ISIMM

Programmation Python 1

CPI1

2026

Plan

01 Introduction

Python est l’un des langages de programmation les plus populaires au monde.

- Simple à utiliser, riche en fonctionnalités et soutenu par un large écosystème de bibliothèques et de frameworks.
- Sa syntaxe claire et épurée le rend particulièrement adapté aux débutants.
- Langage de haut niveau, utilisé en Data science, Automatisation, Intelligence artificielle, Développement web
- Reconnu pour sa lisibilité, ce qui facilite: L’écriture du code, Sa compréhension et Sa maintenance.
- Grâce à son fort support de bibliothèques, il n’est pas nécessaire de tout développer à partir de zéro.

01 Introduction

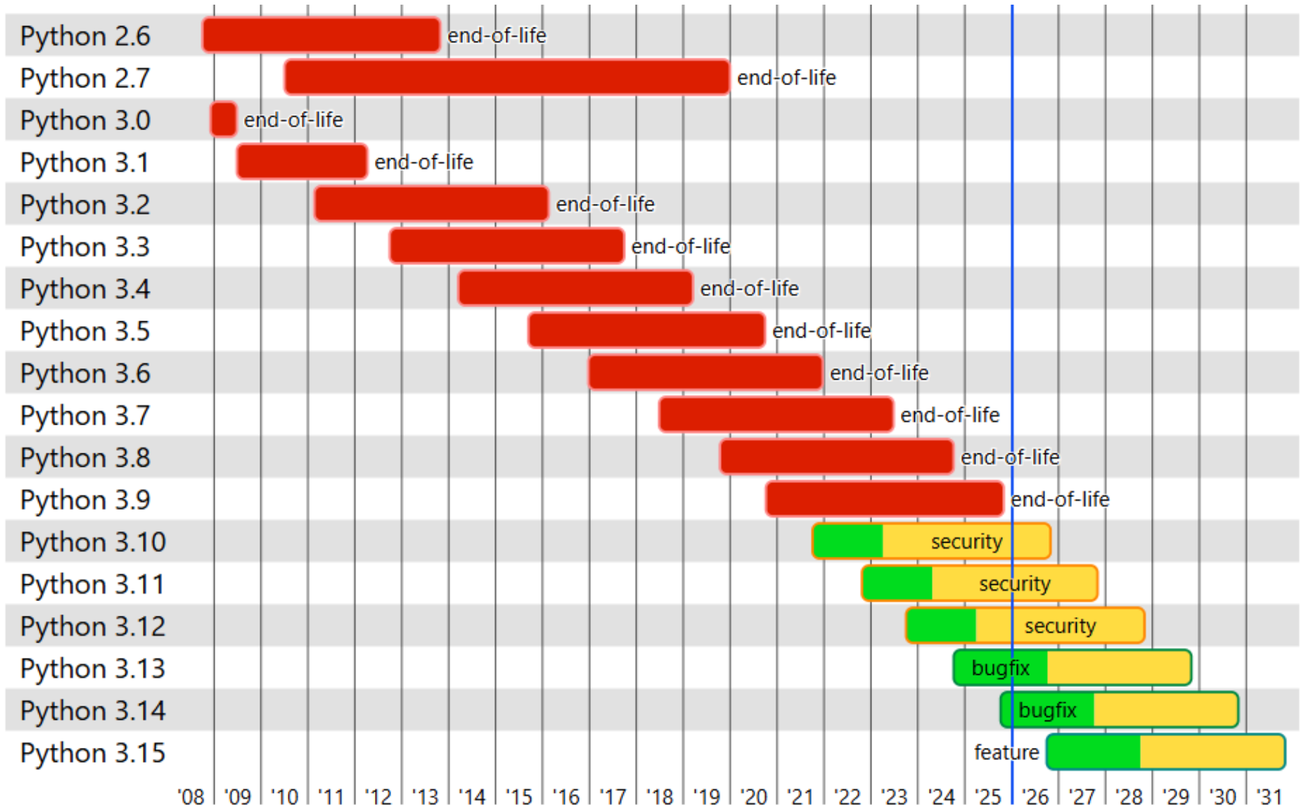
Python est un langage de programmation:

- libre
- multi-plateformes
- multi-paradigmes
- distribué avec une riche bibliothèque standard

01

Introduction
Histoire de Python

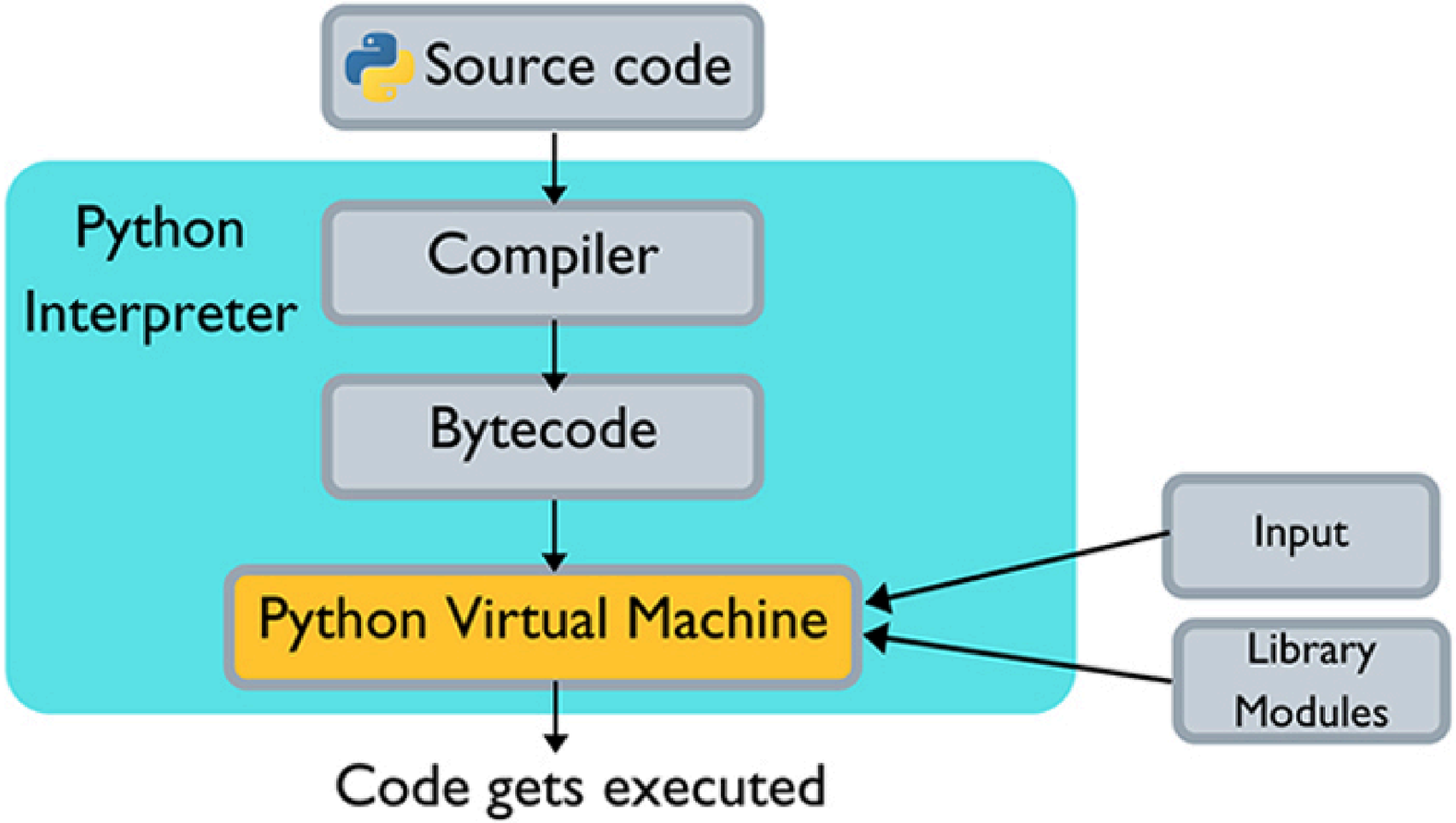
- Python est un langage de programmation qui a été créé en 1989 par Guido van Rossum aux Pays-Bas.
- La première version publique de ce langage a été publiée en 1991.



01

Introduction

Compilation & Interprétation



01

Introduction
Installation

Afin de pouvoir développer en langage Python, il est nécessaire d'installer les outils suivants :

- Télécharger et installer le langage Python depuis le site officiel Python.
- Télécharger et installer un IDE Python : de nombreux choix s’offre à vous : Pycharm, PyScripter, Wing.

02

Les variables
Qu'est-ce qu’une variable ?

- Une variable est définie par un nom dans un langage de programmation, lors que pour l'ordinateur il s'agit d'une référence désignant une adresse mémoire (i.e. un emplacement précis dans la mémoire).
- On attribue une valeur à une variable en utilisant l’opérateur d’affectation (=).
    - Les variables sont utilisées sans être déclarés et leurs types dépenden de leurs contenus : Typage dynamique.
    - Une variable peut changer de type sans qu’il ait d’erreurs.
    - Python est sensible à la casse, ce qui signifie que les variables TesT, test ou TEST sont différentes.

02

Les variables
Les types élémentaires

Python propose 4 types simples de base qui sont :

- Les entiers (integer ou int),
- Les réels (float)
- Les booléens (bool)
- Les nombres complexes (complex)
- Les chaines de caractères (str)

02

Les variables
Les types élémentaires

- Classe **int**: Le type int (entier) n'est pas limité en taille que par la mémoire de la machine.
- Classe **float**: Un float est noté avec un point décimal (jamais avec une virgule) un "e" symbolisant le "10 puissance". Les flottants supportent les mêmes opérations que les entiers. Ils ont une précision infinie limitée.
- Classe **booléenne**: Deux valeurs possibles: False et True.
- Classe **complexe**: Les complexes sont écrits en notation cartésienne formée de deux flottants. La partie imaginaire est suffixée par j.
- Classe **str**: Une chaîne de caractère est une suite de caractères entourée par deux apostrophes $'$ ou deux guillemets $"$. Les éléments d'une chaîne s sont indexés de 0 à len(s)-1, accessibles par l'opérateur [].

02

Les variables
Opérations sur les variables

Python classe les opérateurs selon les groupes suivants:

- Opérateurs logiques
- Opérateurs arithmétiques
- Opérateurs d’affectation
- Raccourcis d’auto-affectation

Conversion de type

Conversion	Fonction	Exemple	Sortie
De l'entier au flottant	`float()`	`float(3)`	`3.0`
De flottant à entier	`int()`	`int(3.9)`	`3`
Chaîne vers nombre entier	`int()`	`int('123')`	`123`
D'un entier à une chaîne de caractères	`str()`	`str(123)`	`'123'`

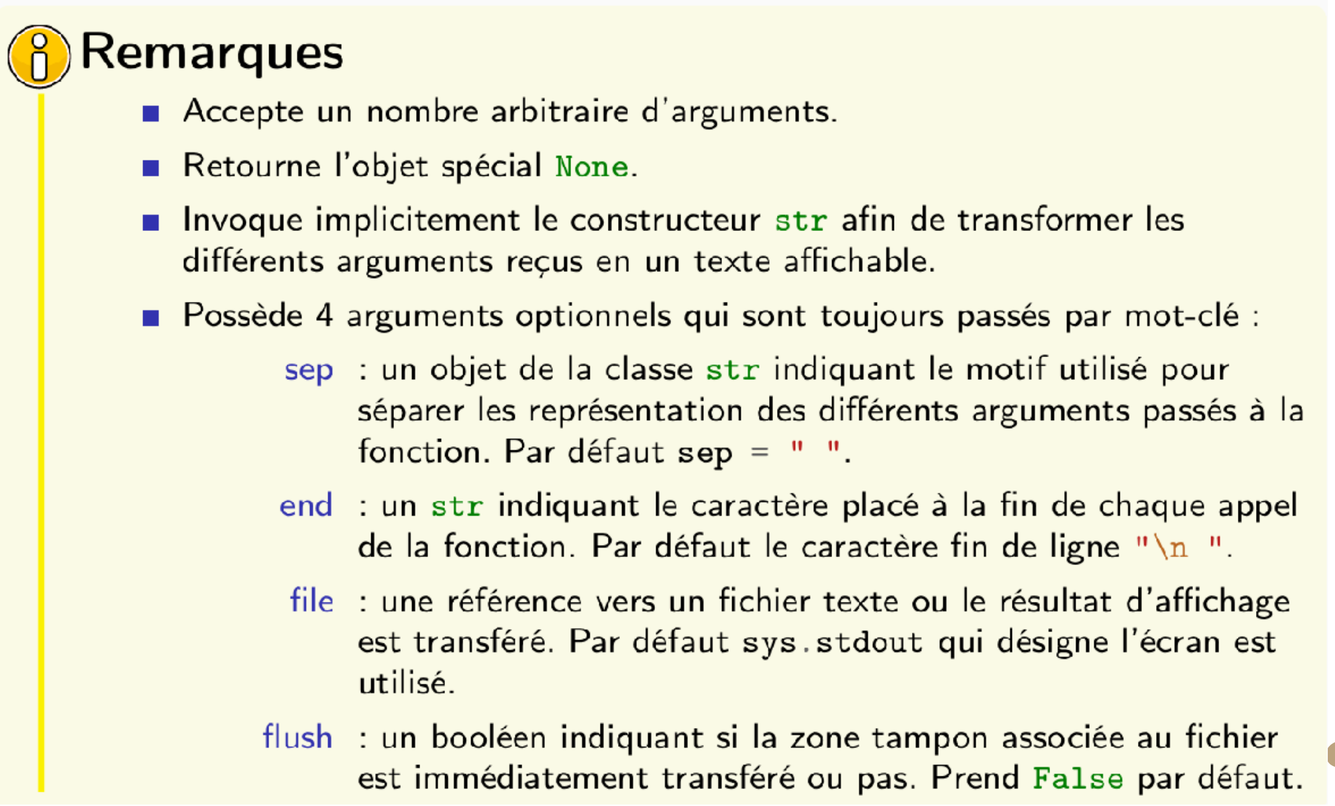
Les fonctions print et input

La fonction print: Permet d'afficher des informations sur la sortie standard, sous forme de texte.

```
>>> x, y = 5, 7
>>> print(" le couple (x,y) = ",(x,y))
 le couple (x,y) =  (5, 7)
>>> for i in range(3):
        print(i)
0
1
2
>>> for i in range(3):
        print(i, end = " ")
0 1 2
>>> print(x,y, sep=",")
5,7
```

Les fonctions print et input

La fonction print: Permet d'afficher des informations sur la sortie standard, sous forme de texte.



Remarques

- Accepte un nombre arbitraire d'arguments.
- Retourne l'objet spécial `None`.
- Invoque implicitement le constructeur `str` afin de transformer les différents arguments reçus en un texte affichable.
- Possède 4 arguments optionnels qui sont toujours passés par mot-clé :

  `sep` : un objet de la classe `str` indiquant le motif utilisé pour séparer les représentation des différents arguments passés à la fonction. Par défaut `sep = " "`.

  `end` : un `str` indiquant le caractère placé à la fin de chaque appel de la fonction. Par défaut le caractère fin de ligne `"\n "`.

  `file` : une référence vers un fichier texte ou le résultat d'affichage est transféré. Par défaut `sys.stdout` qui désigne l'écran est utilisé.

  `flush` : un booléen indiquant si la zone tampon associée au fichier est immédiatement transféré ou pas. Prend `False` par défaut.

Les fonctions print et input

La fonction input: Permet de récupérer un str à partir de l'entrée standard.

```
>>> msg = input()
432
>>> print(msg, type(msg))
432 <class 'str'>
>>> msg = input("donner un message : ")
donner un message : 5.25
>>> msg
'5.25'
>>> float(msg)
5.25
>>> type(_) # _ est un Identificateur spécial, rappelle l'historique de la dernière
↪  opération effectué par l'interpréteur. Défini uniquement pour les sessions
↪  interactives.
<class 'float'>
```

Les fonctions print et input

La fonction input: Permet de récupérer un str à partir de l'entrée standard.

Remarques

- Accepte un seul argument optionnel qui représente le prompt affiché.
- Retourne toujours une instance de la classe `str`.
- Il est possible de convertir l'`str` résultant en une valeur numérique en utilisant les constructeurs `int`, `float` ou `complex`, par exemple pour lire un nombre flottant :

```
x = float(input(" donner x : "))
```

Les conteneurs de bases

Les classes itérables ou conteneurs permettent de conserver une collection d'objets. Ces classes peuvent se ramifier selon deux critères :

- La mutabilité : classes modifiables (list, set, dict) et classes non modifiables (range, str et tuple).
- L'ordre : classes ordonnées appelées encore séquences (str, tuple, list, range) et classes non ordonnées (set et dict).

Description	Syntaxe	Exemples / Remarques
Taille d'un itérable	`len(iter)`	`lst = [1,2,5]` `len(lst)  #donne 3`

Test d'appartenance d'un élément x à un itérable	`x in iter`
	`x not in iter`
		`5 in (1,5,[1,3])  # donne True`
		`3 not in (1,5,[1,3])  # donne True`
		`"li b" in "ali baba"  # donne True`
		`"Ali" in "ali baba"  # donne False`

Somme des éléments d'un itérable (itérable de valeurs numériques uniquement).	`sum(iter)`	`sum([1,5,2,10])  # donne 18`

Valeur maximale/ minimale
TypeError pour un itérable comportant des objets non comparables
	`max(iter)`
	`min(iter)`
		`max([1,5,2,10])  # donne 10`
		`l = (1+3j,2+5j,11j); max(l)# Error`
		`min(1,5,2,10)  #  donne 1`

Les conteneurs de bases

Les classes itérables ou conteneurs permettent de conserver une collection d'objets. Ces classes peuvent se ramifier selon deux critères :

- La mutabilité : classes modifiables (list, set, dict) et classes non modifiables (range, str et tuple).
- L'ordre : classes ordonnées appelées encore séquences (str, tuple, list, range) et classes non ordonnées (set et dict).

Tri d'un itérable d'objets comparables (TypeError si l'itérable comporte un objet non comparable)
Renvoie une liste contenant les éléments dans l'ordre.
	`sorted(iter)`	`d = {'med':10, 'ali':20, 'lilia':5}`
		`sorted(d)  # donne ['ali', 'lilia', 'med']`
		`sorted(d, reverse = True)  # donne l'ordre inverse['med', 'lilia', 'ali']`

Parcours d'un itérable	`for item in iter :`
	`  #opérations`
	`  manipulant item`
		`lst = [1,2,5]`
		`for e in lst:`
		`  print(e)`

Énumération des éléments d'un itérable	`enumerate(iter)`
	`enumerate(iter, start_index)`
		`ch = "cem"`
		`for i, v in enumerate(ch):`
		`  print(i, v)`
		`# donne en sortie`
		`0 c`
		`1 e`
		`2 m`

Les conteneurs de bases

Les itérables ordonnés / Les séquences

- Les itérables ordonnés, appelés encore séquences (tuple, liste et chaîne), conservent l'ordre dans lequel les éléments ont été créés. Ainsi, chaque élément d'une séquence possède un indice indiquant sa position relative par rapport aux autres éléments. Les indices commencent à partir de 0 pour un parcours de la gauche vers la droite. Un intervalle défini par **range**, également considéré comme une séquence, est un itérable ou conteneur formé par une succession d'entiers. **range(debut, fin, pas)** génère l'intervalle **[debut, fin [** par pas entier relatif égal à pas.

Les conteneurs de bases

Les itérables ordonnés / Les séquences

Description	Syntaxe	Exemples / Remarques

Indexage simple	`s[i]`
	avec $0 \leq i \leq len(s) - 1$ ou
	$-len(s) \leq i \leq -1$
		`s = 'python'`
		`s[0]` `#donne 'p'`
		`s[len(s)-1]` `#ou s[-1] donne'n'`
		`s[len(s)]` `# IndexError`

Morcelage (Slicing)	`# fin non inclus`
	`s[dep:fin:pas]`
	`#pas par défaut égal à 1`
	`s[dep:fin]`
	`#du début avec pas=1`
	`s[ :fin]`
	`#jusqu'à la fin avec pas=1`
	`s[dep :]`
		`s = 'python'`
		`s[ ::-1]` `# donne 'nohtyp'`
		`s[ :2]` `# donne 'py'`
		`s[2:]` `# donne 'thon'`
		`s[::2]` `# donne 'pto'`

Concaténation	`s1 + s2`	`[1,2] + [3]` `# donne [1,2,3]`
		`(1,2) + (3,)` `# donne (1,2,3)`
		`'med' + 'ali'# donne 'medali'`

Les conteneurs de bases

Les itérables ordonnés / Les séquences

Répétition	`n * s  # ou bien s * n`
	`avec n entier Naturel`
		`[0]  * 3  # donne [0,0,0]`
		`(1,2) * 2  # donne (1,2,1,2)`
		`'bon' * 2  # donne 'bonbon'`

Comptage du nombre d'occurrences	`s.count(item)`	`lst = [0,1,0,1,1,1]`
		`lst.count(1)  # donne 4`
		`ch ="ali baba"`
		`ch.count("a")  # donne 3`

Retour de l'index d'un item
Déclenche ValueError si item not in s.
	`# première occurrence`
	`s.index(item)`
	`# à partir de start`
	`s.index(item, start)`
		`lst.index(0)  # donne 0`
		`lst.index(0,1)# donne 2`
		`lst.index(5)  # ValueError`

Parcours d'un itérable indexé	`for ind in range(len(iter)):`
	`  #opérations manipulant iter[ind]`
		`lst = [1,2,5]`
		`for i in range(len(lst)):`
		`  print(lst[i])`

Les conteneurs de bases

Les itérables ordonnés / Les séquences

- La classe str: Une chaîne est une séquence non modifiable et ordonnée de caractères. Le tableau suivant résume les principales opérations applicables aux chaînes.

Description	Syntaxe	Exemples / Remarques

Création d'une chaîne de caractères	#chaîne simple:
	("…" ,'…')
	#chaîne multilignes:
	(""" … """ ,''' …''')
		s = "bonsoir" # ou encore
		s = 'bonsoir'
		s1="""chaîne
		multi-
		lignes"""
		#équivalente à
		s1 = 'chaîne\nmulti-\nlignes'

Retour d'une chaîne en minuscule identique à Ch	Ch.**lower** ( )	Ch='fG15Z' Ch.lower( ) # donne 'fg15z'
Retour d'une chaîne en majuscule identique à Ch	Ch.**upper** ( )	Ch='fG15Z' Ch.upper ( ) # 'FG15Z'

Les conteneurs de bases

Les itérables ordonnés / Les séquences

Retour d'une liste des sous-chaînes en utilisant un séparateur sep
Par défaut, le séparateur est le caractère espace
	`Ch.split(sep)`	`Ch='Epsilon 2018/2019'`
		`Ch.split( )  # donne ['Epsilon', '2018/2019']`
		`Ch.split('0') # donne [Epsilon 2', '18/2', '19']`

Retour d'une chaîne identique à Ch en supprimant les occurrences d'un séparateur *sep* du début et de la fin si elles existent. Par défaut sep est l'espace.	`Ch.strip(sep)`	`Ch="*la chaîne*"`
		`Ch.strip('*')`
		`# donne 'la chaîne'`

Retour d'une chaîne de caractères résultante de la concaténation par sep des chaînes de l'itérable
TypeError est généré si l'itérable contient un objet qui n'est pas de type str.
	`Sep.join(itérable)`	`L=['abc','12','azer']`
		`'/'.join(L)`
		`# donne 'abc/12/azer'`
		`'  '.join(L)`
		`# donne 'abc 12 azer'`

Les conteneurs de bases

Les itérables ordonnés / Les séquences

Retour d'une chaîne de caractères en remplaçant chaque '{}' par les paramètres passés à la méthode (objets de classes quelconques).	`Ch.format(paramètres)`	`Ch = 'Epsilon {}/{}'`
		`Ch.format(2019,2020)`
		`# donne 'Epsilon 2019/2020'`

Retour de l'Unicode (entier) du caractère passé en paramètre	`ord(caractère)`	`ord ('A') # 65` `ord('a') # 97`
Retour du caractère associé à la valeur UNICODE passée paramètre.	`chr(entier)`	`chr(65) # donne 'A'` `chr(10) # donne '\n'`

Les conteneurs de bases

Les itérables ordonnés / Les séquences

- La classe list: Une liste est une séquence, ordonnée et modifiable, d'éléments éventuellement hétérogènes séparés par une virgule et délimitée par une paire de crochets.

Description	Syntaxe	Exemples
Création d'une liste vide	`L=[ ] #ou` `L=list( )`	`L=[1, 'python', True, 1+1j]`

Création d'une liste à partir d'un itérable	`L=list(itérable)`	`list('ABCA')  # donne ['A','B','C','A']`
		`list({3, 'A',4}) # donne  [3,4,'A']`
		`list((2,2,(1,2)))# donne  [ 2,2,(1,2)]`
		`list({2 :45, 'x':[1,2]}# donne [2,'x']`
		`list(145)# TypeError`

Ajout d'un objet à la fin d'une liste	`L.append (objet)`	`L.append(2020) ; L`
		`#donne [1, 'python', True, (1+1j), 2020]`
		`#remplissage d'une liste par n chaînes`
		`lst = list()`
		`for i in range(n):`
		`  x = input("donner l[{}]:".format(i))`
		`  lst.append(x)`

Les conteneurs de bases

Les itérables ordonnés / Les séquences

Ajout d'un objet x dans L à l'indice i	`L.insert(i,x)`	`L.insert(-1 ,'concours') ; L  # donne` `[1,'python',True,(1+1j),'concours',2020]`
Suppression de la première occurrence d'un l'objet x dans L ValueError si x not in L.	`L.remove(x)`	`L.remove(1+1j) ; L  # donne` `[1, 'python', True, 'concours', 2020]`
Ajout d'éléments de l'itérable t à la fin de L	`L.extend(t)`	`L.extend({-1,2}) ; L  # donne` `[1,'python',True,'concours',2020,2,-1]`

Suppression et retour du dernier élément de L	`L.pop()`	`L.pop()  # donne  -1`
		`L # donne`
		`[1,'python',True,'concours',2020,2]`

Suppression et retour de l'élément d'indice i de la liste L	`L.pop(i)`	`L.pop(2)  # donne  True`
		`L  # donne`
		`[1, 'python', 'concours', 2020, 2]`

Les conteneurs de bases

Les itérables ordonnés / Les séquences

- La classe tuple: Un tuple est une séquence, ordonnée et non modifiable, d'éléments éventuellement hétérogènes séparés par une virgule.

Description	Syntaxe	Exemples
Création d'un tuple vide	`t=( )  #t=tuple( )`	

Création d'un tuple d'un seul objet (singleton)	`t=objet ,`	`T=255,`
		`T  # donne (255,)`
		`T1=[12,5],1`
		`T1  #donne ([12,5], 1)`

Création d'un tuple à partir d'un itérable	`L=tuple(itérable)`	`tuple({3,'A',4})  # donne (3,'A',4)` `tuple((2,2,(1,2)))  # donne (2,2,(1,2))`

Les conteneurs de bases

Les itérables non ordonnés

- La classe set: Un ensemble est une collection d'éléments distincts non ordonnés et encadrés par des accolades. La classe set représente la notion d'ensemble en mathématique. Un ensemble doit impérativement comporter des éléments non modifiables. La création d'ensembles de listes, de dictionnaires ou d'ensembles déclenche une erreur.

Description	Syntaxe	Exemples/ Remarques
Création d'un ensemble vide	`s = set()` `NB : s={} ne crée pas un ensemble`	

Création d'ensemble à partir d'un itérable iter ou avec les délimiteurs	`s = set(iter)`
	`s={e0, …,en}`
		`s = set("010101")`
		`s  #  donne {'0','1'}`
		`s = set(3.25)  # TypeError`

Ajout d'un objet	`s.add(obj)`	`s = set()`
		`s.add(5)  ; s  # donne  {5}`
		`s.add("eps"); s #donne{5,"eps"}`
		`s.add([1,2,3])  #TypeError`

Les conteneurs de bases

Les itérables non ordonnés

Suppression d'un objet	`s.discard(x)`
	`s.remove(x)`
	`s-= {obj}`
		`s = {1,7,2} ;  s.remove(1)`
		`s  # donne {2,7}`
		`s.remove(5)  # KeyError`
		`s -= {2} ; s  # donne {7}`

Union : s1 U s2	`s1 | s2` `s1.union(s2)`	`s = {1,2} | {2,3}` `s  # donne {1,2,3}`
Intersection : s1 ∩ s2	`s1 & s2` `s1.intersection(s2)`	`s = {1,2} & {2,3}` `s  # donne {2}`
Différence : s1 \ s2	`s1 - s2` `s1.difference(s2)`	`s = {1,2} - {2,3}` `s  # donne {1}`
Différence Symétrique s1 Δ s2	`s1 ^ s2` `s1.symmetric_difference(s2)`	`s = {1,2}^{2,3}` `s  # donne {1,3}`
Inclusion : s1 ⊂ s2	`s1 <= s2`	`{1,2} <= {1,2,5}# donne True` `{1,2} <= {1,2}  # donne True`
Inclusion stricte : s1 ⊊ s2	`s1 < s2` `s1.issuperset(s2)`	`{1,2} < {1,2,5} # donne True` `{1,2} < {1,2}  # donne False`

Les conteneurs de bases

Les itérables non ordonnés

- La classe dict: Un dictionnaire est une structure modifiable et non ordonnée d'associations (clé : valeur). Les clés sont impérativement des objets non modifiables. L'utilisation de listes, de dictionnaires ou d'ensembles comme clé déclenche une erreur. Les valeurs sont des objets de classes quelconques.

Description	Syntaxe	Exemples / Remarques
Création d'un dictionnaire vide	`d = dict()` `d = {}`	

Création d'un dictionnaire à partir d'un itérable de couples ou directement	`d= dict(iter)`
	`d={c1:v1 , …, cn:vn}`
		`t = [(5,2), "ab", [1,2] ]`
		`dict(t) #donne {1:2, 5:2, 'a':'b'}`
		`d = {'a': 3, 'b': 5, 4j: 33}`

Test d'appartenance d'une clé k à un dictionnaire d	`k in d` `k not in d`	`33 in d  # donne False` `'a' in d  # donne True`

Accès à une entrée du dictionnaire d	`d[k]`
	`d.get(k)`
	`d.get(k, vdef)`
		`d = {'a': 3, 'b': 5, 4j: 33}`
		`d['a']  # donne 3`
		`d['c']  # donne KeyError`
		`d.get('c')  # None`
		`d.get('c', 0)  #0`

Les conteneurs de bases

Les itérables non ordonnés

Mise à jour / insertion d'une entrée dans le dictionnaire d	`d[k] = val`
	`d.update({k :val})`
		`d = {}`
		`d['a'] = 33  # insertion`
		`d  # donne {'a' :33}`
		`d['a'] = 22  # mise à jour`
		`d  # donne {'a':22}`

Copie de dictionnaire	`d.copy()`	`d = {1:2}`
		`d1 = d.copy()`
		`d[1] = {2} ;  d1  # donne {1:{2}}`
		`d  # donne {1:2}`

Suppression d'une entrée de à partir de sa clé	`d.pop(k)`
	`del d[k]`
		`d = {1:2, 'a':10}`
		`del d[1]; d  # donne {'a':10}`
		`d.pop('a')  # donne la valeur 10`
		`d  # donne  {}`

Parcours des clés d'un dictionnaire	`for k in d:`
	`  traitements`
	`#ou encore`
	`for k in d.keys()`
	`  traitements`
		`d = {1:2, 'a':10}`
		`for k in d:`
		`  print(k)`

Parcours des valeurs d'un dictionnaire	`for v in d.values():` `  traitements`	`for v in d.values():` `  print(v)`
Parcours des couples clé, valeur d'un dictionnaire	`for k, v in d.items():` `  traitement`	`for k, v in d.items():` `  print(k,v)`

Les conteneurs de bases

Construction des itérables en compréhension

- La construction des itérables en compréhension permet de générer des conteneurs de façon concise et élégante. On peut créer des listes en compréhension, des tuples en compréhension, des ensembles en compréhension et des dictionnaires en compréhension à l'aide du constructeur de la classe ou les délimiteurs des itérables mutables.

1. Listes en compréhension:

   [expression for indice_1 in iterable_1 if condition_1 ... for indice_n in iterable_n if condition_n]

   ou encore

   list(expression for indice_1 in iterable_1 if condition_1 ... for indice_n in iterable_n if condition_n)

Les conteneurs de bases

Construction des itérables en compréhension

- Les conditions if sont facultatives (conditions de filtrage).
- L’ordre d’imbrication des boucles est de la gauche vers la droite.
- On peut utiliser le constructeur list à la place des délimiteurs.
- expression peut elle même être un itérable en compréhension.

Les conteneurs de bases

Construction des itérables en compréhension

```
>>> L= [ i for i in range(1, 21, 2) ]
    L [1, 3, 5, 7, 9, 11, 13, 15, 17, 19]
>>> L=[ (i,j) for i in range(1,3) for j in range(1,4) ]
    L [(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)]
>>> L=[i*j for i in range(1,5) for j in range(1,5) ] ;
    L [1, 2, 3, 4, 2, 4, 6, 8, 3, 6, 9, 12, 4, 8, 12, 16]
>>> L=[(i,j) for i in range(1,5) if i%2 for j in range(1,5) if (i+j)%2==0];
    L [(1, 1), (1, 3), (3, 1), (3, 3)]
>>>[[(100*i+j)//2 if j%2==0 else 100*i+j for j in range(1,i+1)] for i in range(1,5)]
    [[101], [201, 101], [301, 151, 303], [401, 201, 403, 202]]
```

Les conteneurs de bases

Construction des itérables en compréhension

2. Tuples en compréhension:

tuple(expression for indice_1 in iterable_1 if condition_1 ... for indice_n in iterable_n if condition_n)

Attention : sans le constructeur tuple, la construction en compréhension produit un générateur et non pas un tuple.

Exemple:

tuple((i,j) for i in range(1,5) if i%2 for j in range(1,5) if (i+j)%2==0)

((1, 1), (1, 3), (3, 1), (3, 3))

Les conteneurs de bases

Construction des itérables en compréhension

3. Ensembles en compréhension:

set(expression for indice_1 in iterable_1 if condition_1 ... for indice_n in iterable_n if condition_n)

ou encore

{ expression for indice_1 in iterable_1 if condition_1 ... for indice_n in iterable_n if condition_n}

Exemple:

```
e={(x,y) for x in range(5) if x%2 for y in range(5) if y%2==0 }
# ou bien
e=set((x,y) for x in range(5) if x%2 for y in range(5) if y%2==0)
```

{(1, 2), (3, 2), (3, 0), (1, 4), (1, 0), (3, 4)}

05 Les structures conditionnelles

```
if condition:
    # Code to execute if the condition is True
elif other_condition:
    # Code to execute if the first condition was False and this one is True
else:
    # Code to execute if none of the above conditions were True
```

05 Les structures conditionnelles

```
age = 25
if age <= 12:
      print("Child.")
elif age <= 19:
      print("Teenager.")
elif age <= 35:
     print("Young adult.")
else:
     print("Adult.")
# Output: Young adult.
```

05 Les structures conditionnelles

```
match variable:
    case pattern_1:
        # Code to execute if variable matches pattern_1
    case pattern_2:
        # Code to execute if variable matches pattern_2
    # ... more cases
    case _:
        # Code to execute if no other case matches (wildcard, similar to 'default
```

05 Les structures conditionnelles

```
code = 404
match code:
    case 200:
        print("OK")
    case 404:
        print("Not Found")
    case 500:
        print("Server Error")
    case _:
        print("Unknown status")
```

05 Les structures conditionnelles

Python permet de créer des expressions dont l'évaluation dépend d'une condition.

- Principe :

  Si la condition est True → on évalue expression1

  Sinon → on évalue expression2

- Syntaxe générale :

expr1 if condition else expr2

⚠️ Cette construction ne permet pas l'utilisation de elif mais on peut imbriquer plusieurs expressions conditionnelles.

expr1 if cond1 else expr2 if cond2 else expr3

05 Les structures conditionnelles

Exemple 1:

```
x = 5
y = 3
m = x if x > y else y
```

Exemple 2:

```
L1 = [-1, 0, 5, 3]
L2 = [0 if x < 0 else x for x in L1]
```

Exemple 3:

```
x = int(input("? "))
result = ">0" if x > 0 else "<0" if x < 0 else "=0"
print(result)
```

06

Les structures itératives

Boucle non conditionnelle (for)

Permet d'exécuter un bloc pour chaque élément d'un itérable.

- Syntaxe :

```
for v in iterable:
    bloc
```

- Exemple : calcul du produit des éléments d'une liste

```
p = 1
for v in [1, 8, 7]:
  p *= v
print(p)
```

- Avantage :

  - Simple et lisible
  - Très utilisée pour parcourir des collections

06

Les structures itératives

Boucle conditionnelle (while)

Permet d'exécuter un bloc d'instructions tant que la condition est vraie (True).

- Caractéristiques :

  - La condition est testée avant chaque itération

  - Si la condition est False, le bloc n'est jamais exécuté

- Syntaxe :

```
while condition:
    bloc
```

- Exemple : calcul du PGCD de deux entiers

```
while x % y != 0:
    x, y = y, x % y
print(y)
```

06

Les structures itératives

Boucle avec condition d’arrêt (break)

Permet d’exécuter un bloc jusqu’à ce qu’une condition d’arrêt devienne vraie.

- Principe :

  - La condition d’arrêt est testée après l’exécution du bloc

  - On utilise l’instruction break pour sortir de la boucle

- Syntaxe :

```
while True:
    bloc
    if condition_arret:
        break
```

06

Les structures itératives

Boucle avec condition d’arrêt (break)

Permet d’exécuter un bloc jusqu’à ce qu’une condition d’arrêt devienne vraie.

- Exemple : saisie d’une voyelle

```
while True:
    v = input("Saisir une voyelle : ")
    if v in ['a', 'e', 'i', 'o', 'u', 'y']:
    break
```

Exercice 1:

Étant donnée une liste Python, écrivez une fonction qui permute chaque paire d'éléments adjacents. La permutation doit se faire directement dans la liste.

Exemple 1

Entrée : lst = [1, 2, 3, 4]
Sortie attendue : [2, 1, 4, 3]

Exemple 2

Entrée : lst = [5, 8, 9]
Sortie attendue : [8, 5, 9]

Exercice 2 - Corrigé:

```
def permute_paires(lst):
    for i in range(0, len(lst) - 1, 2):
        lst[i], lst[i+1] = lst[i+1], lst[i]
    return lst
```

Exercice 2:

Étant donné un tableau d’entiers nums de longueur n et un entier target, trouvez trois entiers distincts dans nums tels que leur somme soit la plus proche possible de target. Retournez la somme de ces trois entiers.

Exemple 1

Entrée : nums = [-1, 2, 1, -4], target = 1
Sortie : 2
Explication :
La somme la plus proche de la cible est 2 :
 –1+2+1=2-1 + 2 + 1=2

Exemple 2

Entrée : nums = [0, 0, 0], target = 1
Sortie : 0
Explication :
La somme la plus proche de la cible est 0 :
 0+0+0=00 + 0 + 0 =0

Exercice 2 - Corrigé:

```
def threeSumClosest(nums, target):
    nums.sort()
    best = nums[0] + nums[1] + nums[2]

    for i in range(len(nums) - 2):
        l, r = i + 1, len(nums) - 1
        while l < r:
            s = nums[i] + nums[l] + nums[r]
            if abs(s - target) < abs(best - target):
                best = s
            if s < target:
                l += 1
            elif s > target:
                r -= 1
            else:
                return s
    return best
```

Exercice 3:

La suite Count and Say (compter et décrire) est une suite de chaînes de caractères définie récursivement comme suit :

- countAndSay(1) = "1"
- countAndSay(n) est obtenu en appliquant le codage par longueurs (Run-Length Encoding – RLE) sur countAndSay(n - 1).

Codage RLE: Le Run-Length Encoding (RLE) consiste à remplacer une séquence de caractères identiques consécutifs par nombre d'occurrences) + (le caractère).

Exemple 1

Pour la chaîne : "3322251"

On obtient : "23321511"

Exercice 2 - Corrigé:

```
def countAndSay(n):
    result = "1"
    for _ in range(1, n):
        current = ""
        count = 1
        for i in range(1, len(result)):
            if result[i] == result[i - 1]:
                count += 1
            else:
                current += str(count) + result[i - 1]
                count = 1
        # Ajouter le dernier groupe
        current += str(count) + result[-1]
        result = current
    return result
```