



Développeur d'Applications Python

Formation qualifiante



M104 PROGRAMMATION STRUCTURÉE EN PYTHON

Résumé théorique

Abdelkrim Ben yahia
DR Fès Meknès
Formateur en digital & IA





M104 PROGRAMMATION STRUCTURÉE EN PYTHON

OBJECTIF

Ce module permet au stagiaire de maîtriser les bases de la programmation en python

80 heures





PARTIE 1

Mettre en place un environnement Python

Chapitre 1 : Rappeler sur les techniques de la programmation structurée

Chapitre 2 : Présenter le langage Python

Chapitre 3 : Installer et configurer un environnement de développement

PARTIE 2

Appliquer les bases de la programmation en Python

Chapitre 1 : Utiliser les instructions alternatives

Chapitre 2 : Utiliser les instructions itératives

Chapitre 3 : Manipuler les Opérations arithmétique et logiques

Chapitre 4 : Définir les fonctions

PARTIE 3

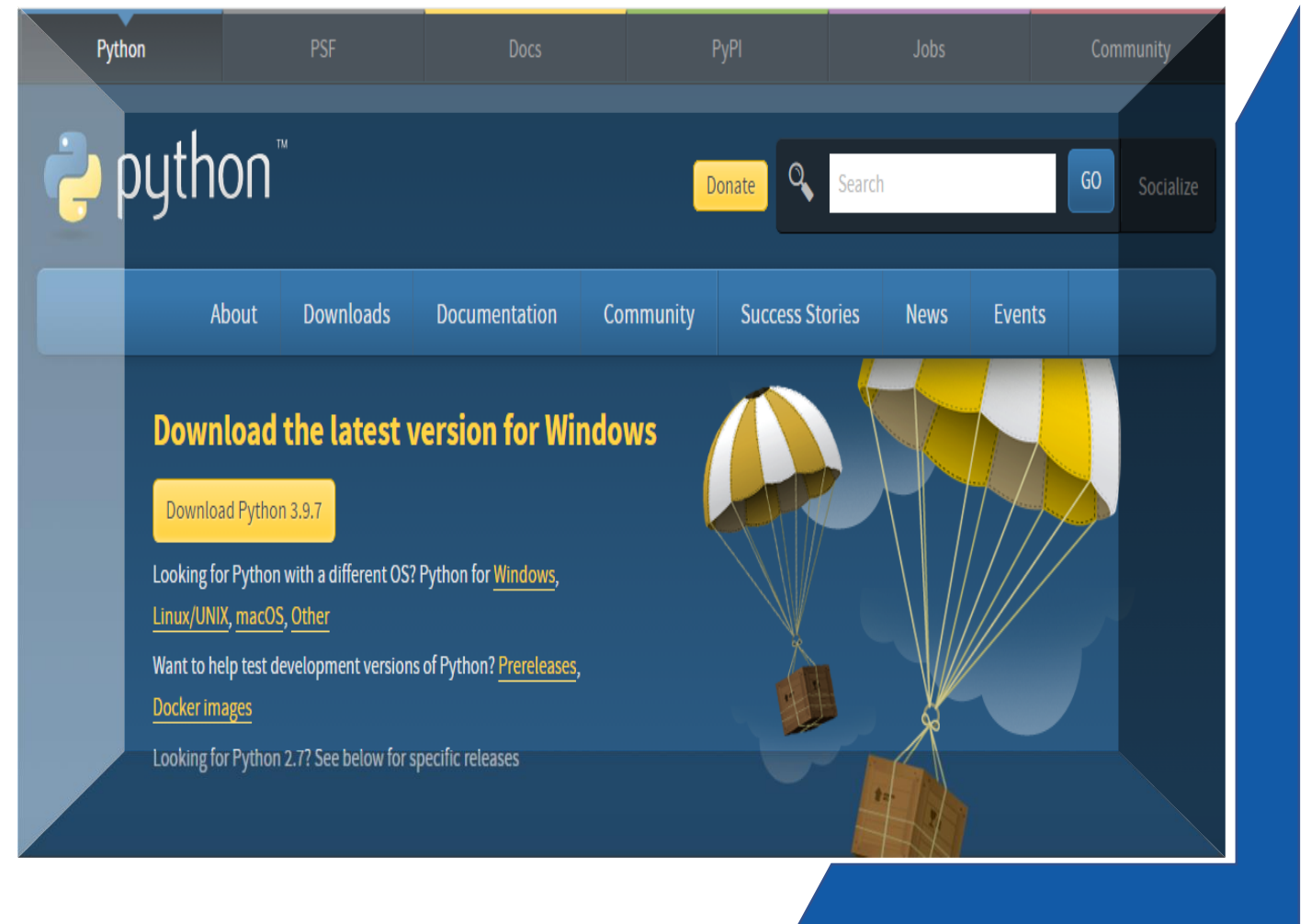
Appliquer les techniques de programmation avancée en utilisant Python

Chapitre 1 : Maîtriser les listes et les tuples

Chapitre 2 : Maîtriser les ensembles et les dictionnaires

PARTIE 1

METTRE EN PLACE UN ENVIRONNEMENT PYTHON





CHAPITRE 1

**Rappeler sur les
techniques de la
programmation
structurée**

- 1. Mettre en place un environnement Python**
2. Appliquer les bases de la programmation en Python
3. Appliquer les techniques de programmation avancée en utilisant Python

Définition de langue et langage



- Une langue est un moyen (et un outil) pour exprimer et enregistrer des pensées. Il y a beaucoup de langues tout autour de nous.
- Les langages naturels (Ex l'anglais, le français ou les gestes) représentent l'ensemble des possibilités d'expression partagé par un groupe d'individus.
- Les ordinateurs ont également leur propre langage, appelé langage machine , Il s'agit d'une suite de 0 et de 1 (du binaire) mais pour plus de "clarté il peut être décrit en hexadécimal

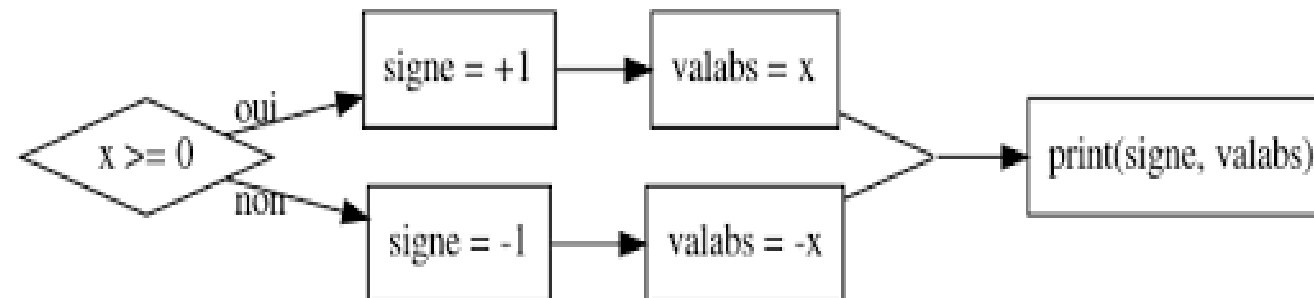


```
0001111100001010101
0011100111001101010
0101010101010000000
1010101010101010101
1010100000111110000
1010101000111000101
1010101010010100100
```

Définition d'une liste d'instructions



- Les commandes qu'il reconnaît sont très simples. On peut imaginer que l'ordinateur réponde à des commandes comme "prendre ce nombre, le diviser par un autre et sauvegarder le résultat".
- Un ensemble complet de commandes connues est appelé une liste d'instructions (l'ensemble des actions consécutives qu'un ordinateur doit exécuter)



Liste des éléments d'un langage



chaque langage (machine ou naturel, peu importe) se compose des éléments suivants:

- UN ALPHABET (symboles utilisés pour construire des mots d'une certaine langue)
- UN LEXIS (un dictionnaire)
- UNE SYNTAXE (règles utilisées pour déterminer si une certaine chaîne de mots forme une phrase valide)
- SÉMANTIQUE un ensemble de règles déterminant si une certaine phrase a du sens (Ex, "j'ai mangé une pomme" est logique, mais "une pomme m'a mangé" Ce n'est pas logique)

Description des Langages de programmation



- Les langages servant aux ordinateurs à communiquer n'ont rien à voir avec des langages informatiques, on parle dans ce cas de protocoles de communication, ce sont deux notions totalement différentes.
- Un langage machine(0,1) n'est pas compréhensible facilement par l'humain moyen.
- langages de programmation c'est un langage intermédiaire, compréhensible par l'homme, qui sera ensuite transformé en langage machine pour être exploitable par le processeur.



Définition d'un programme



Partie
1

Partie
2

Partie
3

- Un programme est une suite d'instructions que la machine doit exécuter.
- La façon d'écrire un programme est liée au langage de programmation que l'on a choisi car il en existe énormément.
- D'une façon générale, le programme est un simple fichier texte (écrit avec un traitement de texte ou un éditeur de texte), que l'on appelle fichier source.
- Le fichier source contient les lignes de programmes que l'on appelle code source

```
33 self.fingerprints = self
34 self.logdupes = True
35 self.debug = debug
36 self.logger = logging.getLogger(__name__)
37 if path:
38     self.file = os.path.join(path, 'fingerprints.log')
39     self.file.seek(0)
40     self.fingerprints.update(self, request)
41
42 @classmethod
43 def from_settings(cls, settings):
44     debug = settings.getbool('DEBUG', False)
45     return cls(job_dir(settings), debug)
46
47 def request_seen(self, request):
48     fp = self.request_fingerprint(request)
49     self.fingerprints
```

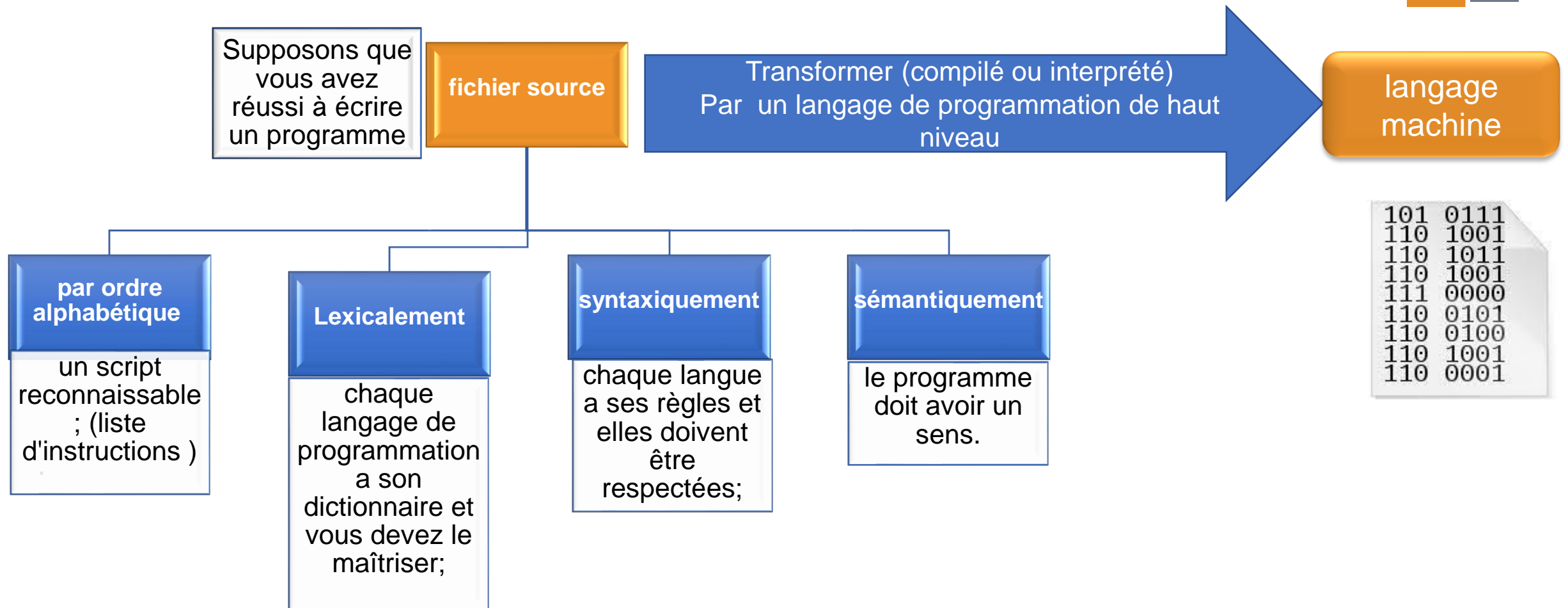
Les éléments pour réussir à écrire un programme



Partie
1

Partie
2

Partie
3

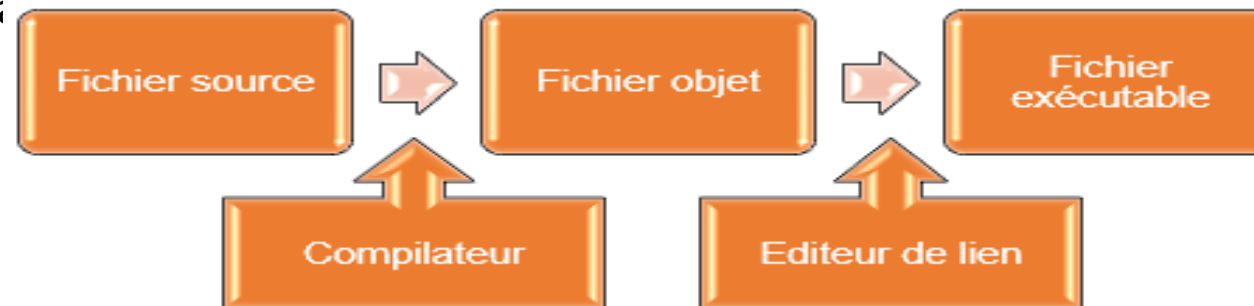


Transformation d'un programme en langage machine



Il existe deux façons différentes de transformer un programme d'un langage de programmation de haut niveau en langage machine :

1. **COMPILATION** - le programme source va être traduit une fois pour toutes par un programme annexe (le compilateur) afin de générer un nouveau fichier qui sera autonome, c'est-à-dire qui n'aura plus besoin d'un programme autre que lui pour s'exécuter (on dit d'



2. **INTERPRÉTATION** - vous a besoin d'un programme auxiliaire (l'interpréteur) pour traduire au fur et à mesure les instructions du programme; cela signifie également que vous ne pouvez pas simplement distribuer le code source tel quel, car l'utilisateur final a également besoin de l'interpréteur pour l'exécuter.

Liste d'avantages et d'inconvénient



	COMPILATION	INTERPRÉTATION
Avantages	<ul style="list-style-type: none">• l'exécution du code traduit est plus rapide;• seul l'utilisateur doit avoir le compilateur• le code traduit est stocké en langage machine	<ul style="list-style-type: none">• il n'y a pas de phases supplémentaires de traduction;• le code est stocké en utilisant le langage de programmation, pas celui de la machine.
Inconvénient	<ul style="list-style-type: none">• la compilation elle-même peut être un processus très long - vous ne pourrez peut-être pas exécuter votre code immédiatement après toute modification;• vous devez avoir autant de compilateurs que de plates-formes matérielles sur lesquelles vous voulez que votre code soit exécuté.	<ul style="list-style-type: none">• ne vous attendez pas à ce que l'interprétation accélère votre code à grande vitesse - votre code partagera la puissance de l'ordinateur avec l'interpréteur, donc il ne peut pas être vraiment rapide;• vous et l'utilisateur final devez avoir l'interprète pour exécuter votre code.



CHAPITRE 2

Présenter le langage Python

1. **Mettre en place un environnement Python**
2. Appliquer les bases de la programmation en Python
3. Appliquer les techniques de programmation avancée en utilisant Python



Définition de python

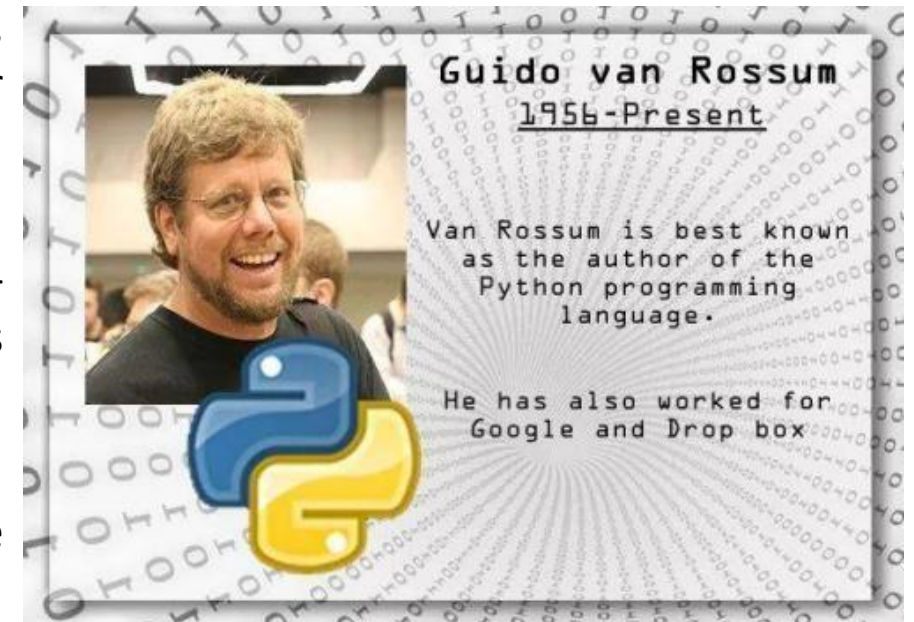


Partie
1

Partie
2

Partie
3

- Python est un langage de programmation largement utilisé, interprété, orienté objet et de haut niveau avec une sémantique dynamique, utilisé pour la programmation à usage général.
- Python a été créé par Guido van Rossum , né en 1956 à Haarlem, aux Pays-Bas. Bien sûr, Guido van Rossum n'a pas développé et fait évoluer tous les composants Python lui-même.
- le nom du langage de programmation Python vient d'une ancienne série de sketches de comédie télévisée de la BBC appelée Monty Python's Flying Circus



Objectifs de Python



En 1999, Guido van Rossum a défini ses objectifs pour Python:

- un langage simple tout aussi puissant que ceux des principaux concurrents;
- l'open source , pour que chacun puisse contribuer à son développement;
- un code aussi compréhensible que l'anglais;
- adapté aux tâches quotidiennes , permettant des temps de développement courts.

Qu'est-ce qui rend Python spécial?



Python est facile à :

- Apprendre - le temps nécessaire pour apprendre Python est plus court que pour de nombreux autres langages; cela signifie qu'il est possible de démarrer la programmation réelle plus rapidement;
- Enseigner - la charge de travail d'enseignement est plus petite que celle requise par les autres langues; cela signifie que l'enseignant peut mettre davantage l'accent sur les techniques de programmation générales (indépendantes de la langue).
- À utiliser pour écrire de nouveaux logiciels - il est souvent possible d'écrire du code plus rapidement lorsque vous utilisez Python;
- À comprendre - il est aussi souvent plus facile de comprendre le code de quelqu'un d'autre plus rapidement s'il est écrit en Python;
- À obtenir, à installer et à déployer - Python est gratuit, ouvert et multiplateforme;

Pourquoi pas Python?



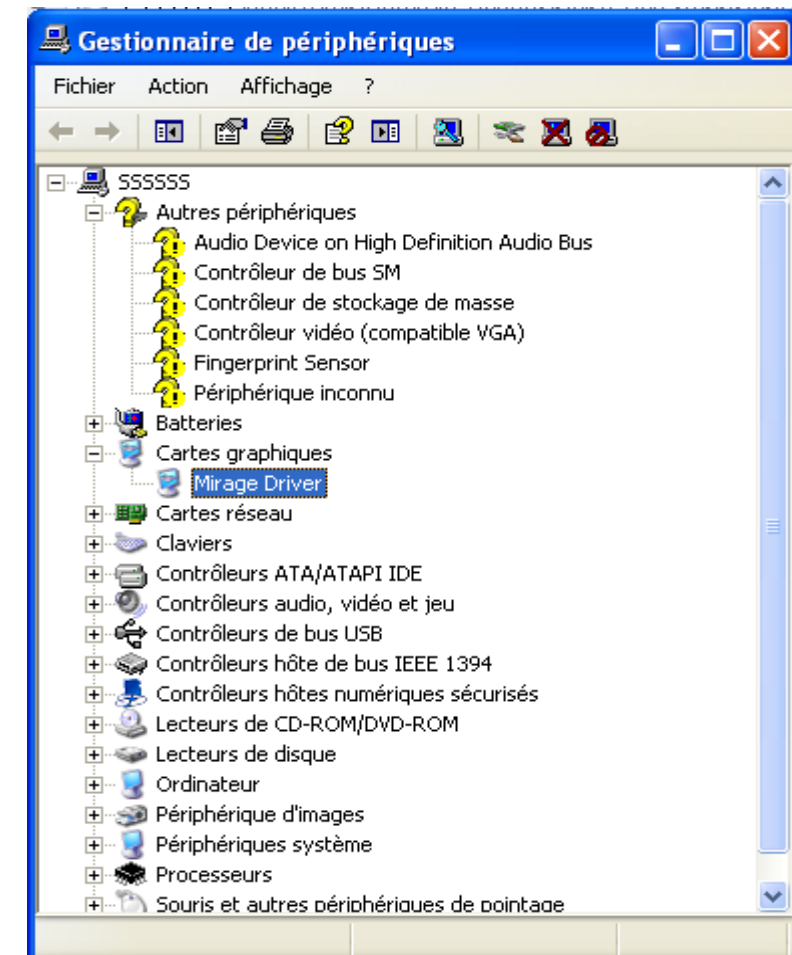
Partie
1

Partie
2

Partie
3

Malgré la popularité croissante de Python, il existe encore des niches où Python est absent ou rarement vu:

- programmation de bas niveau (parfois appelée programmation "proche du métal"): si vous voulez implémenter un pilote ou un moteur graphique extrêmement efficace, vous n'utiliseriez pas Python;



Versions de Python




Il existe deux principaux types de Python, appelés Python2 et Python3.


- Python 2 est une ancienne version du Python d'origine. Son développement a depuis été intentionnellement bloqué, bien que cela ne signifie pas qu'il n'y a pas de mises à jour.
- Python 3 est la nouvelle version (pour être précis, la version actuelle) du langage. Il suit son propre chemin d'évolution, créant ses propres normes et habitudes.




En plus de Python 2 et Python 3, il existe plusieurs versions de chacun.

- CPython est l'implémentation de référence du langage de programmation Python . Écrit en C et Python, CPython est l'implémentation par défaut et la plus utilisée du langage.

 Cython est l'une des solutions possibles pour Les calculs mathématiques volumineux et complexes peuvent être facilement codés en Python, mais l'exécution du code résultant peut être extrêmement longue.

 Jython, anciennement nommé JPython, est un interprète Python écrit en Java, créé en 1997 actuellement suit les normes Python 2, Jusqu'à présent, il n'y a pas de Jython conforme à Python 3.

 PyPy - un Python dans un Python. En d'autres termes, il représente un environnement Python écrit en langage de type Python nommé RPython (Restricted Python). Il s'agit en fait d'un sous-ensemble de Python.



CHAPITRE 3

Installer et configurer un environnement de développement

1. **Mettre en place un environnement Python**
2. Appliquer les bases de la programmation en Python
3. Appliquer les techniques de programmation avancée en utilisant Python

Comment Obtenir Python ?



Il existe plusieurs façons d'obtenir votre propre copie de Python 3, selon le système d'exploitation que vous utilisez.

- Les utilisateurs de Linux ont probablement Python déjà installé - c'est le scénario le plus probable, car l'infrastructure de Python est intensivement utilisée par de nombreux composants du système d'exploitation Linux.
- Si vous êtes un utilisateur Linux, ouvrez le terminal / la console et tapez: `python3`
- Tous les utilisateurs non Linux peuvent télécharger une copie sur <https://www.python.org/downloads/> .
- Si vous êtes un utilisateur Windows , démarrez le fichier .exe téléchargé et suivez toutes les étapes. Laissez les paramètres par défaut suggérés par le programme d'installation pour le moment, à une exception près - regardez la case à cocher intitulée Ajouter Python 3.x à PATH et cochez-la.
- Si vous êtes un utilisateur de macOS , une version de Python 2 peut déjà avoir été préinstallée sur votre ordinateur, mais puisque nous travaillerons avec Python 3, vous devrez toujours télécharger et installer le fichier .pkg correspondant à partir du site Python.

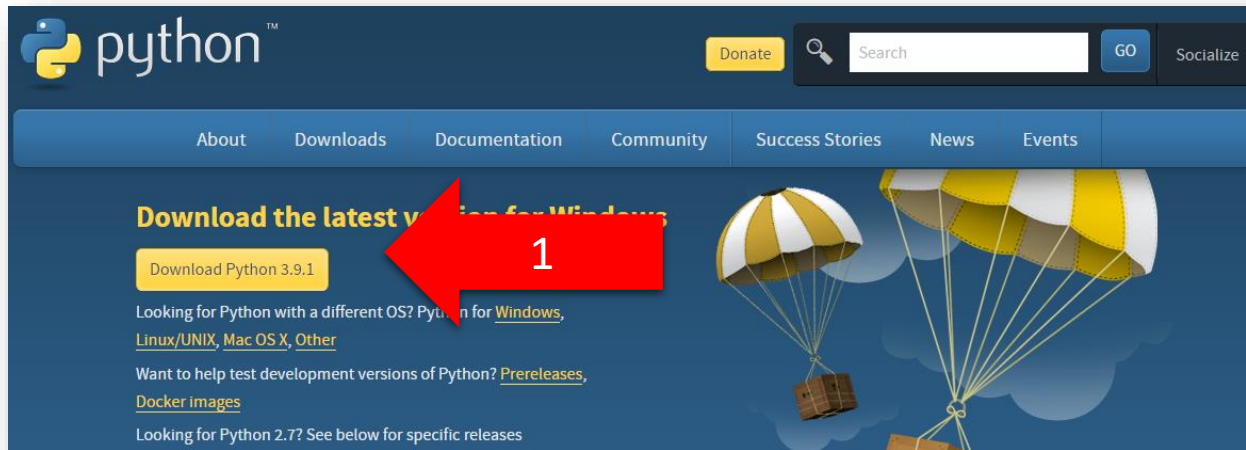
Préparation de l'environnement Python




Partie
1

Partie
2

Partie
3



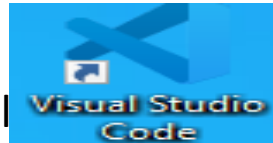
Nom	Modifié le	Type	Taille
 python-3.9.1-amd64.exe		Application	27 544 Ko

Outils pour utiliser Python



Il existe de nombreuses façons d'utiliser Python, pour commencer votre travail, vous avez besoin des outils suivants:

- un éditeur qui vous aidera à écrire le code; cet éditeur dédié vous donnera plus que l'équipement OS standard;



- une console dans laquelle vous pouvez lancer le code que vous venez d'écrire, vous pouvez le faire de nouveau, vous pouvez le modifier, vous pouvez l'arrêter de force lorsqu'il devient hors de contrôle;



Invite de commandes

- un outil nommé débogueur , capable de lancer pas a pas votre code et vous permettant de l'inspecter à chaque instant d'exécution.

Description d'IDLE Python



- Outre ses nombreux composants utiles, l'installation standard de Python 3 contient une application très simple mais extrêmement utile nommée IDLE.
- IDLE est un acronyme: développement intégré et environnement d'apprentissage.
- Parcourez les menus de votre système d'exploitation, recherchez IDLE quelque part sous Python 3.x et lancez-le.

```
IDLE Shell 3.9.6
File Edit Shell Debug Options Window Help
Python 3.9.6 (tags/v3.9.6:db3ff76, Jun 28 2021, 15:26:21) [MSC v.192
9 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information
.>>>
```

PARTIE 2

APPLIQUER LES BASES DE LA PROGRAMMATIO N EN PYTHON



CHAPITRE 1

Utiliser les instructions alternatives

1. Mettre en place un environnement Python
- 2. Appliquer les bases de la programmation en Python**
3. Appliquer les techniques de programmation avancées en utilisant Python



Les différents opérateurs de comparaison 1:



Partie
1

Partie
2

Partie
3

Pour poser des questions, Python utilise un ensemble d'opérateurs très spéciaux .

- L'opérateur ==(égal à) compare les valeurs de deux opérandes. S'ils sont égaux, le résultat de la comparaison est True. S'ils ne sont pas égaux, le résultat de la comparaison est False.
- L'opérateur !=(différent de) compare également les valeurs de deux opérandes. Voici la différence: s'ils sont égaux, le résultat de la comparaison est False. S'ils ne sont pas égaux, le résultat de la comparaison est True.
- l'opérateur > (supérieur à) et l'opérateur < (inférieur à)
- l'opérateur >=(supérieur ou égal à) et l'opérateur <=(inférieur ou égal à)

Priorité	Opérateur	
1	~, +, -	unaire
2	**	
3	*, /, //, %	
4	+, -	binaire
5	<<, >>	
6	<, <=, >, >=	
7	==, !=	
8	&	
9		
10	=, +=, -=, *=, /=, %=, &=, ^=, =, >>=, <<=	

Les différents opérateurs de comparaison 2:



Opérateur	La description	Exemple (x = 0, y = 1 et z = 0)
==	Renvoie True si les valeurs des opérandes sont égales, et False sinon	x == y # False x == z # True
!=	renvoie True si les valeurs des opérandes ne sont pas égales, et False sinon	x != y # True x != z # False
>	True si la valeur de l'opérande gauche est supérieure à la valeur de l'opérande droit, et False sinon	x > y # False y > z # True
<	True si la valeur de l'opérande gauche est inférieure à la valeur de l'opérande droit, et False sinon	x < y # True y < z # False
>=	True si la valeur de l'opérande gauche est supérieure ou égale à la valeur de l'opérande droit, et False sinon	x >= y # False x >= z # True y >= z # True
<=	True si la valeur de l'opérande gauche est inférieure ou égale à la valeur de l'opérande droit, et False sinon	x <= y # True x <= z # True y <= z # False

Syntaxe de if else:



Lorsque vous souhaitez exécuter du code uniquement si une certaine condition est remplie, vous pouvez utiliser une instruction conditionnelle :

- une seule déclaration if, par exemple:

```
if x == 10: # condition
    print("x est égal à 10") # exécuté si la condition est Vrai
```

- une déclaration if-else, par exemple:

```
if x < 10: # condition
    print("x est inférieur à 10") # exécuté si la condition est Vrai
else:
    print("x est supérieur ou égal à 10") # exécuté si la condition est
Faux
```

Description d'éléments de déclaration conditionnelle



déclaration conditionnelle comprend uniquement les éléments suivants

- le mot – clé if;
- un ou plusieurs espaces blancs;
- une expression (une question ou une réponse)
- un deux point (:) suivi d'un saut de ligne;
- une instruction en retrait(quatre espaces ou tabulation) ou un ensemble d'instructions (au moins une instruction est absolument requise);

```
# exemple
if age>=18:
    print('Adulte')
else:
    print('enfant')
```

Syntaxe de if-elif-else



Partie
1

Partie
2

Partie
3

- La déclaration if-elif-else, par exemple:

```
if x > 15: # False
    print("x > 15")
elif x > 10: # False
    print("x > 10")
else:
    print("sinon ne sera pas exécuté")
```

- Instructions conditionnelles imbriquées, par exemple:

```
if x > 5: # True
    if x == 6: # False
        print("nested: x == 6")
    elif x == 10: # True
        print("nested: x == 10")
    else:
        print("imbriqué: else")
else:
    print("else")
```



CHAPITRE 2

Utiliser les instructions itératives

1. Mettre en place un environnement Python
- 2. Appliquer les bases de la programmation en Python**
3. Appliquer les techniques de programmation avancées en utilisant Python



Syntaxe de la boucle Tant que



Il existe deux types de boucles en Python: while et for:

- la boucle while exécute une instruction ou un ensemble d'instructions tant qu'une condition booléenne spécifiée est vraie, par exemple:

Example 1

```
while True:  
    print("Coincé dans une boucle infinie.")
```

Example 2

```
compteur= 5  
while compteur> 2:  
    print(compteur)  
    compteur -= 1
```

Syntaxe de la boucle Pour



- la boucle for exécute plusieurs fois un ensemble d'instructions; il est utilisé pour parcourir une séquence (par exemple, une liste, un dictionnaire, un tuple ou un ensemble - vous en apprendrez bientôt plus) ou d'autres objets qui sont itérables (par exemple, des chaînes). Vous pouvez utiliser la boucle for pour parcourir une séquence de nombres à l'aide de la fonction range intégrée. Regardez les exemples ci-dessous:

Example 1

```
mot= "Python"
```

```
for lettre in mot:
```

```
    print(lettre, end="*")
```

Example 2

```
for i in range(1, 10):
```

```
    if i % 2 == 0:
```

```
        print(i)
```

Description de la fonction range()



- La fonction range() génère une séquence de nombres. Il accepte des entiers et renvoie des objets de plage. La syntaxe de range() ressemble à \rightarrow range(start, stop, step) où:
 - ✓ Start est un paramètre facultatif spécifiant le numéro de départ de la séquence (0 par défaut)
 - ✓ stop est un paramètre facultatif spécifiant la fin de la séquence générée (il n'est pas inclus),
 - ✓ et step est un paramètre facultatif spécifiant la différence entre les nombres dans la séquence (1 par défaut.)

Example 1

```
for i in range(3):  
    print(i, end=" ") # résultat: 0 1 2
```

Example 2

```
for i in range(6, 1, -2):  
    print(i, end=" ") # résultat: 6, 4, 2
```


Utilisation des instructions break et continue



- Vous pouvez utiliser les instructions break et continue pour modifier le flux d'une boucle:
 - ✓ Vous utilisez break pour quitter une boucle, par exemple:
 - ✓ Vous utilisez continue pour ignorer l'itération actuelle et continuer avec l'itération suivante, par exemple:

```
# Example break
```

```
text = "OpenEDG Python Institute"
```

```
for lettre in text:
```

```
    if lettre == "P":  
        break
```

```
    print(lettre, end="")
```

```
# Example continue
```

```
text = "pyxpyxpyx"
```

```
for lettre in text:
```

```
    if lettre == "x":  
        continue
```

```
    print(lettre, end="")
```

Utilisation de la clause else avec les boucles



- Les boucles while et for peuvent également avoir une clause else en Python. La clause else s'exécute une fois que la boucle a terminé son exécution tant qu'elle n'a pas été terminée par break, par exemple:
- Vous utilisez break pour quitter une boucle, par exemple:

Example while

```
n = 0
while n != 3:
    print(n)
    n += 1
else:
    print(n, "else")
```

Example for

```
for i in range(0, 3):
    print(i)
else:
    print(i, "else")
```



CHAPITRE 3

Manipuler les Opérations arithmétique et logiques

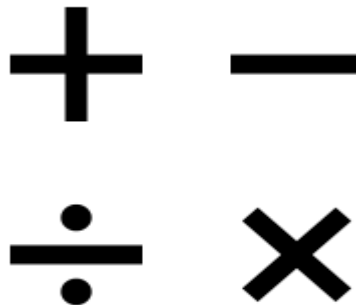
1. Mettre en place un environnement Python
- 2. Appliquer les bases de la programmation en Python**
3. Appliquer les techniques de programmation avancées en utilisant Python



Liste des opération arithmétique



- Les techniques des quatre opérations de base (+, -, * et /) restent exactement les mêmes qu'en notation décimale.
- Seules changent d'une part la forme de la suite de chiffres qui exprime le résultat (elle ne compte que des zéros et un).
- d'autre part la signification de cette suite (10 signifie « deux » et non « dix », 100 signifie « quatre » et non « cent », etc.).



Utilisation d'addition et soustraction



- On passe d'un nombre binaire au suivant en ajoutant 1, comme en décimal, sans oublier les retenues et en utilisant la table ordinaire (mais réduite à sa plus simple expression) :

$$\begin{array}{llll} 0 + 0 = 0 & 0 + 1 = 1 & 1 + 0 = 1 & 1 + 1 = 0 \text{ avec } 1 \text{ retenue} \\ 0 - 0 = 0 & 0 - 1 = 1 \text{ avec } 1 \text{ retenue} & 1 - 0 = 1 & 1 - 1 = 0 \end{array}$$

Utilisation de multiplication et division



- Multiplier par deux se fait en décalant chaque chiffre d'un cran à gauche et en insérant un zéro à la fin.

- Par exemple, deux fois onze :

```
1011 onze
//// décalage et insertion de 0
10110 vingt-deux
```

- La division entière par deux se fait en décalant chaque chiffre d'un cran à droite, le chiffre de droite étant le reste supprimé.

- Par exemple onze divisé par deux :

```
1011 onze
\\ décalage et suppression du chiffre à droite
101 cinq reste un
```

Utilisation de la représentation en binaire signé



Partie
1

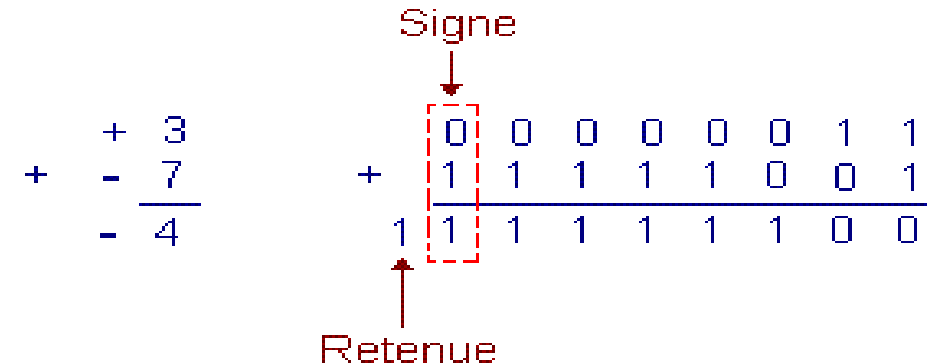
Partie
2

Partie
3

- Pour compléter la représentation des entiers, il faut pouvoir écrire des entiers négatifs.
- la représentation en binaire signé, le bit de poids fort sert à représenter le signe (0 positif et 1 négatif).

0010 = +2 en décimal et 1010 = -2 en décimal

- Cette représentation possède deux inconvénients



Les inconvénients de bit de poids fort

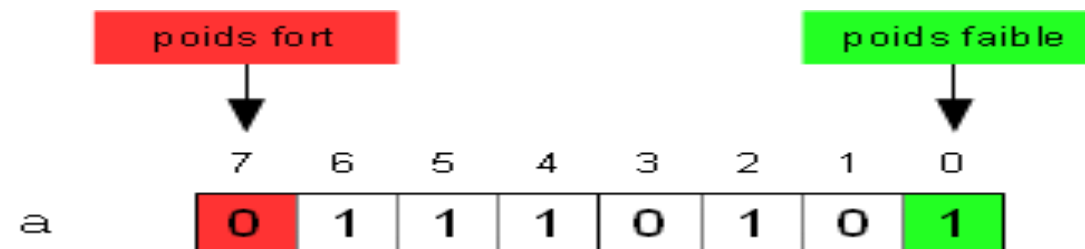


- Le premier (mineur) est que le nombre zéro (0) possède deux représentations :

(+ 0 \rightarrow 0000, - 0 \rightarrow 1000)

- L'autre inconvénient (majeur) est que cette représentation impose de modifier l'algorithme d'addition ; si un des nombres est négatif, l'addition binaire usuelle donne un résultat incorrect :

Soit $3 + (-4) = (-7)$ au lieu de (-1) ($0011 + 1100 = 1111$).



Représentation des entiers négatifs



- Deux représentations existent, le complément à un et le complément à deux.
- Complément à un: consiste à inverser la valeur de chaque bit.
- Le complément à deux consiste à réaliser un complément à un, puis d'ajouter 1 et on ignore alors la retenue sur le bit de poids fort s'il existe.

Solutions de complément à un



- Par exemple pour obtenir -7 :

```
0111 sept  
1000 moins sept
```

- Un défaut de ce système est que zéro a deux représentations : 0000 et 1111 (« +0 » et « -0 »). Il n'est pas utilisé par les ordinateurs courants.

Solutions de complément à deux



- le complément à deux est une opération sur les nombres binaires qui permet d'effectuer simplement des opérations arithmétiques sur les entiers relatifs.

- Le complément à deux opère toujours sur des nombres binaires ayant le même nombre de bits (celui-ci est généralement un multiple de 4). Dans une telle écriture, le bit de poids fort donne le signe du nombre représenté (positif ou négatif).

- Par exemple pour obtenir -7 :

```
0111 sept
1000 complément à un
1001 complément à deux en ajoutant 1
```
- Ce codage a l'avantage de ne pas nécessiter de différenciation spéciale des nombres positifs et négatifs, et évite en particulier le problème de double représentation du zéro

- Voici une addition de -7 et +9 réalisée en complément à deux sur 4 bits :

```
-7      1001
+9      1001
-----
 2      (1) 0010 (on « ignore » la retenue)
```

```
- 4      1110
+3      0011
-----
- 1      1111 complément à deux donne 0001
```

Utilisation des opérateurs logiques



Partie
1

Partie
2

Partie
3

- Les opérateurs élémentaires sont des opérations binaires réalisées simultanément sur l'ensemble des bits d'un mot de manière indépendante.
- Python prend en charge les opérateurs logiques suivants: NOT, AND, OR, XOR

NON [NOT] $a \rightarrow \neg a$

$a \oplus b$ OU-EX [XOR]

ET [AND] $a \rightarrow ab$

\overline{ab} NON-ET [NAND]

OU [OR] $a \rightarrow a + b$

$\overline{a + b}$ NON-OU [NOR]

Utilisation de l'opérateur logique NOT et XOR



- Représente la négation logique, le complément d'une expression. Chaque bit est inversé.

- ✓ Par exemple, sur 4 bits, NOT 7 = 8 :

```
NOT 0111
    = 1000
```

- ✓ not → retourne faux si le résultat est vrai, et retourne vrai si le résultat est faux, par exemple, not True est False.

- Le ou exclusif de deux expressions.

- ✓ Par exemple, 5 XOR 3 = 6:

```
    0101
XOR 0011
    = 0110
```

- ✓ XOR → (OU exclusif) retourne True si les deux opérandes ont des valeurs distinctes seulement et retourne False pour le reste.

Utilisation de l'opérateur logique AND et OR



- Le et logique de deux expressions.

✓ Par exemple, 5 AND 3 = 1 :

```
0101
AND 0011
= 0001
```

✓ and → si les deux opérandes sont vrais, la condition est vraie, par exemple, (True and True) est True,

- Le ou logique de deux expressions.

✓ Par exemple, 5 OR 3 = 7:

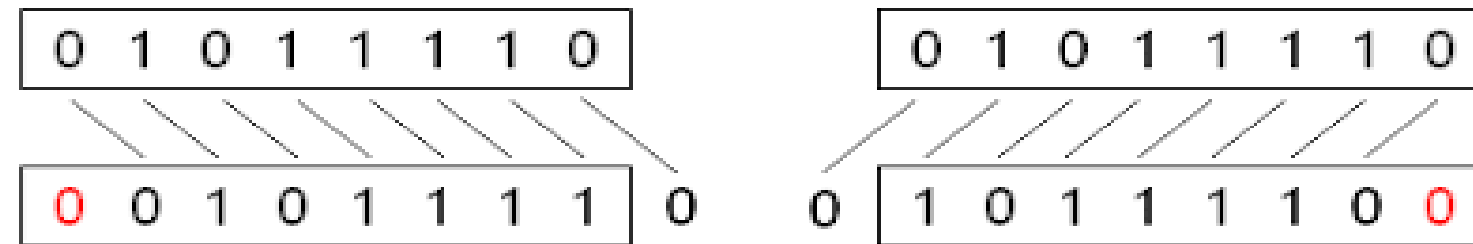
```
0101
OR 0011
= 0111
```

✓ or → si l'un des opérandes est vrai, la condition est vraie, par exemple, (True or False) est True,

Utilisation de décalages logiques



- consiste à décaler tous les chiffres d'un ou plusieurs crans vers la gauche ou la droite.
- Un décalage logique consiste à supprimer un bit d'un côté du vecteur pour le remplacer par un zéro de l'autre côté.
- Cette propriété est souvent utilisée par certains compilateurs, qui préfèrent utiliser des instructions de décalages (qui sont très rapides) à la place d'instructions de multiplication ou de division qui ont une vitesse moyenne ou lente.



Décalage logique

Utilisation de décalage à gauche logique (multiplication)

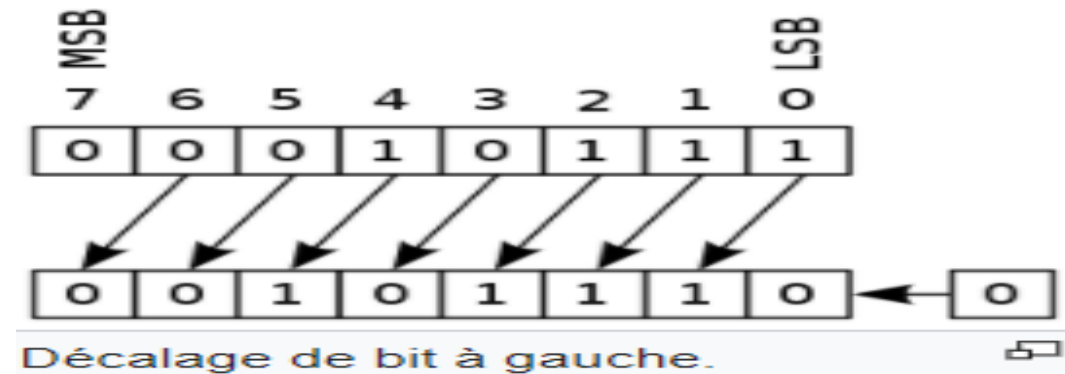


Partie
1

Partie
2

Partie
3

- Ex1 : 00010111 (+23) LEFT-SHIFT
= 00101110 (+46)



- un décalage vers la gauche de n rangs est équivalent à une multiplication par 2^n pour des entiers non-signés ou pour des entiers signés positifs.
- Avec des nombres signés, ce n'est pas le cas : on obtient un résultat qui n'a pas grand sens mathématiquement.

Utilisation de décalage à droite logique (division)



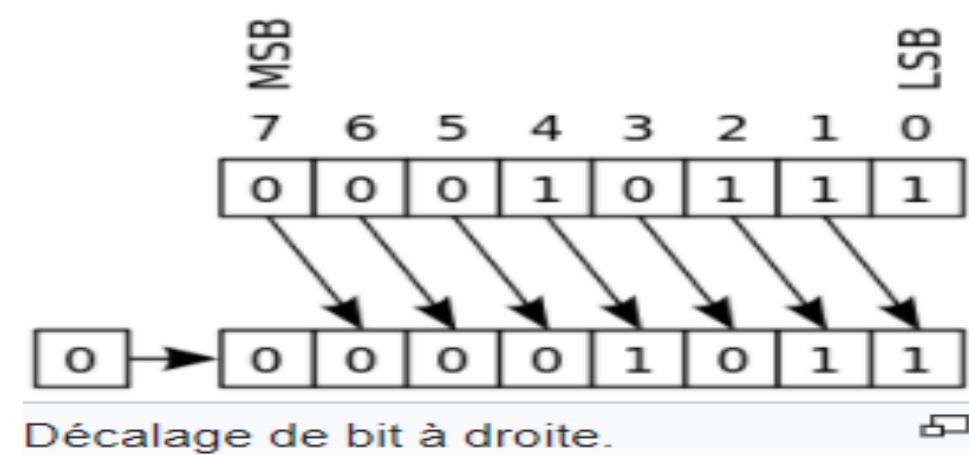
Partie
1

Partie
2

Partie
3

- Ex :

```
00010111 (+23) RIGHT-SHIFT  
= 00001011 (+11)
```



- le décalage vers la droite qui revient à diviser un nombre entier par 2^n

Utilisation de décalage arithmétique



Partie
1

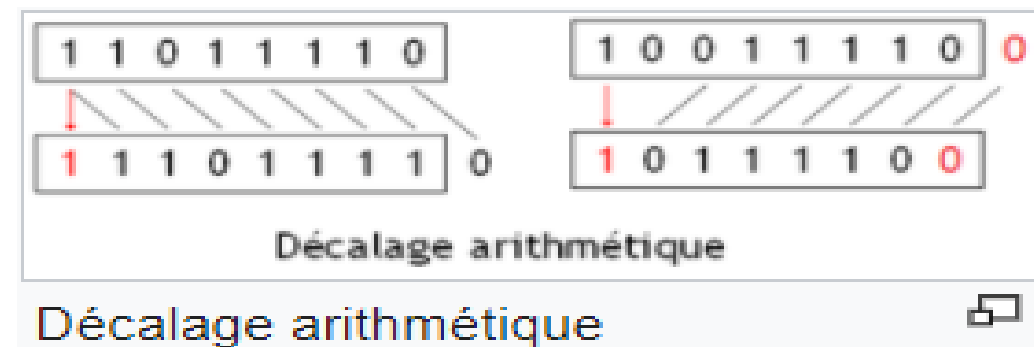
Partie
2

Partie
3

- Les décalages arithmétiques sont similaires aux décalages logiques, à un détail près :
- Le décalage à gauche se comporte comme un décalage logique.
- pour les décalages à droite, le bit de signe de notre nombre n'est pas modifié, et on remplit les vides laissés par le décalage avec le bit de signe.

```
00110111 (+55) RIGHT-SHIFT  
= 00011011 (+27)
```

```
10111010 (-70) RIGHT-SHIFT  
= 11011101 (-35)
```



Les opérations de bit à bit



- une opération bit à bit est un calcul manipulant les données directement au niveau des bits, selon une arithmétique booléenne.
- Elles sont utiles dès qu'il s'agit de manipuler les données à bas niveau : codages, couches basses du réseau (par exemple TCP/IP), cryptographie, etc...
- Les opérations bit à bit courantes comprennent des opérations logiques bit par bit et des opérations de décalage des bits, vers la droite ou vers la gauche.

Utilisation des opérateurs au niveau du bit



- Vous pouvez utiliser des opérateurs au niveau du bit pour manipuler des bits de données uniques. Les exemples de données suivants:

$x = 15$, qui est 0000 1111 en binaire,

$y = 16$, qui est 0001 0000 en binaire.

- sera utilisé pour illustrer la signification des opérateurs, exemples ci-dessous:

- ✓ & fait au niveau du bit et, ex: $x \& y = 0$, qui est 0000 0000 en binaire,
- ✓ | fait un bit ou, par exemple $x | y = 31$, qui est 0001 1111 en binaire,
- ✓ ~ fait un pas au niveau du bit, ex: $\sim x = 240$ (-16), qui est 1111 0000 en binaire,
- ✓ ^ fait un xor au niveau du bit, ex: $x \wedge y = 31$, qui est 0001 1111 en binaire,
- ✓ >> fait un décalage à droite au niveau du bit, ex: $y \gg 1 = 8$, qui est 0000 1000 en binaire $\lfloor (y//2) \rfloor$,
- ✓ << fait un décalage à gauche au niveau du bit, ex: $y \ll 3 = 128$, qui est 1000 0000 en binaire $\lfloor (y*2) \rfloor$,



CHAPITRE 4

Définir les fonctions

1. Mettre en place un environnement Python
- 2. Appliquer les bases de la programmation en Python**
3. Appliquer les techniques de programmation avancées en utilisant Python



Définition d'une fonction




Partie
1

Partie
2

Partie
3

- Une fonction est un bloc de code qui exécute une tâche spécifique lorsque la fonction est appelée (invoquée). Vous pouvez utiliser des fonctions pour rendre votre code réutilisable, mieux organisé et plus lisible. Les fonctions peuvent avoir des paramètres et des valeurs de retour.

```
1  print "Below is a block of code inside of a function!"
2
3
4  def my_function():
5      print "This is inside of the function."
6      my_variable = 10
7      print "All of this makes up the my_function block of code!"
8
9
10 print "This is not part of that block of code inside the function."
11 print "We can tell because of the indentation!"
```

- Il existe au moins quatre types de fonctions de base en Python:
 - ✓ fonctions intégrées qui font partie intégrante de Python (comme la fonction `print()`). Vous pouvez voir une liste complète des fonctions intégrées de Python sur 
 - ✓ ceux qui proviennent de modules préinstallés
 - ✓ fonctions définies par l'utilisateur qui sont écrites par les utilisateurs pour les utilisateurs - vous pouvez écrire vos propres fonctions et les utiliser librement dans votre code,
 - ✓ les fonctions lambda

Création d'une fonction



- Vous pouvez définir votre propre fonction à l'aide du mot – clé def et de la syntaxe suivante:

Syntaxe #	Exemple # 1	Exemple # 2
<pre>def yourFunction(optional parameters): # the body of the function</pre>	<pre>def message(): # définir la fonction print("Hello") # corps de la fonction message() # l'appel de la fonction</pre>	<pre>def hello(name): # définir la fonction print("Hello,", name) # le corps name = input("Enter your name: ") hello(name) # l'appel de la fonction</pre>

Utilisation des fonctions à l'aide de paramètres



- . Vous pouvez transmettre des informations aux fonctions à l'aide de paramètres. Vos fonctions peuvent avoir autant de paramètres que vous le souhaitez.

Syntaxe # 1	Exemple # 2	Exemple # 3
<pre>def hi(nom): print("Hi,", nom) hi("omar")</pre>	<pre>def hiAll(nom1, nom2): print("Hi,", nom2) print("Hi,", nom1) hiAll("yahia", "fatima")</pre>	<pre>def address(rue, ville, postalCode): print("Your address is:", rue, "St.", ville, postalCode) s = input("rue: ") pC = input("Postal Code: ") c = input("ville: ") address(s, c, pC)</pre>

Utilisation des arguments dans une fonction



- Vous pouvez passer des arguments à une fonction en utilisant les techniques suivantes:
 - ✓ passage de l'argument positionnel dans lequel l'ordre des arguments a réussi Ex1
 - ✓ passage d'un mot clé (nommé) dans lequel l'ordre des arguments transmis n'a pas d'importance (Ex.2),
 - ✓ un mélange d'arguments positionnels et de mots-clés passant (Ex. 3).
- Il est important de se rappeler que les arguments positionnels ne doivent pas suivre les arguments des mots clés Ex4.

Exemple # 1

```
def subtra(a, b):  
    print(a - b)  
subtra(5, 2) # résultat: 3  
subtra(2, 5) # résultat: -3
```

Exemple # 2

```
def subtra(a, b):  
    print(a - b)  
subtra(a=5, b=2) # résultat: 3  
subtra(b=2, a=5) # résultat: 3
```

Exemple # 3

```
def subtra(a, b):  
    print(a - b)  
subtra(5, b=2) # résultat: 3  
subtra(5, 2) # résultat: 3
```

Exemple # 4

```
def subtra(a, b):  
    print(a - b)  
subtra(5, b=2) # résultat: 3  
subtra(a=5, 2) # Syntax Error
```

Utilisation d'argument par mot clé



- Vous pouvez utiliser la technique de passage d'argument par mot clé pour prédéfinir une valeur pour un argument donné:

Exemple # 1

```
def name(prénom, nom="benyahia"):
    print(prénom, nom)
```

```
name("abdel")    # résultat: abdel benyahia
```

```
name("omar", "yahyaoui")    # résultat: omar yahyaoui (l'argument mot-clé remplacé par "yahyaoui")
```

Utilisation de mot - clé return



- Vous pouvez utiliser le mot - clé return pour indiquer à une fonction de renvoyer une valeur. L'instruction return quitte la fonction, par exemple:

Exemple # 1

```
def multiply(a, b):  
    return a * b  
print(multiply(3, 4))  # résultat: 12  
  
def multiply(a, b):  
    return  
print(multiply(3, 4))  # résultat: None
```

résultat affecté à une variable



- Le résultat d'une fonction peut être facilement affecté à une variable, par exemple:

Exemple # 1

```
def wishes():
```

```
    return "Happy Birthday!"
```

```
w = wishes()
```

```
print(w) # résultat: Happy Birthday!
```

Example 2

```
def wishes():
```

```
    print("My Wishes")
```

```
    return "Happy Birthday"
```

```
wishes() # résultat: My Wishes
```

Example 3

```
def wishes():
```

```
    print("My Wishes")
```

```
    return "Happy Birthday"
```

```
print(wishes()) # résultat: My Wishes
```

```
# Happy Birthday
```

Utilisation des listes comme argument d'une fonction



- Vous pouvez utiliser une liste comme argument d'une fonction, par exemple 1:
- Une liste peut également être un résultat de fonction, par exemple 2:

Exemple # 1

```
def hiEverybody(myList):  
    for name in myList:  
        print("Hi,", name)  
hiEverybody(["Adam", "John", "Lucy"])
```

Example 2

```
def createList(n):  
    myList = []  
    for i in range(n):  
        myList.append(i)  
    return myList  
print(createList(5))
```

Utilisation des variables en dehors



- Une variable qui existe en dehors d'une fonction a une portée à l'intérieur du corps de la fonction (exemple 1) sauf si la fonction définit une variable du même nom (exemple 2 et exemple 3), par exemple:

Exemple # 1

```
var = 2
def multByVar(x):
    return x * var
print(multByVar(7)) # résultat:
14
```

Example 2

```
var = 2
def mult(x):
    var = 5
    return x * var
print(mult(7)) # résultat: 35
```

Example 3

```
def multip(x):
    var = 7
    return x * var
var = 3
print(multip(7)) # résultat: 49
```

Utilisation des variables à l'intérieur



- Une variable qui existe à l'intérieur d'une fonction a une portée à l'intérieur du corps de la fonction (exemple 4), par exemple:
- Vous pouvez utiliser le mot - clé global suivi d'un nom de variable pour rendre la portée de la variable globale, par exemple 5:

Exemple # 4

```
def adding(x):  
    var = 7  
    return x + var  
print(adding(4))  # résultat : 11  
print(var)  # NameError
```

Exemple # 5

```
var = 2  
print(var)  # résultat : 2  
def retVar():  
    global var  
    var = 5  
    return var  
print(retVar())  # résultat : 5  
print(var)  # résultat : 5
```


Description d'une fonction récursive



- Une fonction peut appeler d'autres fonctions ou même elle-même.
- Lorsqu'une fonction s'appelle elle-même, cette situation est connue sous le nom de récursivité
- la fonction qui s'appelle elle-même et contient une condition de terminaison spécifiée (c'est-à-dire le cas de base - une condition qui ne dit pas à la fonction de faire d'autres appels à cette fonction) est appelée fonction récursive .
- Vous pouvez utiliser des fonctions récursives en Python pour écrire du code propre et élégant et le diviser en morceaux plus petits et organisés .
- D'un autre côté, vous devez être très prudent car il pourrait être facile de faire une erreur et de créer une fonction qui ne se termine jamais .
- Vous devez également vous rappeler que les appels récursifs consomment beaucoup de mémoire et peuvent donc parfois être inefficaces.

Utilisation de la fonction factorielle



- La fonction factorielle est un exemple classique de la mise en pratique du concept de récursivité:

```
Exemple # factorial
# Implémentation récursive de la fonction factorielle
def factorial(n):
    if n == 1: # le cas de base (condition de
termination)
        return 1
    else:
        return n * factorial(n - 1)

print(factorial(4)) # 4 * 3 * 2 * 1 = 24
```

PARTIE 3

APPLIQUER LES
TECHNIQUES DE
PROGRAMMATIO
N AVANCÉE EN
UTILISANT
PYTHON





CHAPITRE 1

Maitriser les listes et les tuples

1. Mettre en place un environnement Python
2. Appliquer les bases de la programmation en Python
- 3. Appliquer les techniques de programmation avancée en utilisant Python**



Manipulation d'indexation d'une liste



- La liste est un type de données en Python utilisé pour stocker plusieurs objets . Il s'agit d'une collection ordonnée et modifiable d'éléments séparés par des virgules entre crochets, par exemple:

✓ `myList = [1, None, True, "je suis un texte", 256, 0]`

- Les listes peuvent être indexées et mises à jour , par exemple:

```
myList = [1, '?', True, 'un texte', 256, 0]
```

```
print(myList[3]) # résultat: un texte
```

```
print(myList[-1]) # résultat: 0
```

```
myList[1] = None
```

```
print(myList) # résultat: [1, None, True, 'un texte', 256, 0]
```

```
myList.insert(0, "premier")
```

```
myList.append("dernier")
```

```
print(myList) # résultat: ['premier', 1, None, True, 'un texte', 256, 0, 'dernier']
```

Opérations sur les listes 1



- Si vous avez une liste l1, l'affectation suivante: l2 = l1 ne fait pas de copie de la liste l1, mais crée les variables l1 et l2 pointant vers une seule et même liste en mémoire Ex1.
- Si vous souhaitez copier une liste ou une partie de la liste, vous pouvez le faire en effectuant un découpage Ex 2:

Exemple # 1

```
autoUn = ['car', 'bicycle', 'motor']  
print(autoUn) # resultat: ['car', 'bicycle', 'motor']  
autoDeux = autoUn  
del autoUn[0] # supprimer 'car'  
print(autoDeux) # resultat: ['bicycle', 'motor']
```

Exemple # 2

```
colors = ['red', 'green', 'orange']  
copyUnColor = colors[:] # copie Entière de liste  
copyPColor = colors[0:2] # copie d'une partie
```

Opérations sur les listes 2



- Les listes peuvent être imbriquées , Ex: myList = [1, 'a', ["list", 64, [0, 1], False]].
- Les éléments de liste et les listes peuvent être supprimés ,ex:

```
myList = [1, 2, 3, 4]
del myList[2]
del myList[-1] #supprimer dernier élément
print(myList) # affichage: [1, 2]
del myList # supprimer la liste
```

- vous pouvez facilement échanger les éléments de la liste pour inverser leur ordre ex : variable1, variable2 = variable2, variable1

Utilisation des boucles pour une liste



Partie
1

Partie
2

Partie
3

- Les listes peuvent être itérées en utilisant la boucle for, Ex:

```
myList = ["blanc", "rouge", "bleu", "jaune", "vert"]  
for couleur in myList:  
    print(couleur)
```

- La fonction len() peut être utilisée pour vérifier la longueur de la liste , Ex :

```
myList = ["blanc", "rouge", "bleu", "jaune", "vert"]  
print(len(myList)) # outputs 5  
del myList[2]  
print(len(myList)) # outputs 4
```


Manipulation des méthodes pour une liste



- Une typique fonction ressemble comme suit invocation: `result = function(arg)`, tandis qu'un typique méthode ressemble invocation comme ceci: `result = data.method(arg)`.
- La méthode `append()` elle prend la valeur de son argument et le place à la fin de la liste qui possède la méthode. Ex: `myList.append(333)`
- La méthode `insert()` elle peut ajouter un nouvel élément à n'importe quel endroit de la liste , pas seulement à la fin. Ex: `myList.insert(1, 333)`

Utilisation de Tri pour une liste simple 1



- De nombreux algorithmes de tri ont été inventés jusqu'à présent, qui diffèrent beaucoup en termes de vitesse et de complexité. Nous allons vous montrer un algorithme (trie par bulles) très simple, facile à comprendre, mais malheureusement pas trop efficace non plus.

```
myList = [8, 10, 6, 2, 4] # list to sort
echange = True # Nous en avons besoin pour entrer dans la boucle while
while échange:
    échange = False # aucun échange pour l'instant
    for i in range(len(myList) - 1):
        if myList[i] > myList[i + 1]:
            échange = True # l'échange s'est produit!
            myList[i], myList[i + 1] = myList[i + 1], myList[i]
print(myList)
```

0	1	2	3	4	5	6	7	8	9
68	50	23	7	72	-84	35	9	15	44
50	68	23	7	72	-84	35	9	15	44
50	23	68	7	72	-84	35	9	15	44
50	23	7	68	72	-84	35	9	15	44
50	23	7	68	72	-84	35	9	15	44
50	23	7	68	-84	72	35	9	15	44
50	23	7	68	-84	35	72	9	15	44
50	23	7	68	-84	35	9	72	15	44
50	23	7	68	-84	35	9	15	72	44
50	23	7	68	-84	35	9	15	44	72
23	7	50	-84	35	9	15	44	68	72
7	23	-84	35	9	15	44	50	68	72
7	-84	23	9	15	35	44	50	68	72
-84	7	9	15	23	35	44	50	68	72
-84	7	9	15	23	35	44	50	68	72

Utilisation de Tri pour une liste simple 2



Partie
1

Partie
2

Partie
3

- Vous pouvez utiliser la méthode `sort()` pour trier les éléments d'une liste,
- Il existe également une méthode de liste appelée `reverse()`, que vous pouvez utiliser pour inverser la liste,

Exemple # `sort()`

```
lst = [5, 3, 1, 2, 4]
```

```
print(lst)
```

```
lst.sort()
```

```
print(lst) # résultat : [1, 2, 3, 4, 5]
```

Exemple # `reverse()`

```
lst = [5, 3, 1, 2, 4]
```

```
print(lst)
```

```
lst.reverse()
```

```
print(lst) # résultat : [4, 2, 1, 3, 5]
```

Utilisation des tranches pour une liste 1



- Vous pouvez également utiliser des indices négatifs pour effectuer des tranches. Exemple 1:
- Les paramètres start et end sont facultatifs lors de l'exécution d'une tranche: `list[start:end]` exemple 2:

Exemple # 1

```
sampleList = ["A", "B", "C", "D",  
              "E"]  
newList = sampleList[2:-1]  
print(newList) # outputs: ['C', 'D']
```

Exemple # 2

```
myList = [1, 2, 3, 4, 5]  
sliceOne = myList[2: ]  
sliceTwo = myList[ :2]  
sliceThree = myList[-2: ]  
print(sliceOne) # outputs: [3, 4, 5]  
print(sliceTwo) # outputs: [1, 2]  
print(sliceThree) # outputs: [4, 5]
```

Utilisation des tranches pour une liste 2



- Vous pouvez supprimer des tranches à l'aide de l'instruction `del` Ex 1:
- Vous pouvez tester si certains éléments existent dans une liste ou ne pas utiliser les mots `in`-clés et `not in`, exemple 2:

Exemple # 1

```
myList = [1, 2, 3, 4, 5]
del myList[0:2]
print(myList) # resultat: [3, 4, 5]
del myList[:]
print(myList) # supprime le contenu, affiche: []
```

Exemple # 2

```
myList = ["A", "B", 1, 2]
print("A" in myList) # resultat: True
print("C" not in myList) # resultat: True
print(2 not in myList) # resultat: False
```

Utilisation de compréhension des listes



- La compréhension des listes vous permet de créer de nouvelles listes à partir des listes existantes de manière élégante. La syntaxe d'une compréhension de liste se présente comme suit:

```
# syntaxe manière élégante  
[expression for element in list if conditional]
```

Exemple # manière élégante

```
imp= [x for x in range(10) if x % 2 != 0 ]
```

```
cubed = [num ** 3 for num in range(5)]
```

```
print(cubed) # résultat : [0, 1, 8, 27, 64]
```

```
#syntaxe méthode simple
```

```
for element in list:
```

```
    if conditional:
```

```
        expression
```

```
imp=[]
```

```
for x in range(10):
```

```
    if x%2!=0:
```

```
        imp.append(x)
```

```
# Exemple méthode simple
```

```
ligne= []
```

```
for i in range(8):
```

```
    ligne.append(0)
```

```
#résultat [0, 0, 0, 0, 0, 0, 0, 0]
```

Utilisation des tableaux bidimensionnels



- si nous voulons créer une liste de listes représentant l'ensemble d'un tableau à deux dimensions(matrice), cela peut être fait de la manière suivante:
- L'accès au champ sélectionné du tableau nécessite deux indices - le premier sélectionne la ligne; le second - le numéro de champ à l'intérieur de la ligne, qui est de facto un numéro de colonne ex: ajouter le texte "ok" à C4 ? `tableau[3][2] = 'ok'`.

```
# Exemple méthode simple
tableau= []
for i in range(8):
    ligne= [0 for i in range(8)]
    tableau.append(ligne)
```

```
#Exemple manière élégante
tableau= [[0 for i in range(8)] for j in range(8)]
```

Utilisation des listes bidimensionnelle (Tableaux)



Partie
1

Partie
2

Partie
3

- Vous pouvez utiliser des listes imbriquées en Python pour créer des matrices (c'est-à-dire des listes bidimensionnelles). Par exemple:

Y: 0	:	(:)	:	(:)
1	:)	:	(:)	:)
2	:	(:)	:	(:)
3	:)	:	(:)	:	(
X:	0	1	2	3				

```
# Un tableau à 4 colonnes / 4 lignes  
#un tableau à deux dimensions (4x4)
```

```
table = [":(", ":)", ":(", ":)",  
          ":)", ":(", ":)", ":)",  
          ":(", ":)", ":)", ":(",  
          ":)", ":)", ":)", ":("]
```

```
print(table)  
print(table[0][0]) # resultat: ':('  
print(table[0][3]) # resultat: ':)'
```


Utilisation des Listes bidimensionnelle dans les applications avancées



- Une appareil enregistre la température de l'air sur une base horaire et le fait tout au long du mois. cela vous donne un total de $24 \times 31 = 744$ valeurs. Essayons de concevoir une liste capable de stocker tous ces résultats.
- déterminer la température mensuelle moyenne à midi. vous pouvez supposer que la température de minuit est enregistrée en premier ex 1.
- la température la plus élevée de tout le mois ex 2:

#Exemple 1

```
temps = [[0.0 for h in range(24)] for d in range(31)]
```

```
total = 0.0
```

```
for jour in temps:
```

```
    total += jour[11]
```

```
moyenne= total / 31
```

```
print("la température moyenne est:", moyenne)
```

#Exemple 2

```
temps = [[0.0 for h in range(24)] for d in range(31)]
```

```
maxT= -100.0
```

```
for jour in temps:
```

```
    for temp in jour:
```

```
        if temp > maxT:
```

```
            maxT= temp
```

```
print("La température la plus élevée était: ", maxT)
```

Utilisation des tableaux tridimensionnels

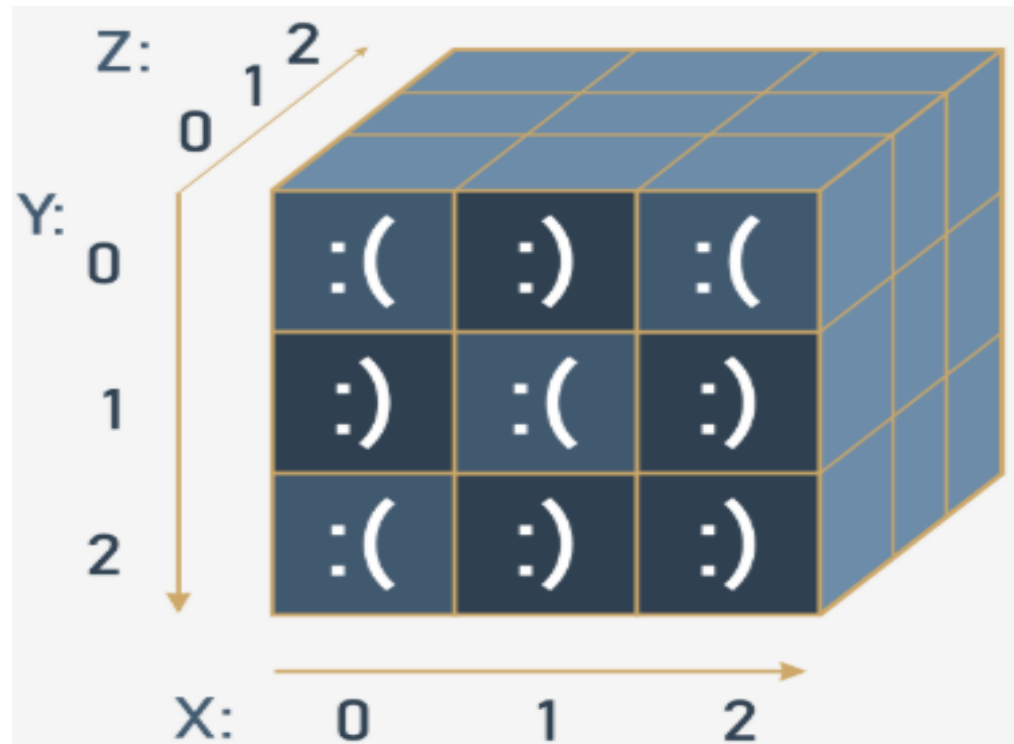


Partie
1

Partie
2

Partie
3

- Vous pouvez utiliser des listes imbriquées en Python pour créer des matrices (c'est-à-dire des listes bidimensionnelles). Par exemple:



Cube - un tableau tridimensionnel (3x3x3)

```
cube = [[['(', 'x', 'x'],  
         [':)', 'x', 'x'],  
         [':(', 'x', 'x']],
```

```
        [[':)', 'x', 'x'],  
         [':(', 'x', 'x'],  
         [':)', 'x', 'x']],
```

```
        [[':(', 'x', 'x'],  
         [':)', 'x', 'x'],  
         [':)', 'x', 'x']]]
```

```
print(cube)
```

```
print(cube[0][0][0]) # outputs: ':('
```

```
print(cube[2][2][0]) # outputs: ':)'
```

Utilisation des listes tridimensionnelles dans les applications avancées



- Un hôtel composé de trois bâtiments de 15 étages chacun. Il y a 20 chambres à chaque étage. Pour cela, vous avez besoin d'un tableau qui peut collecter et traiter des informations sur les chambres occupées / libres.
- réserver une chambre pour deux jeunes mariés: dans le deuxième bâtiment, au dixième étage, chambre 14 Ex2:
- libérer la deuxième chambre au cinquième étage située dans le premier bâtiment Ex2 :
- Vérifiez s'il y a des chambres libre au 15ème étage du troisième bâtiment Ex3 :

#Exemple 1

```
rooms = [[[False for r in range(20)] for f in range(15)] for t in range(3)]
```

#Exemple 2

```
rooms[1][9][13] = True
```

```
rooms[0][4][1] = False
```

#Exemple 3

```
libre= 0
```

```
for roomNumber in range(20):
```

```
    if not rooms[2][14][roomNumber]:
```

```
        libre+= 1
```

Définition d'un tuple



- Les tuples sont des collections de données ordonnées et immuables (inchangeable). Ils peuvent être considérés comme des listes immuables. Ils sont écrits entre parenthèses:
- Chaque élément de tuple peut être d'un type différent (c'est-à-dire des entiers, des chaînes, des booléens, etc.). De plus, les tuples peuvent contenir d'autres tuples ou listes (et inversement).

Exemple # Tuple

```
myTuple = (1, 2, True, "a string", (3, 4), [5, 6], None)
print(myTuple)

myList = [1, 2, True, "a string", (3, 4), [5, 6], None]
print(myList)
```

Création d'un tuple



- Vous pouvez créer un tuple vide comme ceci Ex1:
- Un tuple à un élément peut être créé comme (Ex2) mais si vous supprimez la virgule, vous direz à Python de créer une variable, pas un tuple Ex3:
- Vous pouvez accéder aux éléments de tuple en les indexant Ex4:

Exemple # 1

```
emptyTuple = ()  
print(type(emptyTuple))  
# résultat: <class 'tuple'>
```

Exemple # 2

```
# parenthèse et virgule  
oneElemTup1 = ("one", )  
  
# juste une virgule  
oneElemTup2 = "one",
```

Exemple # 3

```
myTup1 = 1,  
print(type(myTup1))  
# résultat : <class 'tuple'>  
  
myTup2 = 1  
print(type(myTup2))  
# résultat : <class 'int'>
```

Exemple # 4

```
myTuple = (1, 2.0, "string", [3, 4], (5, ),  
True)  
print(myTuple[3]) # résultat : [3, 4]
```

suppression d'un tuple



- Les tuples sont immuables, ce qui signifie que vous ne pouvez pas modifier leurs éléments (vous ne pouvez pas ajouter de tuples, ni modifier ou supprimer des éléments de tuple). L'extrait de code suivant provoquera une exception Ex1:
- Cependant, vous pouvez supprimer un tuple dans son ensemble Ex2:

Exemple # 1

```
myTuple = (1, 2.0, "string", [3, 4], (5, ), True)
myTuple[2] = "guitar" # une exception TypeError sera
levée
```

Exemple # 2

```
myTuple = 1, 2, 3,
del myTuple
print(myTuple) #NameError: le nom 'myTuple' n'est pas
défini
```

parcourir un tuple



Partie
1

Partie
2

Partie
3

- Vous pouvez parcourir un élément d'un tuple (exemple 1), vérifier si un élément spécifique est (pas) présent dans un tuple (exemple 2), utiliser la fonction `len()` pour vérifier combien d'éléments il y a dans un tuple (exemple 3), ou même joindre / multiplier des tuples (exemple 4):

Exemple # 1	Exemple # 2	Exemple # 3	Exemple # 4
			<code>t1 = (1, 2, 3)</code>
<code>t1 = (1, 2, 3)</code>	<code>t2 = (1, 2, 3, 4)</code>	<code>t3 = (1, 2, 3, 5)</code>	<code>t2 = (1, 2, 3, 4)</code>
<code>for elem in t1:</code>	<code>print(5 in t2)</code>	<code>print(len(t3))</code>	<code>t3 = (1, 2, 3, 5)</code>
<code> print(elem)</code>	<code>print(5 not in t2)</code>		<code>t4 = t1 + t2</code>
			<code>t5 = t3 * 2</code>
			<code>print(t4) #(1, 2, 3, 1, 2, 3, 4)</code>
			<code>print(t5) # (1, 2, 3, 5, 1, 2, 3, 5)</code>

Utilisation des fonctions intégrée tuple() et list()



Partie
1

Partie
2

Partie
3

- Vous pouvez également créer un tuple à l'aide d'une fonction intégrée Python appelée tuple(). Ceci est particulièrement utile lorsque vous souhaitez convertir un certain élément itérable (par exemple, une liste, une plage, une chaîne, etc.) en un tuple Ex1:
- De la même manière, lorsque vous souhaitez convertir un élément itérable en liste, vous pouvez utiliser une fonction intégrée Python appelée list() Ex2:

Exemple # 1

```
myTup = tuple((1, 2, "string"))
print(myTup)
lst = [2, 4, 6]
print(lst) # résultat : [2, 4, 6]
print(type(lst)) # résultat : <class 'list'>
tup = tuple(lst)
print(tup) # résultat : (2, 4, 6)
print(type(tup)) # résultat : <class  
'tuple'>
```

Exemple # 2

```
tup = 1, 2, 3,
lst = list(tup)
print(type(lst)) # résultat : <class  
'list'>
```




CHAPITRE 2

Maitriser les ensembles et les dictionnaires

1. Mettre en place un environnement Python
2. Appliquer les bases de la programmation en Python
- 3. Appliquer les techniques de programmation avancée en utilisant Python**

Définition des Ensembles

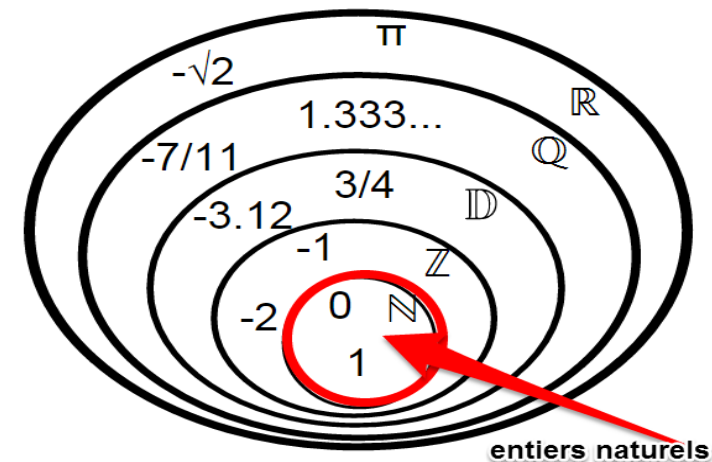


Partie
1

Partie
2

Partie
3

- Un ensemble est une collection. Contrairement aux séquences (liste, tuple...), les éléments d'un ensemble ne sont pas ordonnés et il ne contient pas d'éléments dupliqués.
- Des utilisations basiques concernent par exemple des tests d'appartenance ou des suppressions de doublons.
- Les ensembles savent également effectuer les opérations mathématiques telles que les unions, intersections, différences et différences symétriques, etc.



Création d'un ensemble 1



- Des accolades ou la fonction `set()` peuvent être utilisés pour créer des ensembles.
- Notez que pour créer un ensemble vide, `{}` ne fonctionne pas, cela crée un dictionnaire vide. Utilisez plutôt `set()`.
- Tout comme pour les compréhensions de listes, il est possible d'écrire des compréhensions d'ensembles.
- Il peut avoir n'importe quel nombre d'éléments et ils peuvent être de types différents (int, float, tuple, string, etc.). Mais un ensemble ne peut pas avoir un élément mutable, comme une liste, un ensemble ou un dictionnaire.
- Plusieurs opérations et fonctions applicables aux séquences sont également applicables aux ensembles.
- On peut notamment connaître la taille d'un ensemble avec la fonction `len` et tester si un élément en fait partie ou non avec l'opérateur `in`.

Création d'un ensemble 2



Partie
1

Partie
2

Partie
3

- On ne peut pas accéder aux éléments d'un ensemble à partir d'un indice, comme avec les séquences, puisqu'il s'agit d'une collection non ordonnée d'éléments.
- On ne peut donc parcourir les éléments d'un ensemble qu'avec la boucle for.
- on peut également créer un ensemble à partir d'une séquence. On utilise pour cela la fonction set en lui passant comme paramètre la séquence sur base de laquelle il faut créer l'ensemble. Ex: `A = set([12, 42, 0, 12, 0, -1, 0])`

Exemple 1

```
basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
```

```
print(basket) # {'orange', 'banana', 'pear', 'apple'}
```

```
print('orange' in basket) # True
```

```
print('blue' in basket) # False
```

```
print(len(basket)) # 4
```

```
print(type(basket)) # <class 'set'>
```

```
for element in basket:
```

```
    print(element)
```

Exemple 2

```
a = set('abracadabra')
```

```
b = set('alacazam')
```

```
print(a) # {'r', 'b', 'd', 'c', 'a'}
```

```
print(b) # {'m', 'l', 'z', 'c', 'a'}
```

$$S = \left\{ n \in \mathbb{N} \text{ avec } 0 \leq n < 100 \mid n \text{ est divisible par 3 et 7} \right\}$$

Exemple 3

```
S = {n for n in range(100) if n % 3 == 0 and n % 7 == 0}
```

```
print(S) # {0, 42, 84, 21, 63}
```

Ajout des éléments dans un ensemble 1



- Les ensembles sont mutables. Mais comme ils ne sont pas ordonnés, l'indexation n'a pas de sens.
- Nous ne pouvons pas accéder à un élément d'un ensemble ni le modifier à l'aide de l'indexation ou du découpage en tranches. Set ne le supporte pas.
- Nous pouvons ajouter un seul élément à l'aide de la méthode `add()` et plusieurs éléments à l'aide de la méthode `update()`. La méthode `update()` peut prendre pour argument des tuples, des listes, des chaînes ou d'autres ensembles. Dans tous les cas, les doublons sont évités.

Ajout des éléments dans un ensemble 2



- Les ensembles sont mutables. Mais comme ils ne sont pas ordonnés, l'indexation n'a pas de sens.
- Nous ne pouvons pas accéder à un élément d'un ensemble ni le modifier à l'aide de l'indexation ou du découpage en tranches. Set ne le supporte pas.
- Nous pouvons ajouter un seul élément à l'aide de la méthode `add()` et plusieurs éléments à l'aide de la méthode `update()`. La méthode `update()` peut prendre pour argument des tuples, des listes, des chaînes ou d'autres ensembles. Dans tous les cas, les doublons sont évités.

Suppression des éléments d'un ensemble



- Un élément particulier peut être supprimé de la série à l'aide des méthodes, `discard()` et `remove()`.
- La seule différence entre les deux est que, tout en utilisant `discard()` si l'élément n'existe pas dans l'ensemble, il reste inchangé. Mais `remove()` lève une erreur dans une telle condition.
- De même, nous pouvons supprimer et retourner un élément en utilisant la méthode `pop()`.
- Les ensembles n'étant pas ordonnés, il n'y a aucun moyen de déterminer quel élément sera supprimé. C'est complètement arbitraire.
- Nous pouvons également supprimer tous les éléments d'un ensemble en utilisant `clear()`.

```
# Exemple 4
numbers = {1,2}
numbers.add(3)
numbers.update([50,51],{100,200})
numbers.remove(51)
numbers.discard(2)
print(numbers.pop()) # 1
numbers.clear()
print(numbers)
```

Utilisation des sous-ensemble et sur-ensemble



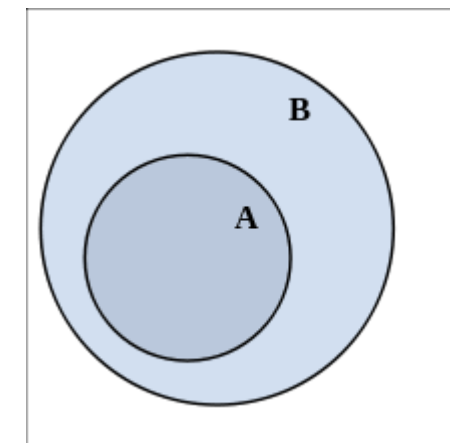
Partie
1

Partie
2

Partie
3

- Soit: $A = \{1, 2, 3, 4, 5, 6, 7, 8, 9, 10\}$ et $B = \{1, 2, 3\}$
 - ✓ L'ensemble B est un sous-ensemble de A, si tous les éléments de l'ensemble B sont également les éléments de l'ensemble A
 - ✓ En d'autres termes, nous pouvons également dire que l'ensemble A est un sur-ensemble de B.
 - ✓ Nous pouvons tester si un ensemble est un sous-ensemble ou un sur-ensemble d'un autre ensemble à l'aide des méthodes `issubset()` et `issuperset()`.

- Ex: `print(B.issubset(A)) #True` | `print(A.issubset(B)) #False`



Manipulation des opération ensembliste 1

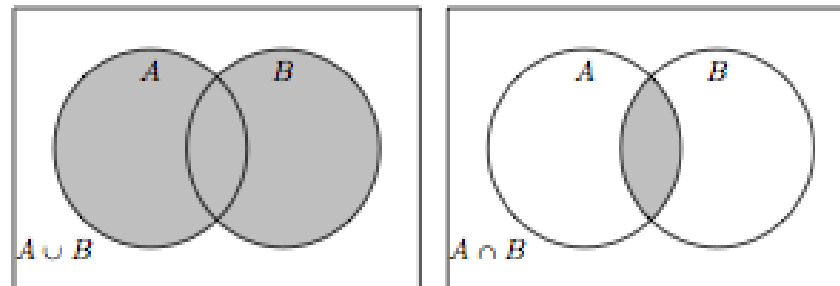


Partie
1

Partie
2

Partie
3

- l'union des ensembles A et B, notée $A \cup B$, contient tous les éléments se trouvant soit dans A, soit dans B (ou dans les deux) ; Ex: $C=A.union(B)$
- l'intersection des ensembles A et B, notée $A \cap B$, contient tous les éléments se trouvant à la fois dans A et dans B. Ex: $C=A.intersection(B)$



Exemple 5

$A = \{1, 2, 3, 4\}$

$B = \{3, 4, 5\}$

`print(A.union(B))` # {1, 2, 3, 4, 5}

`print(A.intersection(B))` # {3, 4}

`print(A | B)` # {1, 2, 3, 4, 5}

`print(A & B)` # {3, 4}

$A \setminus B = \{1, 2\}$

`print(A)` # {1, 2, 3, 4, 5}

$B \setminus A = \{5\}$

`print(B)` # {3, 4, 5}

Manipulation des opération ensembliste 2

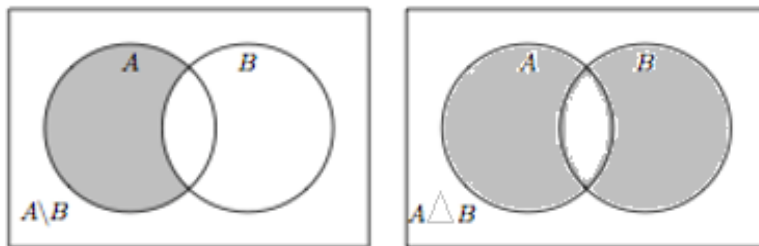


Partie
1

Partie
2

Partie
3

- la différence entre A et B, notée $A \setminus B$, contient tous les éléments de A sauf ceux appartenant également à B ; Ex: `C=A.difference(B)`
- La différence symétrique entre A et B, notée $A \Delta B$, (notée $A \Delta B$) est l'ensemble des éléments appartenant à A ou à B exclusivement, " $A \Delta B = (A \cup B) - (A \cap B)$ ". Ex: `C=A.symmetric_difference(B)`



Exemple 5

```
A = {1, 2, 3, 4}
```

```
B = {3, 4, 5}
```

```
print(A.difference(B)) # {1, 2}
```

```
print(A - B) # {1, 2}
```

```
print(A.symmetric_difference(B)) # {1, 2, 5}
```

```
print(A^B) # {1, 2, 5}
```

```
A -= {3}
```

```
print(A) # {1, 2, 4}
```

```
B ^= {6}
```

```
print(B) # {3, 4, 5, 6}
```

Manipulation des ensembles non modifiable



- Frozenset est une nouvelle classe qui présente les caractéristiques d'un ensemble, mais ses éléments ne peuvent pas être modifiés une fois affectés. Les tuples sont des listes immuables, les frozensets sont des ensembles immuables.
- Les ensembles étant mutables sont unhashable, donc ils ne peuvent pas être utilisés comme des clés de dictionnaires. D'autre part, frozensets sont hashable et peuvent être utilisés comme clés d'un dictionnaire.

```
# Exemple 5
A = frozenset([1, 2, 3, 4])
print(A) # frozenset({1, 2, 3, 4})
A.add(5) # AttributeError
```

Définition de dictionnaire



Partie
1

Partie
2

Partie
3

- Les dictionnaires sont des collections de données non ordonnées *, modifiables (modifiables) et indexées. (* En Python 3.6x, les dictionnaires sont classés par défaut).
- Chaque dictionnaire est un ensemble de paires clé: valeur . Vous pouvez le créer en utilisant la syntaxe suivante:

Exemple # 1

```
myDictionary = {  
    key1 : value1,  
    key2 : value2,  
    key3 : value3,  
}
```

accéder à un élément du dictionnaire



- 2. Si vous souhaitez accéder à un élément du dictionnaire, vous pouvez le faire en faisant référence à sa clé à l'intérieur d'une paire de crochets (ex. 1) ou en utilisant la méthode get() (ex. . 2):

```
polEngDict = {  
    "kwiat" : "flower",  
    "woda" : "water",  
    "gleba" : "soil"  
}  
  
item1 = polEngDict["gleba"] # ex. 1  
print(item1) # résultat : soil  
  
item2 = polEngDict.get("woda")  
print(item2) # résultat : water
```

modification des valeurs d'un dictionnaires



- Si vous souhaitez modifier la valeur associée à une clé spécifique, vous pouvez le faire en vous référant au nom de la clé de l'élément de la manière suivante:

```
polEngDict = {  
    "zamek" : "castle",  
    "woda" : "water",  
    "gleba" : "soil"  
}  
  
polEngDict["zamek"] = "lock"  
item = polEngDict["zamek"]  # résultat: lock
```

Modification d'un dictionnaire



- Pour ajouter ou supprimer une clé (et la valeur associée), utilisez la syntaxe suivante ex1:
- Vous pouvez également insérer un élément dans un dictionnaire à l'aide de la méthode update() et supprimer le dernier élément à l'aide de la méthode popitem(), Ex2:

Exemple # 1

```
myPhonebook = {} # un dictionnaire vide

myPhonebook["Adam"] = 3456 # créer / ajouter une paire
valeur/clé
print(myPhonebook) # résultat : {'Adam': 3456}

del myPhonebook["Adam"]
print(myPhonebook) # résultat : {}
```

Exemple # 2

```
polEngDict = {"kwiat" : "flower"}

polEngDict = update("gleba" : "soil")
print(polEngDict) #résultat : {'kwiat' : 'flower', 'gleba' : 'soil'}

polEngDict.popitem()
print(polEngDict) # résultat : {'kwiat' : 'flower'}
```

parcourir un dictionnaire



Partie
1

Partie
2

Partie
3

- Vous pouvez utiliser la boucle for pour parcourir un dictionnaire, Ex1:
- Si vous souhaitez parcourir les clés et les valeurs d'un dictionnaire, vous pouvez utiliser la méthode items(), Ex2:

Exemple # 1

```
polEngDict = {  
    "zamek" : "castle",  
    "woda" : "water",  
    "gleba" : "soil"  
}  
  
for item in polEngDict:  
    print(item)  # résultat : zamek  
                # woda  
                # gleba
```

Exemple # 2

```
polEngDict = {  
    "zamek" : "castle",  
    "woda" : "water",  
    "gleba" : "soil"  
}  
  
for key, value in polEngDict.items():  
    print("Pol/Eng ->", key, ":", value)
```


Utilisation de la méthode copy()



- Pour vérifier si une clé donnée existe dans un dictionnaire, vous pouvez utiliser le mot – clé in : Ex1
- Pour copier un dictionnaire, utilisez la méthode copy() : Ex2

Exemple # 1

```
polEngDict = {  
    "zamek" : "castle",  
    "woda" : "water",  
    "gleba" : "soil"  
}  
  
if "zamek" in polEngDict:  
    print("Yes")  
else:  
    print("No")
```

Exemple # 2

```
polEngDict = {  
    "zamek" : "castle",  
    "woda" : "water",  
    "gleba" : "soil"  
}  
  
copyDict = polEngDict.copy()
```

suppression d'un élément du dictionnaire



Partie
1

Partie
2

Partie
3

- Vous pouvez utiliser le mot – clé `del` pour supprimer un élément spécifique ou supprimer un dictionnaire. Pour supprimer tous les éléments du dictionnaire, vous devez utiliser la méthode `clear()` :

```
polEngDict = {  
    "zamek" : "castle",  
    "woda" : "water",  
    "gleba" : "soil"  
}  
  
print(len(polEngDict))  # résultat : 3  
del polEngDict["zamek"] # supprimer un élément  
print(len(polEngDict))  # résultat : 2  
polEngDict.clear()     # supprime tous les éléments  
print(len(polEngDict))  # résultat : 0  
del polEngDict         # supprime le dictionnaire
```