Programmation Orientée Objet 2

Python: de l'impératif à l'objet

Pape Abdoulaye BARRO, Ph.D.

Enseignant-chercheur
UFR des Sciences et technologies
Département Informatique

E-LabTP, Laboratoire des TP à Distance, UFR-SET,
Marconi-Lab, Laboratoire de Télécommunications, ICTP, Italie

Email: pape.abdoulaye.barro@gmail.com

Plan

Généralités, installation et prise en main
Données et manipulations
Structures conditionnelles
Structures itératives
Autres types (conteneurs standard)
Les fonctions
Classes et héritages
Les modules
Les entrées/sorties
La gestion des exceptions

Introduction générales

Généralités, installation et prise en main

Généralités

- Python est un langage de programmation haut niveau développer 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

- 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 fallut 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.

Installation

- Pour exécuter du code python sur vote 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'autre 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 téléchargé 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'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

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 ecrire 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 facilité la traque des erreurs dans un programme.
  - Et autres …

Prise en main

Exemple avec l'IDLE

- 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 amusez 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 forcement s'écrire dan un fichier pour pouvoir être partageable.

Données et manipulations

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 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 langages 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 connaitre 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 chaines 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 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 (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 et manipulations

- Manipulation des chaines 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-quottes), 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 chaine 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 et manipulations

- Manipulation des chaines de caractères:
  - D'autres séquences d'échappement sont disponibles, comme **\t** pour représenter une tabulation ou **\n** pour un saut de ligne (exactement comme dan 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'eviter l'utilisation excessif du backslash dans une chaine, il est possible de tripler les quottes comme suit:
    - print("""je suis "Etudiant" """)
    - print(‘’‘je suis ‘Etudiant’ ‘’’)
  - Si on souhaite connaitre la taille de la chaine, on fait appel à la fonction **len**():
    - len(‘Bonjour’) # donne 7
  - On pourra utiliser l'opérateur (**+**) pour concaténer des chaines:
    - ‘Bonjour’ + ‘ ‘ + ‘Papa’ + ’!’ #espace est matérialisé par ‘ ‘
  - La concaténation peut aussi être matérialiser par une expression répétitive grâce à l 'opérateiur (*****) comme suit:
    - ‘Bonjour’ * 2 # donne Bonjour Bonjour
  - Une chaine 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 et manipulations

- Manipulation des chaines de caractères:
  - 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 poura, 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 chaine 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 chaine en majuscules/minuscules et autres ..

Structures conditionnelles

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=**in**t(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

- 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 précèdent

```
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
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**.

Plan

Généralités, installation et prise en main
Données et manipulations
Structures conditionnelles
Structures itératives
Autres types (conteneurs standard)
Les fonctions
Classes et héritages
Les modules
Les entrées/sorties
La gestion des exceptions

Structures itératives

Structures itératives

Les listes

- 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 précèdent est constitué que de nombres entiers. On peut aussi construire une liste composée de valeurs de types différents.
  - Exemple:
    - Liste2=['papa', 26, 2.5, **False**]

Structures itératives

Quelques opérations sur les listes

- Il est possible de connaitre 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 **[ ]** 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 existes 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 nouvel 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 itératives

Quelques opérations sur les listes

- 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 « : » .
  - 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).
    - 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 chaine dans l'intervalle.
  - 'bonjour'[:4] # 'bon'

Structures itératives

Listes – boucle for

- 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.
- 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 itératives

Listes – boucle for

- 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 itératives

Listes – boucle while

- 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.

Autres types de données

Dictionnaires

Tuples

Les dictionnaires

définition - quelques opérations

- 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 connaitre la taille d'un dictionnaire en appelant la fonction len
    - len(contacts) # 2

Les dictionnaires

quelques méthodes

- Nous trouvons , pour es 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.updatle(2)
    - l # {2: 'b', 3: 'ccc', 4: 'd'}
  - La méthode setdefault (cle, default) qui fonctionne comme get() mais si 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 # { }

Les dictionnaires

conversion - itération

- 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)

Les tuples

définition - quelques opérations

- 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

Les tuples

utilisations

- 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'])

Plan

Généralités, installation et prise en main
Données et manipulations
Structures conditionnelles
Structures itératives
Autres types (conteneurs standard)
Les fonctions
Classes et héritages
Les modules
Les entrées/sorties
La gestion des exceptions

Les fonctions

définition – directives
Paramètres d'une fonction

Les fonctions

définition - directives

- Les fonctions permettent d'éviter des répétitions de code. Un programme subdivisé en plusieurs petites portions de code est beaucoup plus aisé à relire et à maintenir.
  - La définition d'une fonction se fait par le biais du mot-clé **def** suivi du **nom** de la fonction. Suivent des **parenthèses** qui contiennent les éventuels paramètres de la fonction puis le caractère **:** qui délimite le début d'une séquence de code.
  - Exemple:

```
def stl(prenom):
    print('Bonjour %s' % prenom)
```

- Lorsque des variables sont définies dans le code, elles sont placées par l'interpréteur soit dans le contexte local(locals()) ou dans le contexte global(globals()).
  - Une variable est insérée dans le contexte local si elle est définie dans un bloc (boucle, fonction, …).
  - Elle est insérée dans le contexte global si elle est définie en dehors de tout bloc. Ce qui veut dire il est impossible d'affecter directement les variables du contexte global depuis un bloc.
  - Exemple:

```
prenom = 'Massamba'
def slt(prenom):
    print(locals())
    print('Bonjour %s' % prenom)
```

```
>>> slt('Mademba')
{'prenom': ' Mademba '}
Bonjour Mademba
```

```
>>> print(globals())
{'__builtins__': <module '__builtin__' (built-in)>,
'__name__':'__main__',
'slt': <function slt at 0xb7fedf0c>,
'__doc__': None, 'prenom': 'Massamba'}
```

Les fonctions

directives

- Pour pouvoir contourner cette limitation il est nécessaire d'utiliser la directive global qui permet de spécifier que la variable est dans le contexte global.
- Exemple:

```
identite = 'Soundiata KeÏta'
def slt(prenom, nom):
    global identite
    identite = '%s %s' %(prenom, nom)
    print (locals())
    print (identite)
```

```
>>> slt('Samory', 'Touré')
{'prenom': 'Samory', 'nom': 'Touré'}
Samory Touré
>>> print(identite)
Samory Touré
```

- Lorsqu'une fonction doit renvoyer un résultat explicite, la directive return est utilisée. À précisé qu'en Python, il n'y a pas de distinction entre les fonctions et les procédures, contrairement à certains langages fortement typés. Les procédures sont tout simplement des fonctions qui ne renvoient pas de résultat comme en C.
  - Exemple:

```
def carre(nombre):
    return nombre*nombre
```

```
>>> carre(2)
4
```

  - Il est possible de retourner plusieurs résultats en les séparant par des virgules. Dans ce cas, l'interpréteur renvoie ces éléments dans un tuple.
  - Exemple:

```
def trois_valeur():
    return 1, 2, 3
```

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

Les fonctions

Paramètres d'une fonction

- Paramètres d'une fonction
  - Un paramètre est une variable définie dans la fonction qui recevra une valeur lors de chaque appel. Cette valeur pourra être de tout type, suivant ce qui est fourni en argument.
  - En Python, Il existe trois types de paramètres :
    - les **paramètres explicites** sont définit par des noms et sont séparés par des virgules. Certains paramètres peuvent prendre des valeurs par défaut et devenir optionnel ;
    - Exemple:

```
def slt(prenom, nom='Diop'):
    print('Bonjour %s %s' % (prenom, nom))
```

```
>>> slt('Elhadji')
Bonjour Elhadji Diop
>>> slt('Elhadji', 'Kaly')
Bonjour Elhadji Kaly
```

      - ✓ Il est cependant nécessaire de regrouper tous les paramètres optionnels à la fin de la liste des paramètres.
    - Lorsqu'il y a plusieurs paramètres optionnels, le code appelant peut définir ou non la valeur de chacun sans avoir à respecter un ordre précis, en utilisant la notation nom=valeur pour ce paramètre. On parle alors de ***nommage des paramètres***.
    - Exemple:

```
def somme(a, b=3, c=4):
    return a + b + c
```

```
>>> somme(5)
12
>>> somme(5, 2, 8)
15
>>> somme(5, c=2)
10
>>> somme(c=5, a=2, b=6)
11
```

Les fonctions

Paramètres d'une fonction

- les **paramètres non explicites** sont laissés à l'appréciation de l'utilisateur. Il peut mettre autant de valeurs nommées qu'il le souhaite sans qu'il soit nécessaire de les définir dans la liste des arguments.
- Ces paramètres sont fournis sous la forme nom=valeur à la fonction. L'interpréteur place ces valeurs dans un dictionnaire qu'il faut au préalable définir en fin de liste par son nom précédé de deux étoiles.
- Exemple:

```
def phrase(**mots):
    print ('phrase de %d mot(s)' % len(mots))
    print ('Liste des mots: %s' % ' '.join(mots.values()))
    print ('Nom des paramètres: %s' % ' '.join(mots.keys()))
```

```
>>> phrase(mot1='devoir', mot2='annulé')
phrase de 2 mot(s)
Liste des mots:  devoir annulé
Nom des paramètres: mot1 mot2
>>> phrase()
phrase de 0 mot(s)
Liste des mots:
Nom des paramètres:
```

Les fonctions

Paramètres d'une fonction

- Les **paramètres arbitraires** sont équivalents aux paramètres non explicites sauf qu'ils ne sont pas nommés. L'interpréteur les regroupe dans un tuple nommé qu'il passe à la fonction. Le nom du tuple est fourni préfixé cette fois-ci d'une seule étoile.
- Exemple:

```
def planning(phrase, *vals):
    print (phrase % vals)
```

```
>>> planning('L\'examen est fixé pour le %d du mois de %s', 25, 'Mars')
L'examen est fixé pour le 25 du mois de Mars
```

- Lorsque des paramètres arbitraires sont combinés avec des paramètres explicites ou non explicites, la déclaration du nom du tuple qui contiendra les valeurs se place toujours après les paramètres explicites et avant les paramètres non explicites.
- def nom_fonction(a, b, c, ..., *arbitraires, **explicites)

Classes et héritages

Classes et héritages

définition de classe

- Une classe peut être vue comme un regroupement logique de fonctions et de variables permettant de définir un comportement et un état du programme. Rappelons que Python est Orienté Objet, ce qui veut dire que tous les éléments qu'il manipule sont considérés comme étant des objets.
- Une classe définit un modèle d'objet que l'on peut ensuite instancier autant de fois que nécessaire.
  - Une instance devient un objet indépendant qui contient les fonctions et les variables définies dans le modèle.
- Le mot réservé ***class*** sert à définir un modèle en associant un certain nombre de variables et de fonctions à un nom.
  - Exemple:

```
class Voiture:
    couleur = 'Rouge'
```

    - Les éléments (variables et fonctions membres) de la classe sont nommés attributs et on parle plus précisément de méthodes pour les fonctions et d'attributs de données pour les variables.
    - La classe Voiture pourra donc être utiliser pour instancier des objets en l'appelant comme une fonction.
    - Exemple:
      - v=Voiture()
      - On pourra donc accéder à couleur en faisant: v.couleur

Classes et héritages

Paramètre self

- De la même manière que pour une fonction, l'interpréteur met à jour les variables locales et globales lors de l'exécution des méthodes. Le code exécuté a donc une visibilité locale aux éléments définis dans la méthode et globale aux éléments en dehors de l'instance.
- Pour atteindre les éléments définis dans l'espace de noms de l'instance de la classe, il est donc nécessaire d'avoir un lien qui permette de s'y référer. L'interpréteur répond à ce besoin en fournissant l'objet instancié en premier paramètre de toutes les méthodes de la classe.
- Par convention, et même si ce nom n'est pas un mot-clé du langage, ce premier paramètre prend toujours le nom *self*.
  - Exemple:

```
class C:
    x = 8
    y = x + 5
    def affiche(self):
        self.z = 20
        print(C.y)
        print(self.z)
```

```
>>> obj = C()
>>> obj.affiche()
13
20
```

  - Les méthodes définies dans les classes ont donc toujours un premier paramètre fourni de manière transparente par l'interpréteur, obj.affiche() étant remplacé au moment de l'exécution par obj.affiche(obj).
  - ***self*** représente l'objet sur lequel la méthode sera appliquée.

Classes et héritages

Héritage simple - Héritage multiple

- L'héritage est la faculté d'une classe B de s'approprier les fonctionnalités d'une classe A. On dit que B hérite de A ou encore que B dérive de A. Python permet de définir des classes dérivées très simplement :

```
class B(A):
    pass
```

- Au moment de l'instanciation de la classe Mercedes, l'interpréteur mémorise le nom de la classe parente afin de l'utiliser lorsque des attributs de données ou des méthodes sont utilisés :
  - si l'attribut en question n'est pas trouvé dans la classe, l'interpréteur le recherche dans la classe parente.
  - Si l'attribut n'est pas trouvé dans la classe parente, l'interpréteur remonte l'arbre de dérivation à la recherche d'une méthode portant la même signature avant de provoquer une exception AttributeError.

```
class Voiture:
    type = 'voiture'
    def affiche(self):
        print(self.type)

class Mercedes(Voiture):
    pass
class MercedesTurbo(Mercedes):
    pass
```

```
>>> v=MercedesTurbo()
>>> v.affiche()
voiture
```

- Python supporte l'héritage multiple en laissant la possibilité de lister plusieurs classes parentes dans la définition.

```
class C(A, B):
    pass
```

Classes et héritages

Surcharge des attributs - Polymorphisme

- Toutes les méthodes et attributs de données peuvent être surchargés, en utilisant la même signature.

```
class Voiture:
    type = 'voiture'
    def affiche(self):
        print(self.type)
    def utilise(self):
        self.affiche()

class Mercedes(Voiture):
    def utilise(self):
        print('Mercedes est une %s' % self.type)
```

```
>>> v=Mercedes()
>>> v.affiche()
Voiture
>>> v.utilise()
Mercedes est une voiture
```

- Si une méthode doit spécifiquement utiliser un attribut que la règle de surcharge ne lui renvoie pas, il est possible de préciser à l'interpréteur de quelle classe il s'agit, en utilisant un préfixe de la forme : ClasseDeBase.methode(self, parametres).

```
class Mercedes(Voiture):
    def utilise(self):
        print('Mercedes est une %s' % self.type)
    def affiche(self):
        Voiture.affiche(self)
```

```
>>> v=Mercedes()
>>> v.affiche()
Voiture
```

Classes et héritages

Constructeur et destructeur

- Lorsqu'une classe est instanciée, la méthode spéciale __init__() est invoquée avec en premier paramètre l'objet nouvellement instancié par l'interpréteur. Ce fonctionnement permet de procéder à un certain nombre d'initialisations lorsque l'on crée une instance de classe.

```
class Voiture:
    def __init__(self):
        print("Nouvelle voiture n°%s" % id(self))
        self.immatriculation = 'TH %s' % id(self)
```

```
>>> v=Voiture()
Nouvelle voiture n°322569512
>>> v. immatriculation
'TH 322569512'
```

  - Il est d'usage de déclarer les attributs de données directement dans le constructeur lorsque ceux-ci ne sont pas partagés par toutes les instances: ils sont attachés à l'objet au moment de leur initialisation comme c'est le cas dans notre exemple pour immatriculation.

- Comme pour une méthode classique, le constructeur peut recevoir des paramètres supplémentaires, qui sont directement passés au moment de l'instanciation.

```
class Voiture:
    def __init__(self, type):
        self.type = type
```

```
>>> v=Voiture('Mercedes')
>>> v. type
'Mercedes'
```

- Un destructeur peut également être défini grâce à la méthode spéciale __del__() lorsque du code doit être appelé au moment de la destruction de l'instance. Le code contenu dans cette méthode doit explicitement appeler la méthode __del__() des classes parentes, si elles existent.

```
class Voiture:
    def __del__(self):
        print('destruction')
```

```
>>> v=Voiture()
>>> del v
destructeur
```

Classes et héritages

Attributs privés

- En ce qui concerne la protection des attributs, il est possible de définir des attributs privés à la classe en préfixant le nom de deux espaces soulignés. Si l'attribut se termine aussi par des espaces soulignés, ils ne doivent pas être plus de deux pour qu'il reste considéré comme privé.
- L'interpréteur repère ces attributs et modifie leurs noms dans le contexte d'exécution. Pour un attribut __a de la classe Class, le nom devient _Class__a.
- Le mapping étend alors la recherche à cette notation lorsque les appels se font depuis le code de la classe, de manière à ce que les appelants extérieurs n'aient plus d'accès à l'attribut par son nom direct.

```
class Voiture:
    __defaults=['silencieuse']
    qualites=['rapide', 'economique']
    def caracteristiques(self):
        print(self.__defaults)
        print(self.qualites)
```

```
>>> v=Voiture()
>>> v.caracteristiques()
['silencieuse']
['rapide', 'economique']
>>> v.qualites
['rapide', 'economique']
>>> v.__defaults
Traceback (most recent call last):
File "<stdin>", line 1, in ?
AttributeError: Voiture instance has no attribute '__defaut'
```

Classes et héritages

Méthodes spéciales

- Il est possible en Python de définir d'autres méthodes spéciales que __init__() et __del__(), qui déterminent un fonctionnement spécifique pour une classe lorsqu'elle est utilisée dans certaines opérations.
- Ces méthodes permettent de faire varier le comportement des objets et sont regroupées en fonction des cas d'utilisation :
- représentation et comparaison de l'objet ;
  - __str__() doit renvoyer une représentation sous forme de chaine de caractères d'un objet.

Exemple:

```
class Voiture:
    __defaults=['silencieuse']
    qualites=['rapide', 'economique']
  def  caracteristiques(self):
    print(self.__defaults)
      print(self.qualites)
  def __str__(self):
    return 'je suis un objet de type Voiture'
```

```
>>> o = Voiture()
>>> str(o)
'je suis un objet de type Voiture'
```

Classes et héritages

Méthodes spéciales

- représentation et comparaison de l'objet ;
  - __repr__() similaire à la méthode __str__() sauf qu'elle renvoie une chaîne plus riche en informations qui peut être utilisée pour recréer l'objet (v1 = eval(repr(o))).
  - Généralement, la chaîne __str__() est destinée aux utilisateurs et la chaîne __repr__() est destinée aux développeurs.
  - Pour implémenter cette méthode, il suffit de la surcharger à l'intérieur d'une classe :

Exemple:

```
class User:
    def __init(self, prenom, nom):
        self.prenom = prenom
        self.nom = nom
    def __repr__(self):
        return "Utilisateur(prenom='{}', nom='{}')".format(self.prenom, self.nom)
```

```
>>> u = User(prenom="Thomas", nom="SANKARA")
>>> print(repr(u))
Utilisateur(prenom='Thomas', nom='SANKARA')
```

Classes et héritages

Méthodes spéciales

- utilisation de l'objet comme fonction ;
  - __call__() qui permet d'écrire des classes dont les instances se comportent comme des fonctions et peuvent être appelées comme une fonction.

Exemple:

```
class Somme:
    def __init__(self):
        print("Instance Créé")

    def __call__(self, a, b):
        print(a + b)
```

```
>>> som = somme()
Instance Créé
>>> som(3, 5)
8
```

Classes et héritages

Méthodes spéciales

- accès aux attributs de l'objet ;
  - Lorsque l'interpréteur rencontre une écriture de type objet.attribut, il utilise le dictionnaire interne __dict__ pour rechercher cet attribut, et remonte dans les dictionnaires des classes dérivées si nécessaire.
  - L'utilisation des trois méthodes suivantes permet d'influer sur ce fonctionnement.
  - __setattr__() qui est utilisée lorsqu'une valeur est assignée, en lieu et place d'une modification classique de l'attribut __dict__ de l'objet.
  - objet.attribut = 'valeur' devient équivalent à objet.__setattr__('attribut','valeur')
  - Le code contenu dans __setattr__() ne doit pas appeler directement l'attribut à mettre à jour, au risque de s'appeler lui-même récursivement. Il faut utiliser un accès à __dict__.

Exemple:

```
class Personne:
    def __init__(self, prenom):
        self.prenom = prenom
```

```
>>> p = Personne('Kinta')
>>> setattr(p, 'nom', 'KINTE')
>>> print(p.__dict__)
{'prenom':'Kinta', 'nom':'KINTE'}
```

```
class Majuscule:
    def __setattr__(self, nom, valeur):
        self.__dict__[nom] = valeur.upper()
```

```
>>> m = Majuscule()
>>> m.nom = 'kinté'
>>> m.nom
'KINTE'
```

Classes et héritages

Méthodes spéciales

- __getattr__() est appelée en dernier recours lorsqu'un attribut est recherché dans un objet. Cette méthode ne surcharge pas le fonctionnement normal afin de permettre à __setattr__(), lorsqu'elle est surchargée, d'accéder aux attributs normalement.

Exemple:

```
class Personne:
    def __init__(self, prenom):
        self.prenom = prenom
    def __getattr__(self, nom):
        return 'pas de`{}` dans les attributs'.format(str(nom))
```

```
>>> p = Personne('Kinta')
>>> p.nom
'pas de`nom` dans les attributs'
>>> p.prenom
'Kinta'
```

- getattr(object, name) ou getattr(object, name, default) renvoie la valeur de l'attribut nommé de l'objet. name doit être une chaîne de caractères.
  - Si la chaîne est le nom d'un des attributs de l'objet, le résultat est la valeur de cet attribut. Par exemple, getattr(o,'nom') est équivalent à o.nom.
  - Si l'attribut nommé n'existe pas, la valeur par défaut est renvoyée si elle est fournie, sinon AttributeError est levée.

Classes et héritages

Méthodes spéciales

- __getattribute__() est appelé inconditionnellement pour mettre en œuvre les accès aux attributs pour les instances de la classe. Cette méthode doit renvoyer la valeur de l'attribut ou lever une exception AttributeError.
- Si la classe définit également __getattr__(), cette dernière ne sera pas appelée à moins que __getattribute__() ne l'appelle explicitement ou ne lève une AttributeError.
- Afin d'éviter une récursivité infinie dans cette méthode, son implémentation doit toujours appeler la méthode de la classe de base portant le même nom pour accéder aux attributs dont elle a besoin, par exemple, object.__getattribute__(self, name).

Exemple:

```
class Personne (object):
    def __getattribute__(self, name):
        print(f"obtenir `{str(name)}`")
        return  object.__getattribute__(self, name)
```

```
>>> p = Personne()
>>> p.nom = 'Kounte'
>>> p.nom
obtenir `nom` Kounte
```

Classes et héritages

Méthodes spéciales

- __delattr__() est le complément des deux méthodes précédentes, objet.__delattr__('attribut') est équivalent à del objet.attribut.

Exemple:

```
class Personne:
    def __getattr__(self, name):
        print('getattr %s' % name)
        if name in self.__dict__:
            return self.__dict__[name]
        else:
            print("attribut '%s' inexistant" % name)
    def __setattr__(self, name, valeur):
        print('set %s: %s' % (name, str(valeur)))
        self.__dict__[name] = valeur
    def __delattr__(self, name):
        print('del %s' % name)
        if name in self.__dict__:
            del self.__dict__[name]
        else:
            print("attribut '%s' inexistant" % name)
```

```
>>> p= Personne()
>>> p.age = 20
set age: 20
>>> p.first_name
getattr first_name
attribut 'first_name' inexistant
>>> p.first_name = 'Kinta'
set first_name: Kinta
>>> del p.first_name
del first_name
>>> p.first_name
getattr first_name
attribut 'first_name' inexistant
```

Classes et héritages

Méthodes spéciales

- utilisation de l'objet comme conteneur ;
  - Les mappings et les séquences sont tous des objets de type conteneurs, qui implémentent un tronc commun de méthodes. Ces méthodes sont présentées ci-dessous et peuvent être définies dans toute classe.
    - __getitem__(key) est utilisée lorsqu'une évaluation de type objet[key] est effectuée.
      - Pour les objets de type séquences, key doit être un entier positif ou un objet de type slice. Les mappings, quant à eux, utilisent des clés de tout type non modifiable.

Exemple:

```
class Personne:
    def __init__(self):
        self._data = {}
    def __getitem__(self, key):
        if key in self._data:
            return self._data[key]
        else:
            print("pas de %s" % key)
```

```
>>> p = Personne()
>>> p['nom']
Pas de nom
>>> p['penom']= 'kinta'
>>> p['penom']
kinta
```

- Si la clé fournie n'est pas d'un type compatible, une erreur TypeError est retournée.
- Si la clé est en dehors des valeurs autorisées, une erreur de type IndexError est retournée.

Classes et héritages

Méthodes spéciales

- __setitem__(key, value) utilisée lorsqu'une assignation de type objet[key] = valeur est effectuée.
- Les mêmes erreurs peuvent être utilisées que celles de __getitem__. Les mappings ajoutent automatiquement la clé lorsqu'elle n'existe pas, contrairement aux séquences qui retournent une erreur si la clé n'existe pas.
- __delitem__(key) permet de supprimer une entrée du conteneur.
- __len__() est appelée par la primitive len(), et permet de renvoyer le nombre d'éléments du conteneur.
- __iter__() est appelée par la primitive iter(), et doit renvoyer un iterator capable de parcourir les éléments.
- __contains__(item) renvoie vrai si item se trouve parmi les éléments.

Exemple:

```
class Personne:
    ....
    def __setitem__(self, key, value):
        self._data[key] = value
    def __delitem__(self, key):
        print('objet supprime')
    def __len__(self):
        return len(self._data)
    def __contains__(self, item):
        return item in self._data.values()
```

```
>>> p = Personne()
>>> p['nom']
Pas de nom
>>> p['penom']= 'kinta'
>>> p['penom']
Kinta
>>> len(p)
1
>>> del p['penom']
```

Classes et héritages

Méthodes spéciales

- utilisation de l'objet comme type numérique.

Ces méthodes peuvent être utilisées pour définir le fonctionnement de l'objet lorsqu'il est employé dans toute opération numérique, que ce soit une addition, un décalage de bits vers la gauche, ou encore une inversion.

Méthode	Opération
__add__(other)	objet + other
__sub__(other)	Objet - other
__mul__(other)	objet * other
__floordiv__(other)	objet // other
__mod__(other)	objet % other
__divmod__(other)	divmod(objet, other)
__pow__(other[, modulo])	objet ** other
__lshift__(other)	objet << other
__rshift__(other)	objet >> other
__and__(other)	objet & other
__xor__(other)	objet ^ other
__or__(other)	objet | other
__div__(other)	objet / other
__truediv__(other)	objet / other
__neg__()	- objet
__pos__()	+ objet
__abs__()	abs(objet)
__invert__()	~ objet
__complex__()	complex(objet)
__int__()	int(objet)
__long__()	long(objet)
__float__()	float(objet)
__oct__()	oct(objet)
__hex__()	hex(objet)
__coerce__(other)	coerce(objet, other)

- Pour toutes ces méthodes, un appel à objet opérateur other déclenche un appel à objet.methode(other).

Classes et héritages

Méthodes spéciales

- utilisation de l'objet comme type numérique.

Exemple: Surcharge de l'addition

```
class Addition:
    def __init__(self, value):
        self.value = value
    def __add__(self, other):
        return Addition(self.value + other.value )
    def __iadd__(self, other):
        return self.__add__(other)
    def __str__(self):
        return str(self.value)
```

```
>>> a1= Addition(2)
>>> a2= Addition(3)
>>> a3= a1 + a2
>>> str(a3)
'5'
```

Les modules

Les modules

- Les modules forment un espace de noms et permettent ainsi de regrouper les définitions de fonctions et variables, en les liant à une même entité.
- Ils prennent la forme de fichiers Python (un nom et une extension .py) et leur nomenclature repose sur les même règles que les noms de variables ou de fonction:
- ✓ uniquement composés de lettres, de chiffres et d'underscores (_), et ne commençant pas par un chiffre.
- Ainsi, un fichier monModule.py correspondra à un module monModule.

Exemple:

```
def addition(a, b):
    return a + b
def soustraction(a, b):
    return a - b
```

- Pour charger le code d'un module et avoir accès à ses définitions, il est nécessaire de l'importer. On utilise pour cela le mot-clé import suivi du nom du module:

```
import monModule
```

- On pourra donc accéder aux methode addition et soustraction en faisant monModule.addition et monModule.soustraction.
- Il est possible lors de l'import de choisir un autre nom que monModule pour l'objet créé à l'aide du mot-clé as suivi du nom souhaité:

```
import monModule as mod
```

- Il est aussi possible de préciser plusieurs objets à importer en les séparant par des virgules: from monModule import addition, soustraction
  - ❖ from monModule import * // permet d'importer tous les noms présents dans le module
- La méthode help() permettra de se documenter sur le module.

Les modules

Bibliothèque standard

- Python dispose par défaut de nombreux modules déjà prêts à être utilisés. Ils sont regroupés dans ce qu'on appelle la bibliothèque standard. Ces modules apportent des fonctions concernant des domaines particuliers qui ne sont pas incluses dans l'espace de noms global pour ne pas le surcharger.

Exemple:

Dans le module math , nous retrouvons toutes les fonctions mathématiques usuelles (sqrt, exp, cos, sin) ainsi que les constantes (pi, e). import math

D'autres modules existent (a parcourir). Quelques-uns: os, sys, time, warnings, hashlib, array, …

- Si on souhaite tester des fonctionnalité on pourra placer les appels des fonctions de tests à la toute fin de notre module. Dans ce cas, il faut les inclure dans un bloc conditionnel if __name__ == '__main__':
  - Cet bloc d'instruction est sera exécuté uniquement lorsqu'on exécute le ficher module lui-même et ne sera jamais exécuté lors d'un import.

À suivre

Feedback sur:
pape.abdoulaye.barro@gmail.com

Les modules

Bibliothèque scientifique

Fin

Feedback sur:
pape.abdoulaye.barro@gmail.com