



Programmation orientée objets [C++ & Python]

Dr. Pape Abdoulaye BARRO

La POO

- ❑ Généralités
- ❑ Données et manipulations
- ❑ Structures conditionnelles et itératives
- ❑ Autres types (conteneurs standard)
- ❑ Les fonctions
- ❑ Classes et héritages
- ❑ Les modules
- ❑ Les entrées/sorties
- ❑ La gestion des exceptions

Généralités

Introduction générales

- Python est un langage de programmation haut niveau développé par Guido Van Rossum (depuis 1989) et de nombreux contributeurs bénévoles. Il fait à la fois de l'impératif et de l'orienté objet et est portable, dynamique, extensible et gratuit.
- Sa syntaxe très simple, combinée à des types de données évolués (listes, dictionnaires...), conduit à des programmes à la fois très compacts et très lisibles.
- Comparé au C/C++ (valable aussi pour le Java), un programme Python est largement plus court, un temps d'exécution relativement plus rapide et une grande facilité de maintenance.
- Son extensibilité lui confère, la facilité de l'interfacer avec des bibliothèques de C. Il peut aussi servir comme langage d'extension pour des systèmes logiciels complexes.

Généralités

Introduction générales

- Ci bien qu'il fait de l'impératif, Python est orienté-objet. Il supporte donc l'héritage multiple et autres.
- Python, comme le C++, supporte également la gestion des exceptions.
- Son interpréteur principal est écrit en C.
- Python est en évolution continue et soutenu par une communauté d'utilisateur passionnés et responsables, la plupart supportant le concept d'« open source ».
- Sa version 1.0 est arrivé en 1994, puis les versions 2.0 en 2000 et celle de la 3.0 en 2008.
- Sa version 3.0 a carrément cassé la rétrocompatibilité avec les versions précédentes pour corriger des erreurs de conception. Il a fallu une dizaine d'années pour achever la transition de Python 2 vers Python 3.
- Ce cours est destiné à un apprentissage de Python 3. Une version au moins supérieure à 3.8 sera donc recommandée.

Généralités

Installation

- Pour exécuter du code Python sur votre ordinateur, il va falloir installer le logiciel (Python) qui lui serve d'interprète. À l'installation, choisissez toujours la dernière version.
- Son implémentation officielle est le CPython qui est basée sur le C (que nous utilisons), mais nous avons aussi d'autres comme le Jython qui est basé sur Java et le pypy qui est carrément basé sur Python.
 - Pour installer Python, il faut se rendre au site officiel: <https://www.python.org/downloads/> et télécharger la bonne version. À l'installation, il vous proposera un IDLE (un environnement de développement) pour pouvoir exécuter un code sous Python.
 - Si vous souhaitez l'installer sur une distribution Linux (Ubuntu, Debian, ...), à l'aide d'un invité de commande, vous pouvez faire:
 - `sudo apt-get update`
 - `Sudo apt-get install python3`
 - Pour savoir s'il est déjà installé, vous pouvez entrer la commande `python -V` ou `python3 -V` dans un terminal et vous assurer de voir apparaître ce message:
Python 3.x.y

Généralités

Prise en main

- En exécutant l'IDLE installer par défaut, vous avez la possibilité de dialoguer avec un interpréteur Python. Il nous sera utile pour tester nos premiers codes.
- Il est aussi possible d'exécuter Python depuis un terminal.
- Pour écrire du bon code, il va falloir utiliser un éditeur de texte. Les éditeurs en Python sont nombreux.
- Nous avons:
 - **Geany** qui est un logiciel libre, entièrement gratuit et sans publicité.
 - **PyCharm** qui est un logiciel édité par JetBrains, entreprise spécialisée dans les IDE (environnements de développement). Il est disponible en version gratuite mais limitée. Il intègre un débogueur pour faciliter la traque des erreurs dans un programme.
 - Et autres ...

Généralités

Prise en main

- Il est possible de manière interactive de dialoguer avec Python via Terminal. En lançant l'IDLE, le prompt (`>>`) vous invitera à taper quelque chose. Vous pourrez donc vous amuser avec de petit calcul pour constater.
 - L'interpréteur interactif est très pratique pour tester un bout de code rapidement ou explorer une valeur : on peut facilement tester si une ligne de code est correcte ou pas;
 - Il permet généralement de réaliser des testes étape par étape.
- Il est également possible d'exécuter un fichier Python (**fichier.py**) depuis le terminal. Pour cela, il suffit de lancer un terminal dans le répertoire où se trouve votre fichier de code (ou de se rendre dans le bon répertoire) puis de lancer la commande: `python fichier.py` (ou `python3 fichier.py`).
 - Écrire du code dans un fichier est toujours bénéfique. On peut le sauvegarder et y revenir plus tard si on le souhaite.
 - Un programme doit forcément s'écrire dans un fichier pour pouvoir être partageable.

La POO

- ❑ Généralités
- ❑ **Données et manipulations**
- ❑ Structures conditionnelles et itératives
- ❑ Autres types (conteneurs standard)
- ❑ Les fonctions
- ❑ Classes et héritages
- ❑ **Les modules**
- ❑ **Les entrées/sorties**
- ❑ **La gestion des exceptions**

Données, types et quelques instructions

Données et manipulations

Python est simple, concise et efficace. Quelques lignes de code est nécessaire pour produire beaucoup de choses, contrairement aux autres langages. Nous parcourons dans cette section, les données, les types et quelques instructions typiques.

- Utilisation des variables:
 - Une variable est une zone mémoire dans laquelle une valeur est stockée. **En Python**, la déclaration d'une variable et son initialisation se font **en même temps**. Par exemple, `a=2`.
 - Il faut cependant respecter les règles usuelles suivantes:
 - Le nom doit commencer par une lettre ou par un underscore ;
 - Le nom d'une variable ne doit contenir que des caractères alphanumériques courants;
 - On ne peut pas utiliser certains mots réservés.

Données, types et quelques instructions

Données et manipulations

- Les types:
 - Un type caractérise le contenu d'une variable. Il peut s'agir d'un chiffre, d'un caractère, d'un texte, d'une valeur de type vrai ou faux, d'un tableau de valeurs, etc.
 - Python est un langage à typage dynamique.
 - Ce qui veut dire qu'il n'est pas nécessaire de déclarer les variables avant de pouvoir leur affecter une valeur.
 - Le type de données peut aussi changer au cours de l'exécution du programme.
 - La fonction `type()` permet de connaître le type de la valeur d'une variable. Par exemple, si `a=10`, alors `type(a)` donnera `int`.
 - Les types sont: `int` pour les nombres entiers, `float` pour les nombres à virgules flottante, `str` pour les chaînes de caractères, `bool` pour les booléens, `list` pour une collection d'éléments séparés par des virgules, `complex` pour les nombres complexes, etc. .
 - À partir des types de base, il est possible d'en élaborer de nouveaux comme :
 - Le `tuple` qui est une collection ordonnée de plusieurs éléments. Par exemple `a=(3, 7, 10)`;
 - Le `dictionnaire` qui est un rassemblement d'éléments identifiables par une clé. Par exemple `d={"x": 4, "y": 2}`;

Données, types et quelques instructions

Données et manipulations

- Quelques instructions typique:
 - L'instruction **print** comme `printf` en C ou `cout` en C++, permet d'évaluer une expression et d'afficher le résultat.
 - Exemple:
 - `print "toc toc!"`
 - `print 9`
 - `print result`
 - `print ("toc toc!") # Python 3`
 - `print ("Bonjour", "Papa") # Python 3`
 - À l'inverse de `print`, il existe aussi une fonction **input** pour lire une entrée (saisie par l'utilisateur) depuis le terminal. Il peut optionnellement prendre en argument un message à affiché indiquant à l'utilisateur quoi faire.
 - `value=input('donnez la taille')`
 - `print('la taille est', value)`
 - Un **commentaire**, en Python, est précédé par le caractère dièse (`#`) comme suit:
 - `# ceci est un commentaire`
 - les lignes précédées par dièse sont ignorées par l'interpréteur syntaxique et marque la fin d'une ligne logique.
 - `print("Bonjour") # commentaire à la fin d'une ligne logique`

Données, types et quelques instructions

Données et manipulations

- Manipulation des chaînes de caractères:
 - Comme dans la plupart des langages de programmation, une chaîne de caractères est définie à l'aide d'une paire de guillemets (double-quotes), entre lesquels on place le texte voulu. Il est également possible d'utiliser des apostrophes à la place.
 - `msg="toc toc!"`
 - Il est possible d'entourer la chaîne d'apostrophes pour lui faire contenir des guillemets, et vis versa:
 - `'je suis "Etudiant' '`
 - `"je suis 'Etudiant' "`
 - L'exemple suivant engendra des **erreurs** tout simplement car Python pensera arrivé à la fin de la chaîne en rencontrant le deuxième guillemet et vis versa:
 - `"je suis "Etudiant" "`
 - `'je suis 'Etudiant' '`
 - Pour pouvoir les aligner comme cela, il va falloir utiliser un **backslash** (ou **antislash**, `\`) afin que le caractère ne soit pas interprété par Python. Il est nommé séquences d'échappement dans le jargon.
 - `"je suis \"Etudiant\" "`
 - `'je suis \'Etudiant\' '`

Données, types et quelques instructions

Données et manipulations

- D'autres séquences d'échappement sont disponibles, comme `\t` pour représenter une tabulation ou `\n` pour un saut de ligne (*exactement comme dans C et autres*);
- Le **backslash** étant lui-même un caractère spécial, il est nécessaire de l'échapper (donc le doubler) si on veut l'insérer dans une chaîne.
- Afin d'éviter l'utilisation excessif du **backslash** dans une chaîne, il est possible de tripler les quotes comme suit:
 - `print(""" je suis "Etudiant" """)`
 - `print('‘ je suis ‘Etudiant’ ’')`
- Si on souhaite connaître la taille de la chaîne, on fait appel à la fonction **len()**:
 - `len('Bonjour')` # donne 7
- On pourra utiliser l'opérateur (+) pour concaténer des chaînes:
 - `'Bonjour' + ' ' + 'Papa' + '!'` #espace est matérialisé par ' '
- La concaténation peut aussi être matérialisée par une expression répétitive grâce à l'opérateur (*) comme suit:
 - `'Bonjour' * 2` # donne Bonjour Bonjour
- Une chaîne est en effet un tableau de caractères. Il est donc possible d'accéder à n'importe quel caractère de la chaîne à partir de son index, s'il est compris dans les bornes (de 0 à **len(msg)-1**).
 - `msg='Bonjour'`
 - `msg[0]` # donne 'B'

Données, types et quelques instructions

Données et manipulations

- La fonction **input** renvoie toujours une **chaîne de caractères**. Il va falloir convertir les données en de type souhaité. En Python, Les nombres entiers correspondent au type **int** (pour integer, entier), les nombres à virgule au type **float** (flottant) et les chaînes de caractère au type **str** (pour string, chaîne). Ces types sont aussi utilisé comme une fonction pour la conversion des données.
 - `int('4')` # donne 2
 - `float(4)` # donne 4.0
 - `str(4)` # donne '4'
- On pourra, ainsi, convertir une entrée de l'utilisateur en **int** par exemple comme suit:
 - `value=int(input('donnez la taille'))`
- En Python, toutes les valeurs sont des objets c'est à dire, les entiers, les flottants ou les chaînes de caractères sont des objets. Ce qui veut dire qu'il existe des méthodes qui peuvent être appelées par ces types. Par exemple pour les types **str**, il existe une méthode nommée **strip** permettant de renvoyer la chaîne en retirant les espaces présents au début et à la fin.
 - `'Bonjour'.strip()` #donne 'Bonjour'
 - `input().strip()`
- En Python, on peut en voire d'autres comme **capitalize** pour mettre en majuscule le premier caractère de la chaîne, **title** pour mettre en majuscule le premier caractère de chaque mot de la chaîne, **upper** / **lower** pour mettre la chaîne en majuscules/minuscules et autres ...

La POO

- ❑ Généralités
- ❑ Données et manipulations
- ❑ **Structures conditionnelles et itératives**
- ❑ Autres types (conteneurs standard)
- ❑ Les fonctions
- ❑ Classes et héritages
- ❑ **Les modules**
- ❑ **Les entrées/sorties**
- ❑ **La gestion des exceptions**

Structures conditionnelles et itératives

Structures conditionnelles

Python, comme tous les autres langages de programmation, utilise les conditions pour changer le comportement d'un programme suivant unes certaines logiques. Ici, les opérateurs de comparaison sont largement utilisés.

- Une condition en Python correspond à un bloc **if**. il s'agit de plusieurs lignes de code réunies au sein d'une même entité logique. Le contenu du bloc ne sera exécuté que si l'expression du if est évaluée à «vrai» (**True**).

```
if 1 == 1 :  
    print(' Boc à exécuter! ')  
print('Hors du bloc')
```
- En interagissant avec l'utilisateur à l'aide d'un **input**, on peut avoir un schéma du genre:

```
value=int(input('donnez une valeur'))  
if value==10 :  
    print('Bravo! ')  
print('Fin!')
```
- Python fournit pour cela le bloc **else** («sinon») qui se place directement après un bloc **if**.

```
value=int(input('donnez une valeur'))  
if value==10 :  
    print('Bravo! ')  
else:  
    print('Perdu')
```

Structures conditionnelles et itératives

Structures conditionnelles

- Au lieu d'avoir un simple «**si** / **sinon**» nous pourrions avoir «**si** / **sinon si** / **sinon**». Ce «**sinon si**» est matérialisé par le mot-clé **elif**. Reprenons l'exemple précédent

```
secret=10
value=int(input('donnez une valeur'))
if value==secret :
    print('Bravo! ')
elif value== secret - 1 :
    print('Un peu plus...')
elif value == secret + 1 :
    print('Un peu moins...')
else:
    print('Perdu')
print('Fin')
```

- Un bloc **elif** dépend du **if** et des autres **elif** qui le précèdent. Il est aussi possible d'avoir un **if** suivi de **elif** mais sans **else**.

Structures conditionnelles et itératives

Quelques expressions booléennes

- En plus de l'égalité (`==`), il existe plusieurs opérateurs de comparaison permettent d'obtenir des booléens (`True` ou `False`).
- On trouve ainsi l'opérateur de différence, `!=`, qui teste si deux valeurs sont différentes l'une de l'autre.
- Nous avons aussi les opérateurs d'inégalités `<` et `>` pour tester les relations d'ordre. `<=` et `>=` correspondant respectivement aux opérations «inférieur ou égal» et «supérieur ou égal».
- Nous avons la négation qui se note par `not`, la conjonction («ET») par `and`, et puis la disjonction («OU») par `or`.

Structures conditionnelles et itératives

Structures itératives

En Python, on peut utiliser les structures itératives pour **parcourir** une **liste**. C'est quoi alors une liste?

- Une **liste** en Python peut être vue comme une séquence de valeurs.
 - Exemple: `Liste1=[4,8,5,9,3,4]`
 - Une liste est donc délimiter par des crochets et ses éléments sont séparés par des virgules.
 - Chaque case de la liste est associée à une position: son index.
- L'exemple précédent est constitué que de nombres entiers, mais on peut aussi construire une liste composée de valeurs de types différents.
 - Exemple: `Liste2=['papa', 26, 2.5, False]`

Structures conditionnelles et itératives

Structures itératives

- Il est possible de connaître la taille d'une liste à l'aide d'un appel à la fonction **len**: `len(liste1) # 7`
- Il est possible d'accéder aux éléments de la liste à l'aide de l'opérateur d'indexation `[]` associé à une position (**0** étant sa première position): `liste1[3] # 9`
- L'égalité et la concaténation sont aussi acceptées.
- Sur les listes, il est possible de remplacer certains éléments par d'autres, grâce à l'opérateur d'indexation (`[]`) couplé à une affectation (`=`): `liste1[3]=10 # liste1=[4,8,5,10,3,4]`
- D'autres méthodes existent aussi:
 - **count** permettant de compter le nombre d'occurrences d'un élément dans une liste: `liste1.count(4) # 2`
 - **index** permettant de retourner la position d'un élément de la liste: `liste1.index(5) # 2`
 - La méthode **append** permet comme son nom l'indique, d'ajouter un nouveau élément en fin de liste: `liste1.append(9) # liste1=[4,8,5,10,3,4,9]`
 - La méthode **insert** qui permet d'insérer un élément à une position donnée, décalant ainsi (s'il y en a) les éléments à sa droite: `liste1.insert(4, 20) # liste1=[4,8,5,10,20,3,4,9]`
 - La méthode **pop** sert quant à elle, à supprimer un élément de la liste. Utilisée sans argument, elle en supprimera le dernier élément. La méthode renvoie l'élément qui vient d'être supprimé:
 - `liste1.pop() # liste1=[4,8,5,10,20,3,4]`
 - `liste1.pop(1) # liste1=[4,5,10,20,3,4]`
 - **del** permet quant à lui, de supprimer une valeur (sans la renvoyer):
 - `del liste1[2] # liste1=[4,5,20,3,4]`

Structures conditionnelles et itératives

Structures itératives

- Il est aussi possible d'obtenir une partie d'une liste. Il suffit dans ce cas, de préciser **entre les crochets**, la position de **début** et la position de **fin**, séparées par « : » .
 - Exemple: `liste1[2:5] # [20,3,4]`
 - Il est possible d'utiliser des index négatifs pour se positionner à partir de la fin de la liste.
 - on peut aussi omettre la position de début si l'on part du début de la liste ou la position de fin si l'on va jusqu'à la fin.
 - On peut ajouter un troisième paramètre pour indiquer le **pas** (par défaut c'est 1).
 - Exemple: `liste1[::2] # [4,20,4]`
- Il est possible de modifier une partie d'une liste. Dans ce cas, on pourra récupérer jusqu'à position, puis ajouter d'autres valeurs.
 - `liste1[-1:] = [22, 17, 99] # liste1= [4, 5, 20, 3, 22, 17, 99]`
 - `liste1[:2]=[1, 6] # liste1=[1, 6, 20, 3, 22, 17, 99]`
 - `liste1[2:4]=[] # liste1=[1, 6, 22, 17, 99]`
- L'opération de **slicing** est aussi applicable aux chaînes de caractères. Il permet, dans ce cas, de renvoyer la chaîne dans l'intervalle.
 - `'bonjour'[:3] # 'bon'`

Structures conditionnelles et itératives

Structures itératives

- Une liste peut contenir toutes sortes de données, même des plus complexes: une liste peut contenir d'autres listes. D'où sa **multi-dimensionnalité**.
 - `liste2 = [1, 2, [3, [4, 5], 6]]`
 - `liste2[2][1][0] # 4`
- Pour réaliser un traitement pour chacune des valeurs de la liste, il va falloir la parcourir d'élément en élément grâce à une boucle . Une boucle est utilisée pour exécuter en plusieurs fois un bloc d'instructions tant qu'une condition donnée est vérifiée. Nous avons accès à deux boucles en Python : la boucle **for** et la boucle **while**.
- Pour la boucle **For**:
 - La syntaxe est la suivante :

```
for element in liste:  
    ...
```
 - Exemple :

```
v_max=0  
for i in liste1:  
    if i>v_max:  
        v_max=i  
print("la plus grande valeur de la liste est:", v_max)
```
 - Ceci peut aussi être réaliser par la fonction **max()**. Ex: `max(liste1) # même valeur que v_max`

Structures conditionnelles et itératives

Structures itératives

- Avec la boucle **for**, on peut utiliser la fonction **range()** pour définir une plage de valeurs:
range(5) permet de générer les valeurs 0, 1, 2, 3 et 4;
range(5, 10) permet de générer les nombres 5, 6, 7, 8 et 9;
range(6, 10, 2) permet de générer les nombres entre 6 et 10 par pas de 2 (6, 8 et 10);

Exemple:

```
for n in range(1, 10):  
    print(n)    # renvoie les valeurs 1, 2, 3, ..., 9
```

- Pour les tableaux à deux dimensions, on pourra utiliser une deuxième boucle (puis les imbriquées) comme suite:

```
for i in range(n):  
    for j in range(m):  
        # traitement
```


Structures conditionnelles et itératives

Structures itératives

- Pour la boucle While:
 - Signifiant « **tant que** » et permettant de boucler tant qu'une condition n'est pas remplie. Elle est d'habitude utilisée lorsque le nombre d'itération n'est pas connu à l'avance.
 - La syntaxe est la suivante :
while condition:
...
• Exemple:
q= 'o'
while q== 'o':
 print('Vous êtes dans la boucle')
 q= input('Souhaitez-vous rester dans la boucle (o/n) ? ')
print('Vous êtes sorti de la boucle')
- L'instruction **break** permet de stopper l'exécution d'une boucle lorsqu'une certaine condition est vérifiée.
- L'instruction **continue** permet elle d'ignorer l'itération actuelle de la boucle et de passer directement à l'itération suivante.

La POO

- ❑ Généralités
- ❑ Données et manipulations
- ❑ Structures conditionnelles et itératives
- ❑ **Autres types (conteneurs standard)**
- ❑ Les fonctions
- ❑ Classes et héritages
- ❑ **Les modules**
- ❑ **Les entrées/sorties**
- ❑ **La gestion des exceptions**

Autres types de données

Les dictionnaires

- Comme les listes, les dictionnaires sont des conteneurs. Un **dictionnaire** est un ensemble formé de couples clé-valeur (clé: valeur). Ce qui veut dire que les valeurs ne sont accessibles que par clé non plus par index comme ce fut le cas des listes.
 - Exemple:

```
contacts={'Massamba' : '765261606', 'Mademba' : '774511707'}
contacts['Massamba'] # 765261606
```
 - À la récupération d'une valeur, il suffit de préciser une clé de notre dictionnaire plutôt qu'un index.
- Quelques opérations:
 - Les dictionnaire sont modifiable. Ce qui veut dire qu'on peut lire la valeur d'une clé ou la modifier (écriture et suppression).

```
contacts['Mafatou'] = '789881808'
contacts # {'Massamba' : '765261606', 'Mademba' : '774511707', 'Mafatou' : '789881808'}
del contacts['Mademba']
contacts # {'Massamba' : '765261606', 'Mafatou' : '789881808'}
```
 - On retrouve l'opérateur (**in**), fonctionnant sur les **clés** et non sur les valeurs.
 - 'Massamba' **in** contacts # True
 - On peut aussi connaître la taille d'un dictionnaire en appelant la fonction **len**
 - **len**(contacts) # 2

Autres types de données

Les dictionnaires

- Nous trouvons , pour les dictionnaires, quelques méthodes intéressantes comme:
 - `get(cle, default)` qui renvoie la valeur d'une clé. Si la clé n'existe pas, elle renvoie la valeur `default` fournie. Si aucune valeur n'est fournie, elle renvoie `None`.
`l = {1: 'a', 2: 'b', 3: 'c'}`
`l.get(1) # 'a'`
 - La méthode `pop(cle, default)` renvoie la valeur identifiée par la cle et retire l'élément du dictionnaire. Si la clé n'existe pas, pop se contente de renvoyer la valeur `default`. Si le paramètre default n'est pas fourni, une `erreur` est levée.
`l.pop(1) # 'a'`
`l # {2: 'b', 3: 'c'}`
 - La méthode `update` permet d'étendre le dictionnaire avec les données d'un autre dictionnaire. Dans ce cas, si une clé existe déjà dans le dictionnaire actuel, sa valeur est remplacée par la nouvelle qui est reçue.
`l2 = {3: 'ccc', 4: 'd'}`
`l.update(l2)`
`l # {2: 'b', 3: 'ccc', 4: 'd'}`
 - La méthode `setdefault(cle, default)` qui fonctionne comme `get()` mais si la cle n'existe pas et default est fourni, le couple (cle, default) est ajouté à la liste.
`l.setdefault(5, 'e') # 'e'`
`l # {2: 'b', 3: 'ccc', 4: 'd', 5: 'e'}`
 - La méthode `clear()` sert à vider complètement un dictionnaire.
`l.clear()`
`l # {}`

Autres types de données

Les dictionnaires

- Considérons la liste suivante:
 - `liste=[['Massamba' : '765261606'], ['Mafatou' : '789881808']]`
 - Il est donc possible de le convertir en dictionnaire en utilisant la fonction `dict()`:
 - `contacts=dict(liste)`
 - `contacts # {'Massamba' : '765261606', 'Mafatou' : '789881808'}`
- Il est possible d'itérer un dictionnaire en utilisant les boucles. Itérer sur un dictionnaire revient donc à itérer sur ces clés. Avec `for`, nous avons:
 - `for n in contacts:`
 - `print (n, ' : ', contacts[n])`
- On peut aussi utiliser la méthode `values()` renvoyant l'ensemble des valeurs du dictionnaire, sans les clés. La méthode `keys()` permet aussi de renvoyer les clés.
 - `for contact in contacts.values():`
 - `print ('numero:', contact)`
- La methode `items()` renvoie quant à elle les couples clé/valeur du dictionnaire.
 - `for nom, num in contacts.items():`
 - `print(nom, ' : ', num)`

Autres types de données

Les tuples

- Les **tuples** sont semblables aux listes mais non modifiables. À la définition, on ne peut ni ajouter, ni supprimer, ni remplacer d'élément.
- Un **tuple** est généralement défini par une paire de parenthèses contenant les éléments séparés par des virgules. Comme une liste, un tuple peut contenir des éléments de types différents.
 - (1, 2, 3)
 - ('a', 'b', 'c')
 - (1, 'b')
 - 1, 2, 3
- On peut accéder aux éléments d'un tuple (**en lecture uniquement**) avec l'opérateur d'indexation [], qui gère les index négatifs et les slices.
 - values = (4, 5, 6)
 - values[0] # 4
 - values[-1] # 6
 - values[::2] # (4, 6)
 - 5 in values # True
- On peut concaténer deux tuples avec +, et multiplier un tuple par un nombre avec *.
- Les fonctions **len**, **min**, **max**, **all**, **any**, etc., sont aussi applicables aux tuples.
- Enfin, les tuples sont pourvus de deux méthodes, **index** et **count**, pour respectivement trouver la position d'un élément et compter les occurrences d'un élément.
 - values.index(5) # 1
 - values.count(5) # 1

Autres types de données

Les tuples

- On peut parfois se demander quand utiliser un tuple et quand utiliser une liste.
 - Le tuple étant comparable à une liste non modifiable, il peut donc être utilisé pour toutes les opérations attendant une liste et ne cherchant pas à la modifier. Il est même préférable de l'utiliser si l'on sait qu'il ne sera jamais question de modification, ou si l'on veut empêcher toute modification.
 - Étant non modifiables, ils peuvent être utilisés en tant que clés de dictionnaires.
 - `matrice= {(0, 0): 1, (0, 1): 5, (1, 0): 8, (1, 1): 3}`
 - `matrice[1, 1] # 3`
- Si un tuple contient une liste, rien n'empêche d'ajouter des éléments à cette liste.
 - `agenda = ('25/03/2023', ['examens', 'C++'])`
 - `agenda[1].append('Python')`
 - `agenda # ('25/03/2023', ['examens', 'C++', 'Python'])`

À suivre

Feedback sur:

pape.abdoulaye.barro@gmail.com