed in between parentheses. If there is no parameter, the function must be called with empty parentheses.

Custom Function (3)

A Python example:

```
# Creating the function
def func(a):
    print(f"The argument is a = {a}")

# Calling the function
print("before the function")
output = func(a=5)
print("after the function")
```

output

```
>>> "before the function"
>>> "The argument is a = 5"
>>> "after the function"
```

Argument Passing

In most programming languages, passing an argument to a function can be done in two ways:

- By **reference** - in this case, the argument reference (memory location) will be used in the instructions the function. It means that the argument can be modified outside of the function.
- By **value** - in this case, the function will use a copy of the argument. It also means that the argument cannot be changed outside of the function.

In Python, passing an immutable object such as int, str or tuple to function acts like pass-by-value. The function can’t modify the object in the calling environment. Passing a mutable object such as list, dict or set acts like pass-by-reference. The function can change items in place within the object.

Argument Passing (2)

Positional Arguments

Most functions will have parameters to take arguments as inputs. The most straightforward way to pass arguments to a Python function is with **positional arguments**.

The number of arguments passed to the function when calling it should match the number of parameters. If the function asks for 3 parameters a, b and c then the function should be called with three arguments (one for each).

The **order of the arguments matter** !

If you give more arguments than parameters (or less), a **TypeError** will occur and tell what the problem is.

Argument Passing (3)

Positional Arguments


```
def func(a,b,c):
    print(f"The arguments are a={a}, b={b} and c={c}")

# Calling the function
func(1,'Blue',3.14)
>>> "The arguments are a=1, b=Blue and c=3.14"
```

Argument Passing (4)

Keyword Arguments

You can also include the **parameter keyword** before the argument when calling the function.

```
def func(a,b,c):
    print(f"The arguments are a={a}, b={b} and c={c}")

# Calling the function
func(a=1,b='Blue',c=3.14)
>>> "The arguments are a=1, b=Blue and c=3.14"
```

Argument Passing (5)

Default arguments

You can also include **defaults arguments** to the parameters when defining a function.

Then, the arguments of these **parameters will become optional** and the function can be call with less arguments than parameters.

The defaults arguments must be placed on the latest parameters (position-wise) when creating the function.

```
def func(a, b, c=0):
    print("Hello World")
```

```
def func(a=0, b, c):
    print("Hello World")
```

Argument Initialization

The default parameter values of a function are **defined once** when the `def` statement is executed. This means that repeating calls of a function will not use more memory to store the default values.

It is a convenient feature. However, if the function contains a **mutable default argument** such as a list, any function call will refer to the same object.

```
def func(mylist=[]):
    mylist.append(0)
    print(mylist)

func()
func()
func()
```

```
>>> [0]
>>> [0, 0]
>>> [0, 0, 0]
```

Return statement

The **return** statement allows the function :

- to **return values** (not print);
- to **exit** the function.

Most of the time, the return statement is used to return a value. If no return statement is used, the function always returns None.

```
def func(a):
    b = a**2
    return

output = func(5)
print(output)
>>> None
```

```
def func(a):
    return a**2

output = func(5)
print(output)
>>> 25
```

Return statement (2)

There is **no limit to the number of return statements** in a function. Thus, a function can have multiple exits.

```
def func(a)
    if a > 10 :
        return "a is higher than 10"
    elif a == 10 :
        return "a is equal to 10"
    else :
        return "a is lower than 10"

output = myfunc(5)
print(output)
>>> "a is lower than 10"
```

Return statement (3)

Multiple values can be returned with the return statement using comma separators. In that case, the output of the function is a **tuple** that can directly be unpacked.

```
def func(a)
    return a**2, a*2

output = myfunc(5)
print(output)
>>> (25, 10)
```

```
def func(a)
    return a**2, a*2

out1, out2 = myfunc(5)
print(out1, out2)
>>> (25, 10)
```

Docstring

A documentation can be added to any custom function using the triple quotes. This documentation can be accessed using the `__doc__` method.

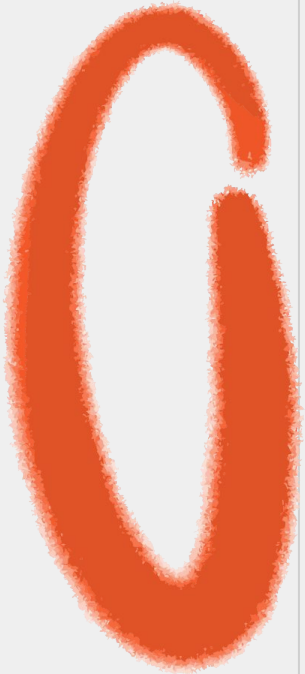
```
def func(a):
    """This is the function documentation.
        New lines, spaces, punctuations as well as code
        examples can be added there.
        Code example :
            for i in range(x):
                print(i)"""
    return a**2
```

```
print(func.__doc__)
```

Storing functions

Functions can be stored in variables. These variables can then be used to call the function because it refers to the function object.

```
def func(a):
    return a**2

myfunc = func
print(myfunc)
>>> <function func at 0x000001BE00E3F040>

output = myfunc(5)
print(output)
>>> 25
```



Advanced

Argument Unpacking

args

`*args` and `**kwargs` allow you to pass multiple arguments or keyword arguments to a function without specifying them. `*args` specifically allows you to pass as many arguments as you want. These arguments will be stored in a tuple. Thus, you can use index, slice or even iterate over the args.

```
def func(*args):
    print(args)
    for arg in args :
        print(arg, end= ' ')

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

Argument Unpacking (2)

kwargs

`**kwargs` works just like `*args`, but instead of accepting positional arguments it accepts keyword (or named) arguments. These arguments will be stored in a dictionary.

```
def func(**kwargs):
    print(kwargs)
    for key,value in kwargs.items() :
        print(f"{key} --> {value}", end= ' / ')

func(a=1,b=2,c=3)
>>> {'a': 1, 'b': 2, 'c': 3}
>>> a --> 1 / b --> 2 / c --> 3
```

Argument Unpacking (3)

args & kwargs

Of course, you can use both `*args` and `**kwargs` in a function.

```
def func(*args, **kwargs):
    print(args)
    print(kwargs)

func(0,1,a=2,b=3,c=4)
>>> (0, 1)
>>> {'a': 2, 'b': 3, 'c': 4}
```

Lambda Expressions

The **Lambda expression** is an anonymous function which purpose is to live inside larger expressions representing a computation. In contrast to a normal function, a Python lambda function is a single expression.

```
lambda input(s) : output
```

```python
# Regular function
def func(x):
    return x**2

# Lambda function
lambda_func = lambda x:x**2
```

Lambda Expressions (2)

Like a normal function object defined with `def`, Python lambda expressions support all the different ways of passing arguments : positional arguments, named arguments, defaults, args, kwargs and any combination of these.

```
lambda_func = lambda x : x**2

lambda_func = lambda x,y,z : x+y+z

lambda_func = lambda x,y,z=3 : x+y+z

lambda_func = lambda *args: sum(args)

lambda_func = lambda **kwargs: sum(kwargs.values())
```

Lambda Expressions (3)

Lambdas can also be used inside of other functions. For example, a normal function can output a lambda function.

x =? y=4 z=?

x =5 y=4 z=?

x =5 y=4 z=1

```
def func(x):
    y = 4
    return lambda z: x + y + z

lbda = func(5)

print(lbda)
>>> <function func.<locals>.<lambda> at 0x000001C93FC39CA0>

print(lbda(1))
>>> 10
```

Lambda Expressions (4)

Similarly to comprehension, lambda expression should be overused. In particular, they should not be used if:

- It doesn't follow the Python style guide (*PEP 8*)
- It is too complicated.
- It is difficult to read.

Lambda expression are mainly used along with some **inbuilt python functions** such as `filter()`, `map()`, `sorted()` or `min()`.

filter()

Filter is an in-built function that allow to filter any iterable given a logic function. The logic function can be a regular function or a lambda and should return True or False. Filtering can also be done be using a comprehension.

```
mylist = [1,2,3,4,5,6]
even = lambda x: x%2 == 0
l_filtered = filter(even, mylist)

print(l_filtered)
>>> <filter object at 0x0000019CB1561AF0>
print(list(l_filtered))
>>> [2, 4, 6]
```

map()

Map is an in-built function that allow to map a function to any iterable (i.e. **it modifies the values of the iterable**). The logic function can be a regular function or a lambda and can return anything. Mapping can also be done be using a comprehension or a regular function.

```
mylist = ['cat', 'dog', 'cow']
l_mapped = map(lambda x: x.capitalize(), mylist)

print(l_mapped)
>>> <map object at 0x0000015FBE611AF0>
print(list(l_mapped))
>>> ['Cat', 'Dog', 'Cow']
```

sorted()

Sorted is used to sort any iterable and returns a list. This in-built function also accepts a parameter `key` that allows to **customize the sorting step**.

```
mylist = ['id1', 'id2', 'id30', 'id3', 'id22', 'id100']

sorted_l = sorted(mylist)
print(sorted_l)
>>> ['id1', 'id100', 'id2', 'id22', 'id3', 'id30']

sorted_l = sorted(mylist, key = lambda id : int(id[2:]))
print(sorted_l)
>>> ['id1', 'id2', 'id3', 'id22', 'id30', 'id100']
```

Sorted according to the string order

Sorted according to the integer part

Inner functions

Inner functions, also known as **nested functions**, are functions that you define inside other functions. They are used in order to Provide encapsulation and hide functions from external access, write helper functions to facilitate code reuse and even create closure factory functions that retain state between calls.

```
def outer_func():
    def inner_func():
        print("Hello, World!")
    inner_func()

outer_func()
>>> "Hello World!"
```

Inner functions (2)

Encapsulation

A common use case of inner functions arises when you need to protect, or hide, a given function from everything happening outside of it so that the function is totally hidden from the global scope. This kind of behavior is commonly known as **encapsulation**.

You can't call `inner_increment()`

```
def increment(number):
    def inner_increment():
        return number + 1
    return inner_increment()

print(increment(5))
>>> 6
```

Inner functions (3)

Closure Factory Functions

In Python, when you return an inner function object, the interpreter packs the function along with its containing environment or closure.

The function object keeps a **snapshot of all the variables and names** defined in its containing scope. To define a closure, you need to take three steps:

- Create an inner function.
- Reference variables from the enclosing function.
- Return the inner function.

Inner functions (4)

Closure Factory Functions

```
def generate_power(exponent):
    def power(base):
        return base ** exponent
    return power

powerof2 = generate_power(2)
powerof3 = generate_power(3)

print(powerof2(3))
>>> 9

print(powerof3(3))
>>> 27
```

Inner functions (5)

Closure Factory Functions

```
def mean():
    sample = []
    def inner_mean(number):
        sample.append(number)
        return sum(sample) / len(sample)
    return inner_mean

sample_mean = mean()
print(sample_mean(1))
print(sample_mean(2))
print(sample_mean(2))
print(sample_mean(2))
```

```
>>> 1.0
>>> 1.5
>>> 1.6666666666666667
>>> 1.75
```

Others

There are a lot more functionalities to Python functions:

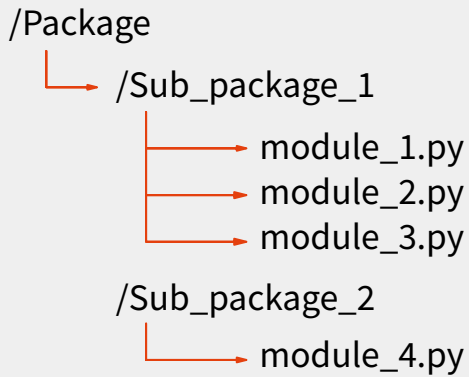
- **Generators** - a generator function is a special kind of function that returns a lazy iterator. It is a lazy iterator because it does not store its contents in memory.
- **Decorators** - a decorator function takes another function and extends the behavior of the latter function without explicitly modifying it.
- **Recursion** - a recursive function calls itself. It make the code simpler by limiting the usage of loops

Modules & Packages

Introduction

A **module** is a file containing definitions and statements. A module can define functions, classes, and variables. A module can also include runnable code. Essentially, **a module is a .py file**.

A **package** is a group of modules.



Module

There are three different ways to define a module in Python:

- A module can be written in Python itself.
- A module can be written in C and loaded dynamically at run-time, like the ***re*** module.
- A built-in module is intrinsically contained in the interpreter, like the ***itertools*** module.

Module (2)

The `import` statement is used to import a module definitions and statements.

```
import module_name
```

This statement must be placed before using anything from the imported modules. Generally, import statements are placed at the beginning of the Python script.

```
1 import module_1
2 import module_2
3
4 """
5 Script Instructions
6 """
```

Module (3)

Sometimes, the module may not be found by Python. Thus, to ensure your module is found, you need to do one of the following:

- Put the module file in the directory where your script is located
- Or, make sure that the PYTHONPATH environment variable contains the directory where the module is located before starting the interpreter
- Or, put the module file in any directory contained in `sys.path`
- Or, put the module file in any directory of your choice and then modify `sys.path` at run-time so that it contains that directory.

Module (4)

Working directory

The **working directory** (also known as current directory) is the folder where you are currently working with your main Python script.

Any module placed on this working directory can be called inside of your main Python script.

/MainFolder
- main_script.py
- module.py

```
### This is the main script ###
import module

"""
Script Instructions
"""
```

Module (5)

Working directory

You can get the **working directory** path by using the `pathlib` module.

```
import pathlib
print(pathlib.Path().absolute())

>>> "C:\\Users\\palisson\\MainFolder"
```

For the VsCode users, you can get your working directory by checking the terminal logs.

Module (6)

The sys module

The `sys` module provides access to some variables used or maintained by the interpreter and to functions that interact strongly with the interpreter.

It can be used to get a list of the search paths for modules.

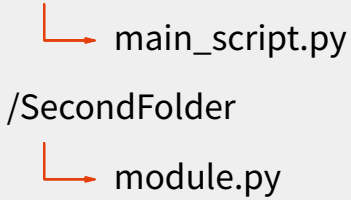
```
import sys
print(sys.path)

>>> ['C:\\Users\\palisson\\MainFolder',
     'C:\\Users\\palisson\\anaconda3\\envs\\python_39\\python39.zip',
     'C:\\Users\\palisson\\anaconda3\\envs\\python_39\\DLLs',
     'C:\\Users\\palisson\\anaconda3\\envs\\python_39\\lib',
     'C:\\Users\\palisson\\anaconda3\\envs\\python_39',
     'C:\\Users\\palisson\\anaconda3\\envs\\python_39\\lib\\site-packages']
```

Module (7)

The sys module

In order to use a module **stored in a different directory** than the main one, you can add it to the list of the search paths obtained with the `sys` module.

Then, and only then, you can import it using the import statement.




```
import sys
sys.path.append("C:\\Users\\palisson\\SecondFolder")
import module
```

Module (7)

Access module content

To access the module content such as a variable or a function, the name of the module must be specified. This helps to avoid any confusion about the source of the content used.

```python
### Module Content ###
var1 = 'Hello World!'
var2 = 1234

def f(x):
    return x**2
```

module.py

```python
### Main Script ###
import module

print(module.var1)
>>> "Hello World !"

res = module.f(5)
print(res)
>>> 25
```

main_script.py

Module (8)

from module import

You can specifically access an object from a module by using the following syntax:

```
from module_name import object_name
```

```
1 ### Main Script ###
2 from module import f
3
4 res = f(5)
5 print(res)
6 >>> 25
```

Overwriting Effect

Module content may overwrite the main script content when using the from module import syntax and vice-versa.

```
1 ### Main Script ###
2 def f(x):
3     return x
4
5 from module import f
6
7 res = f(5)
8 print(res)
9 >>> 25
```

```
1 ### Main Script ###
2 from module import f
3
4 def f(x):
5     return x
6
7 res = f(5)
8 print(res)
9 >>> 5
```

Aliases

Module, and their content, **may be renamed with aliases**. This is particularly useful when the original name is too long or already taken by an object of the main script.

```
### Main Script ###
import module as m

res = m.f(5)
print(res)
>>> 25
```

```
### Main Script ###
from module import f as g
def f(x):
    return x

f_res = f(5)
print(f_res)
>>> 5

g_res = g(5)
print(g_res)
>>> 25
```

Module as a Script

Any **module can be used as a script**. However, it might not be a good idea to add `print` in a module code if it is intended to be used in other scripts because **the module prints will be displayed every time the module is imported**.

```
### Module Content ###
def f(x):
    return x**2

print(f(5))
>>> 25
```

```
### Main Script ###
import module as m
>>> 25

res = m.f(3)
print(res)
>>> 9
```

Module as a Script (2)

__name__ & '__main__'

Fortunately, there is a solution to this printing problem.

When a .py file is imported as a module, Python sets a special variable `__name__` to the name of the module. However, if a .py file is run as a script, `__name__` is set to the string `'__main__'`.

This is particularly useful because it makes possible to **discern the module usage from the script usage**.

Module as a Script (3)

if __name__ == '__main__':

In order to discern the module from the script usage, you can pass the print and any other 'script' instructions inside of a conditional structure that checks if the `__name__` variable is the string `'__main__'` or not.

From now on, this syntax should be added to the end of every single .py file that is intended to be used as a script or as a script and a module at the same time.

```
def f(x):
    return x**2

if __name__ == '__main__':
    print(f(5))
```

Module as a Script (4)

if __name__ == '__main__':

```python
### Module Content ###
def f(x):
    return x**2

if __name__ == '__main__':
    print(f(5))
    >>> 25
```

```python
### Main Script ###
import module as m
# Nothing is printed

if __name__ == '__main__':
    print(m.f(3))
    >>> 9
```

Packages

Suppose you have built a large application that includes multiple modules. For clarity, it may be a good idea to store them into a common folder. Packages may also contain executables. The same methods can be used to call a module and to call a package.

/Package

- module1.py
- module2.py

```
import package.module1                  # import syntax
from package import module1             # from import syntax
import package.module1 as m1            # import with alias
from package import module1 as m1       # from import with alias
from package.module1 import f           # function from module
from package import module1, module2    # multiple module import
```

Packages (2)

Most packages contain a __init__.py file in order to automatically import modules from it. This allows you to import the package without specifying the modules and let you call them afterwards.

```
### __init__.py ###
print("Some info about the package")
import package.module1, package.module2
```

/Package

- __init__.py
- module1.py
- module2.py

```
### Main Script ###
import package as pk
res = pk.module1.f(5)
print(res)
>>> 25
```

Inbuilt modules

Python contains a lot of **inbuilt modules/packages**. Such modules are natively present in Python but, as for every module, they need to be imported in order to use their content. Some of the most used ones:

- **collections**, **functools**, **itertools** — Container, iterator and function tools
- **csv, json**, **pickle**, **zipfile** — Handling file and file types
- **datetime**, **time** & **timeit** — Date and time access and conversions
- **multiprocessing** — Run multiple functions at the same time
- **math**, **random**, **statistics** — Math and stat functions/tools
- **os**, **sys** — OS and system parameters/tools
- **re** — Regular expression operations
- **tkinter**, **turtle** — Graphical user interface

Inbuilt modules (2)

The best part about inbuilt modules is that they can be imported and used right away without any installation.

A complete list of the Python inbuilt modules can be found here : Documentation.

```
1 import math
2 print(math.pi)
3 >>> 3.141592653589793
4
5 import random
6 # a random number in [0,5[
7 print(random.randint(0, 5))
8 >>> 3
```

Third Party modules

Python contains even more third party modules. More than 400.000 packages/modules have been currently deposited on the **PyPi** repository.

Some of the most used Python third party libraries are:

- **Django**, **Fast API**, **Flask** — Web framework
- **Keras**, **PyTorch**, **Scikit-learn**, **Tensorflow** — Machine & Deep Learning
- **Matplotlib**, **Plotly**, **Seaborn** — Data exploration and visualization tools
- **NumPy** — Fast and advanced operations
- **Pandas** — Data analysis and manipulation
- **Requests**, **Urllib3** — HTTP client
- And many more : **Pillow**, **Pytest**, **PyQt5**, **SciPy**, **Bokeh**, **BeautifulSoup** …

Third Party modules (2)

Third Party Modules can be installed using `pip`. Pip is a package manager for Python i.e. it's a tool that allows you to install and manage libraries and dependencies that aren't distributed as part of the standard library.

You can install a module using the Terminal of your interpreter:

```
pip install <package_name>
```

Alternatively, you can install the modules in the windows powershell/MacOs shell:

windows

```
python -m pip install <package-name>
```

macos

```
$ python -m pip install
<package-name>
```