

Introduction à la pensée algorithmique avec Python (2)

📍 Ecole du Louvre, Master Documentation et Huménités Numériques, 2020

Alix Chagué

✉ alix.chague@inria.fr (<mailto:alix.chague@inria.fr>)

👛 Ingénieure Recherche et Développement @ Inria



Crédits

Librement inspiré des [supports de cours \(https://github.com/gguibon/essec-python-1\)](https://github.com/gguibon/essec-python-1) de Gaël Guibon

Librement inspiré des [supports de cours \(https://goo.gl/UFqu2U\)](https://goo.gl/UFqu2U) de Julien Pilla

Retrouver l'ensemble du cours sur 📁 github.com/alix-tz/enc-intro-algo (<https://github.com/alix-tz/enc-intro-algo>) 📁

Syllabus

Plan du cours

- Syllabus
- Comparer des valeurs
- Opérateurs booléens
- Conditions (si... alors)
- Répétition (tant que / répéter x fois)
- Nouveau type : listes
- Nouveau type : dictionnaires
- Aller plus loin

Liens utiles

📁 Simulateur d'environnement Python en mode pseudo-IDE : <https://repl.it/languages/python3> (<https://repl.it/languages/python3>)

📁 Simulateur d'environnement Python en mode console : <https://www.python.org/shell/> (<https://www.python.org/shell/>)

📁 Documentation officielle de Python : <https://docs.python.org/3/> (<https://docs.python.org/3/>)

📁 Visualisateur d'exécution de code Python : <http://pythontutor.com/> (<http://pythontutor.com/>)

Quelques ressources pour continuer à se former

📁 "Automate the Boring Stuff with Python" (en) : <https://automatetheboringstuff.com/> (<https://automatetheboringstuff.com/>)

📁 Leçons dédiées à Python sur *Programming Historian* (en , fr ou es) : <https://programminghistorian.org/en/lessons/introduction-and-installation> (<https://programminghistorian.org/en/lessons/introduction-and-installation>)

📁 "Apprendre à coder avec Python", MOOC de l'Université Libre de Bruxelles (fr) : <https://www.fun-mooc.fr/courses/course-v1:ulb+44013+session04/about> (<https://www.fun-mooc.fr/courses/course-v1:ulb+44013+session04/about>)

📁 "Apprenez à programmer en Python", cours en ligne sur OpenClassroom (fr) : <https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python> (<https://openclassrooms.com/fr/courses/235344-apprenez-a-programmer-en-python>)

📁 "Introduction à Python" pour le Master Ingénierie Multilingue de l'Inalco, Loïc Grobol et Yoann Dupont (fr) : <https://loicgrobol.github.io/python-im/m2-2018/> (<https://loicgrobol.github.io/python-im/m2-2018/>)

Récapitulatif

- variables (nom, affectation de valeur, type)
- fonctions built-ins et mots réservés
- opérations arithmétiques et retype

Comparer des valeurs

Opérateurs de comparaison

On utilise des opérateurs pour comparer des valeurs ou des variables entre elles. Ces comparaisons renvoient un booléen : False ou True.

On peut comparer si deux valeurs sont identiques ou égales (`==`) si au contraire elles sont différentes (`!=`), ou encore si l'une est supérieure ou inférieure à l'autre (`>` , `<` , `>=` , `<=`).

Opérateur d'égalité (*is the same as*) : `==`

```
In [23]: "a" == "a"
```

```
Out[23]: True
```

```
In [24]: 42 == 42.0
```

```
Out[24]: True
```

```
In [25]: "42" == 42
```

```
Out[25]: False
```

```
In [27]: True == False
```

```
Out[27]: False
```

Opérateur de non-égalité (*is different from*) : `!=`

```
In [28]: "a" != "a"
```

```
Out[28]: False
```

```
In [29]: "alix" != "chagué"
```

```
Out[29]: True
```

```
In [30]: 42.1 != 42
```

```
Out[30]: True
```

Plus grand que et plus petit que : `>` et `<`

```
In [31]: 2020 < 2019
```

```
Out[31]: False
```

```
In [32]: 2020 > 2019
```

```
Out[32]: True
```

```
In [33]: "ab" > "ac"
```

```
Out[33]: False
```

Quand on compare l'ordre de grandeur de deux chaînes de caractères, on compare en fait leur classement alphabétique.

```
In [36]: "alix" < "chagué"
```

```
Out[36]: True
```

Plus grand ou égal et plus petit que ou égal : \geq et \leq

```
In [38]: 3 >= 3
```

```
Out[38]: True
```

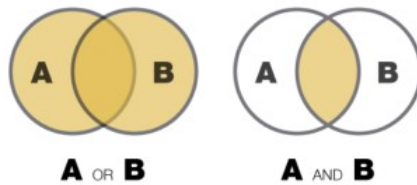
```
In [39]: 1821 <= 2020
```

```
Out[39]: True
```

Opérateurs booléens

Les opérateurs booléens permettent de compléter les opérations de comparaison.

On peut vérifier qu'une condition n'est pas vraie (`not`), qu'une condition parmi plusieurs est vraie (`or`) ou encore que plusieurs conditions sont vraies (`and`). `or` et `and` permettent donc de faire plusieurs comparaisons simultanément.



Inverser le résultat de la comparaison : `not`

```
In [40]: "a" != "a"
```

```
Out[40]: False
```

```
In [41]: not "a" != "a"
```

```
Out[41]: True
```

```
In [43]: 42 == 42.0
```

```
Out[43]: True
```

```
In [44]: not 42 == 42.0
```

```
Out[44]: False
```

Toutes les conditions testées sont vraie : `and`

```
In [49]: True and False
```

```
Out[49]: False
```

```
In [50]: True and True
```

```
Out[50]: True
```

```
In [51]: False and False
```

```
Out[51]: False
```

```
In [53]: "a" == "a" and 42 <= 43.0
```

```
Out[53]: True
```

Au moins l'une des conditions est vraie : `or`

```
In [54]: True or False
```

```
Out[54]: True
```

```
In [55]: True or True
```

```
Out[55]: True
```

```
In [57]: False or False
```

```
Out[57]: False
```

```
In [63]: not "ceci" == "cela" or 42 < 43.0
```

```
Out[63]: True
```

Exercices pratiques

Réaliser les 6 exercices du fichier `expressions.py`

👉 [https://repl.it/@AlixChagu/ENCintroalgo#basics/expressions.py_\(https://repl.it/@AlixChagu/ENCintroalgo#basics/expressions.py\)](https://repl.it/@AlixChagu/ENCintroalgo#basics/expressions.py_(https://repl.it/@AlixChagu/ENCintroalgo#basics/expressions.py))

Conditions (si... alors)

En résumé, pour formuler un test (dont le résultat sera soit `True`, soit `False`) on peut utiliser une combinaison de variables et/ou de valeurs associées aux opérateurs suivants:

- `==` (égal à)
- `!=` (différent de)
- `>` (plus grand que)
- `<` (plus petit que)
- `>=` (plus grand ou égal à)
- `<=` (plus petit ou égal à)
- `or` (ou)
- `and` (et)
- `not` (négation)
- `in` (présent dans un ensemble)

Si l'on élabore des expressions qui peuvent être vraies ou fausses, c'est parce qu'en programmation on peut définir une série d'actions à effectuer si et seulement une condition est vraie. C'est ce qu'on appelle les `structures conditionnelles`.



Les structures conditionnelles peuvent être très simple (un seul scénario) ou prévoir plusieurs scénarios associés à plusieurs cas de figures testés.

```
Si TEST est vrai, alors: INSTRUCTION
```

```
Si TEST est vrai, alors: INSTRUCTION1  
Sinon: INSTRUCTION2
```

```
Si TEST1 est vrai, alors: INSTRUCTION1  
Si TEST2 est vrai, alors: INSTRUCTION2  
Sinon: INSTRUCTION3
```

IF, ELIF, ELSE

En Python, on utilise les mots-clefs `if`, `elif` et `else` pour formuler une structure conditionnelle.

```
if test is True:  
    # Instruction...
```

```
if test is True:  
    # Instruction 1  
else:  
    # Instruction 2
```

```
if test1 is True:  
    # Instruction 1  
elif test2 is True:  
    # Instruction 2  
else:  
    # Instruction 3
```

{mot-clef} {expression} :

```
In [15]: if True:  
         print('formulation correcte!')
```

```
In [16]: if True  
         print('formulation incorrecte!')
```

File "<ipython-input-16-8cf0a3fb8455>", line 1
if True
 ^
SyntaxError: invalid syntax

```
In [20]: if not(3.14 > (2048 % 430) / 2 and type(True) != type(not('hello' == 'world'))):  
         print('formulation correcte!')
```

formulation correcte!

```
In [23]: if len("2048") = 2048/512:  
         print('formulation incorrecte!')
```

File "<ipython-input-23-41999e5b1a5a>", line 1
if len("2048") = 2048/512:
 ^
SyntaxError: invalid syntax

```
In [24]: if True or:  
         print('formulation incorrecte!')
```

File "<ipython-input-24-3b5ebb2a74b1>", line 1
if True or:
 ^
SyntaxError: invalid syntax

Indentations

Notez qu'en Python, les structures conditionnelles sont `indentées` de manière à rendre compte des blocs logiques.

Une indentation c'est soit une `tabulation`, soit **quatre espaces**.

On peut enchaîner plusieurs niveaux d'indentation (et plusieurs conditions).

```
if a is True:
    if b is True:
        if c is False:
            ...
        else:
            ...
    else:
        ...
```

En Python, l'indentation n'est pas seulement esthétique ni seulement pour faciliter la lecture du code, elle fait partie de la syntaxe de ce langage. Une erreur de tabulation provoque le plantage d'un programme.

```
In [9]: if type("hello") != type(1):
1 + 1

File "<ipython-input-9-7f0051953f50>", line 2
  1 + 1
    ^
IndentationError: expected an indented block
```

```
In [13]: if True or False:
        if len("TNAH") == 4:
            elif len("ENC") < 3:







File "<ipython-input-13-e0319fc2e368>", line 3
    elif len("ENC") < 3:
        ^
SyntaxError: invalid syntax
```

Exercices pratiques

Réaliser les 10 exercices du fichier `conditions.py`

 <https://repl.it/@AlixChagu/ENCintroalgo#basics/conditions.py> (<https://repl.it/@AlixChagu/ENCintroalgo#basics/conditions.py>)

Répétitions (tant que/ répéter x fois)

-  Les `boucles` sont un moyen de répéter une série d'actions identiques sans avoir à les écrire plusieurs fois.
-  Il y a toujours une condition qui détermine à quel moment une boucle doit être interrompue.
-  Il y a deux catégories de boucles : `for` et `while`.
-  Une boucle `for` prédéfinit le nombre d'itérations à effectuer : "pendant x itération(s), faire l'action y"
-  Une boucle `while` ne définit pas systématiquement un nombre maximum d'itérations : "tant que la condition a n'est pas remplie, faire l'action b"
-  Les instructions à effectuer dans une boucle sont toujours indentées.

Boucle *while*

```
while {expression}:
    # keep going
```

Une boucle ***while*** est interrompue dès que la condition exprimée est fausse.

Une boucle ***while*** dont la condition exprimée est toujours vraie tourne à l'infini. Il faut donc toujours s'assurer qu'une boucle *while* peut-être interrompue par le programme (ex: paramétrer un nombre maximum d'itérations).

```
In [96]: limite = 10
         nb_d_iteration = 1

         while 1 == 1 and nb_d_iteration < limite:
             1 + 1
             nb_d_iteration += 1
             print(nb_d_iteration)
```

10

Exemple d'utilisation d'une boucle *while*

On souhaite que l'utilisatrice réponde soit "y", soit "n" à la question. Pour s'assurer que la réponse donnée correspond à nos attentes, on utilise une boucle *while* qui teste la valeur de la variable *reponse* et redemande à l'utilisatrice d'entrer une valeur si celle-ci n'est pas conforme.

```
reponse = ""
while reponse != "y" and reponse != "n":
    reponse = input("voulez-vous continuer ? [y/n] ")
```

notez : la valeur de la variable *reponse*, pivot de l'expression utilisée pour interrompre la boucle *while*, est **modifiée à l'intérieur de la boucle**.

Boucle *for*

```
for var_instance in {serie}:
    # do something
```

Une boucle *for* prend une série de valeurs (ou *iterable*) et pour chaque valeur contenue dans cette série, effectue les instructions données. A chaque itération, la valeur est assignée à une variable définie après le mot-clé *for* . Si cette variable n'existe pas avant la boucle *for*, elle est automatiquement créée.

On peut lire l'instruction `for x in "hello":` ainsi : pour chaque valeur (lettre) contenue dans la chaîne de caractères "hello", l'assigner à la variable *x*.

Une boucle *for* a donc un nombre limité d'itérations, défini par la longueur de la série. En ce sens, les boucles *for* sont plus sécurisées que les boucles *while* et sont donc à privilégier.

```
In [137]: for lettre in "WASD":
           print(lettre)
```

W
A
S
D

On peut imbriquer des blocs de type boucle et des structures conditionnelles.

`range(a,b)` est une fonction qui permet de créer une suite de chiffres allant de *a* jusqu'à *b* (exclu)

```
In [101]: for chiffre in range(1,10):
           if chiffre % 2 == 0:
               print(chiffre, "est un nombre pair.")
           else:
               print(chiffre, "n'est pas un nombre pair.")
```

1 n'est pas un nombre pair.
2 est un nombre pair.
3 n'est pas un nombre pair.
4 est un nombre pair.
5 n'est pas un nombre pair.
6 est un nombre pair.
7 n'est pas un nombre pair.
8 est un nombre pair.
9 n'est pas un nombre pair.

Exercices logiques

Exercice 1

En utilisant une boucle `for`, écrivez un algorithme permettant d'obtenir l'affichage suivant :

```
3 x 0 = 0
3 x 1 = 3
3 x 2 = 6
3 x 3 = 9
3 x 4 = 12
3 x 5 = 15
```

Exercice 2

Ecrivez un script Python appliquant l'algorithme de l'exercice 1.

Exercice 3

En utilisant une boucle `while`, écrire un algorithme permettant de dessiner une pyramide de `*`, dont le nombre d'étage souhaité est stocké dans une variable.

Par exemple, si `etages = 4` alors le programme affiche la pyramide suivante :

```
*
**
***
****
```

Exercice 4

Ecrivez un script Python appliquant l'algorithme de l'exercice 3.

Nouveau type : Listes

- 📖 Les listes sont un type de variable pouvant contenir une ou plusieurs valeurs à la suite.
- 📖👉 En Python, une liste peut contenir plusieurs types de variables en même temps.
- 📖😬 En fonction des langages de programmation, on les appelle aussi `array` ou `vecteur`.
- 📖 Une liste est un objet appartenant à la catégorie des `iterables` (comme les chaînes de caractères)
- 📖 Les valeurs contenues dans une liste sont `indexées`. On peut donc cibler précisément telle ou telle valeur contenue dans la liste.
- 📖😬 L'indexation dans une liste commence à `0`

Syntaxe

```
In [35]: # créer une liste
ma_liste_vide = []
mon_autre_liste_vide = list()
ma_liste = ['A', 'B', 'C', 'D']

# Accéder au contenu d'une liste
print('ma_liste[2] :', ma_liste[2])
une_variable = ma_liste[1]
print('une_variable :', une_variable)

# Une liste peut contenir plusieurs types... y compris des listes
super_liste = ["ceci n'est pas un chiffre", 3.14, True, type('hello'), 2048, ma_liste]
print('super_liste :', super_liste)

ma_liste[2] : C
une_variable : B
super_liste : ["ceci n'est pas un chiffre", 3.14, True, <class 'str'>, 2048, ['A', 'B', 'C', 'D']]
```


Indexation

Moyen mnémotechnique : l'indexation des listes fonctionne comme les étages en France.

index	0	1	2	3
valeur	A	B	C	D
palier	RDC	1E	2E	3E

Si vous demandez un index qui n'est pas associé à une valeur (parce que la liste est trop courte), vous optez une erreur (`IndexError`).

```
In [38]: print(ma_liste[4])

-----
IndexError                                Traceback (most recent call last)
<ipython-input-38-b214c2428d1f> in <module>
----> 1 print(ma_liste[4])

IndexError: list index out of range
```

Indexation par la fin

On peut viser un emplacement dans la liste en partant du début ou de la fin.

```
In [18]: liste_exemple = ["RDC", "1er", "2e", "3e", "4e"]
print(liste_exemple[-1])
print(liste_exemple[-3])

4e
2e
```

```
In [19]: print(liste_exemple[-10])

-----
IndexError                                Traceback (most recent call last)
<ipython-input-19-c4b99e6a1b85> in <module>
----> 1 print(liste_exemple[-10])

IndexError: list index out of range
```

Bonnes pratiques

Pour éviter les déconvenues, pensez à utiliser `len()` sur une liste avant de cibler les valeurs indexées.

```
In [20]: print(len(liste_exemple))

5
```

```
In [21]: print(liste_exemple[len(liste_exemple) - 1])

4e
```

```
In [22]: index = 6
if index < len(liste_exemple):
    print(liste_exemple[index])
else:
    print("No can't do!")

No can't do!
```

Sections, *sublists* ou encore *slices*

Si on peut cibler une seule valeur à la fois, on peut aussi cibler une section dans une liste : `liste[début:fin:pas]`

```
In [15]: exemple = ['p','y','t','h','o','n', ' ', '!']
print(exemple[2:6])

['t', 'h', 'o', 'n']
```

0	1	2	3	4	5	6	7
"P"	"Y"	"T"	"H"	"O"	"N"	" "	"!"
-	-	<i>start</i>	-	-	-	<i>stop</i>	-

```
In [16]: # on peut utiliser des index négatifs
print('1 :', exemple[-6:-2])
# en parcourant la liste de gauche à droite
print('2 :', exemple[-2:-6])
# le pas de 1 est implicite, on peut le modifier
print('3 :', exemple[2:6:2])
# sans index de fin, on continue jusqu'à la fin de la liste
print('4 :', exemple[2:])
# sans index de début, on part du début de la liste
print('5 :', exemple[:5])
# on peut faire une section allant du début à la fin (copie)
print('6 :', exemple[:])
# on peut ne préciser que le pas
print('7 :', exemple[::2])
# n'importe lequel des éléments peut être implicite
print('8 :', exemple[:5:2])

1 : ['t', 'h', 'o', 'n']
2 : []
3 : ['t', 'o']
4 : ['t', 'h', 'o', 'n', ' ', '!']
5 : ['p', 'y', 't', 'h', 'o']
6 : ['p', 'y', 't', 'h', 'o', 'n', ' ', '!']
7 : ['p', 't', 'o', ' ']
8 : ['p', 't', 'o']
```

Itérables

Une chaîne de caractères fait aussi partie de la catégorie des itérables, on peut donc se servir des *slices* et des *index* sur ce type de valeur.

```
In [1]: chaine = 'python !'
print(chaine[len(chaine) - (len(chaine)*2)])
print(chaine[2:6])

p
thon
```

En revanche, dans une chaîne de caractères, on ne peut pas réassigner une valeur à un index. On dit qu'une chaîne de caractères est *immutable*.

```
In [2]: chaine[4] = '*'

-----
TypeError                                 Traceback (most recent call last)
<ipython-input-2-b6a7f31ceb5f> in <module>
----> 1 chaine[4] = '*'

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

```
In [3]: # mais on peut toujours contourner ce problème...
modif = chaine[:3] + '*' + chaine[4:]
print(modif)

pyt*on !
```

Méthodes des listes

Les listes ont des fonctions et méthodes qui leur sont associées.

- `liste.append(x)` : ajouter un élément à la fin de la liste
- `liste.pop({index})` : supprimer un élément de la liste à partir de son index (le récupérer en mémoire)
- `liste.remove(x)` : supprimer de la liste la première occurrence de l'élément recherché
- `liste.index(x)` : renvoyer l'index de la première occurrence de l'élément recherché
- `liste.count(x)` : compter le nombre d'occurrence de l'élément recherché dans la liste
- `liste.sort()` : trier la liste selon un ensemble de [règles \(https://docs.python.org/3.6/howto/sorting.html#sortinghowto\)](https://docs.python.org/3.6/howto/sorting.html#sortinghowto)

```
In [9]: liste_1a4 = [1, 2, 3, 4]
liste_5a8 = [5, 6, 7, 8]
liste_lettres = ['Q', 'Z', 'S', 'D']

# On peut concaténer 2 listes
liste_1a8 = liste_1a4 + liste_5a8
print(liste_1a8)

# Attention .append() n'a pas le même effet
liste_1a8.append(liste_lettres)
print(liste_1a8)

[1, 2, 3, 4, 5, 6, 7, 8]
[1, 2, 3, 4, 5, 6, 7, 8, ['Q', 'Z', 'S', 'D']]
```

Exercices pratiques

Réaliser les 5 exercices du fichier `iterations.py`

👉 <https://repl.it/@AlixChagu/ENCintroalgo#basics/iterations.py> (<https://repl.it/@AlixChagu/ENCintroalgo#basics/iterations.py>)

Nouveau type : Dictionnaires

- 🔑 Les dictionnaires sont un type de variables contenant une série de `valeurs` associées à des `clefs` d'accès.
- 🔑 Contrairement aux listes, les dictionnaires n'ont pas d'index (ce ne sont pas des itérables).
- 🔑 Comme les listes, les dictionnaires peuvent contenir plusieurs types de données en même temps, y compris d'autres dictionnaires.
- 🔑 Un dictionnaire peut contenir plusieurs fois la même valeur mais chaque clef doit être unique.

Syntaxe

```
In [114]: # créer un dictionnaire
mon_dico_vide = {}
mon_autre_dico_vide = dict()
mon_dico = {"clef" : "valeur"}

print("mon_dico =", mon_dico)

# Ajouter ou modifier des valeurs dans un dictionnaire
print("mon_dico_vide =", mon_dico_vide)
mon_dico_vide["hello"] = "world"
print("mon_dico_vide =", mon_dico_vide)
mon_dico_vide["hello"] = "world!"
print("mon_dico_vide =", mon_dico_vide)

# Supprimer une valeur dans un dictionnaire
del mon_dico_vide["hello"]
print("mon_dico_vide =", mon_dico_vide)

mon_dico = {'clef': 'valeur'}
mon_dico_vide = {}
mon_dico_vide = {'hello': 'world'}
mon_dico_vide = {'hello': 'world!'}
```

```
In [113]: # Accéder au contenu d'un dictionnaire
print('mon_dico["clef"] =', mon_dico["clef"])

# Un dictionnaire peut contenir plusieurs types... y compris des dictionnaires
super_dico = {"chiffre": 42, 113: "entier", "clef3" : [1,2,3]}
print('super_dico :', super_dico)

mon_dico["clef"] = valeur
super_dico : {'chiffre': 42, 113: 'entier', 'clef3': [1, 2, 3]}
```

Dans certains cas, utiliser un dictionnaire plutôt qu'une liste permet de naviguer plus facilement entre les données qu'il contient.

```
identité = ["Berthe", "Morisot", 1865, 1841, 1895]
identité = {"prénom": "Berthe", "nom": "Morisot", "debut": 1865, "naissance": 1841, "mort": 1895}
```

Méthodes des dictionnaires

- `dico.keys()` : créer un objet itérable contenant la liste des clefs utilisées dans le dictionnaire
- `dico.values()` : créer un objet itérable contenant la liste des valeurs contenues dans le dictionnaire
- `dico.get()` : envoie la valeur associée à la clef recherchée ou une valeur par défaut si la clef n'existe pas
- `dico.items()` : renvoie le contenu du dictionnaire sous la forme d'une liste de `tuples`.

```
In [3]: dico_demo = {"prénom": "Berthe", "nom": "Morisot", "debut": 1865, "naissance": 1841, "mort": 1895}
```

```
In [4]: # .keys()
print(dico_demo.keys())
print(type(dico_demo.keys()))
for key in dico_demo.keys():
    print(key)

dict_keys(['prénom', 'nom', 'debut', 'naissance', 'mort'])
<class 'dict_keys'>
prénom
nom
debut
naissance
mort
```

```
In [5]: # .values()
print(dico_demo.values())
print(type(dico_demo.values()))
for value in dico_demo.values():
    print(value)

dict_values(['Berthe', 'Morisot', 1865, 1841, 1895])
<class 'dict_values'>
Berthe
Morisot
1865
1841
1895
```

```
In [6]: # .get(key, default)
print(dico_demo.get('prénom'))
print(dico_demo.get('métier'))
print(dico_demo.get('métier', 'métier inconnu'))

Berthe
None
métier inconnu
```

```
In [7]: # .items()
print(dico_demo.items())
print(type(dico_demo.items()))
for duo in dico_demo.items():
    print(type(duo), duo, sep=" ; ")

dict_items([('prénom', 'Berthe'), ('nom', 'Morisot'), ('debut', 1865), ('naissance', 1841), ('mort', 1895)])
<class 'dict_items'>
<class 'tuple'> ; ('prénom', 'Berthe')
<class 'tuple'> ; ('nom', 'Morisot')
<class 'tuple'> ; ('debut', 1865)
<class 'tuple'> ; ('naissance', 1841)
<class 'tuple'> ; ('mort', 1895)
```

Aller plus loin

- Création de fonctions
- Listes en compréhension
- Programmation modulaires et import de librairies externes
- JSON et CSV
- Manipulation de fichiers
- Documenter son code
- Tests et exceptions
- Classes et programmation orientée objets

FIN

