

Recapitulatif des syntaxes "de base" en Python

Demander et afficher des informations

Syntaxe	Description
<code>print("message")</code>	Affiche "message" dans la console
<code>v = input("message")</code>	Demande une valeur et la stocke dans <code>v</code>

Calculs

Syntaxe	Description
<code>a + b</code>	Addition de <code>a</code> et <code>b</code>
<code>a - b</code>	Soustraction de <code>a</code> et <code>b</code>
<code>a / b</code>	Division de <code>a</code> par <code>b</code>
<code>a * b</code>	Multiplication de <code>a</code> par <code>b</code>
<code>a % b</code>	Modulo (reste de division) de <code>a</code> par <code>b</code>
<code>a ** b</code>	Exponentiation de <code>a</code> par <code>b</code>

Toutes ces opérations peuvent être appliquées directement sur une variable via la syntaxe du type `a += b` (ajouter `b` à `a` et directement modifier la valeur de `a` avec le résultat).

Types de variable et conversion

Syntaxe	Description
<code>type(v)</code>	Renvoie le type de <code>v</code>
<code>int(v)</code>	Converti <code>v</code> en entier
<code>float(v)</code>	Converti <code>v</code> en float
<code>str(v)</code>	Converti <code>v</code> en string

Chaînes de caractères

Syntaxe	Description
<code>chaine1 + chaine2</code>	Concatène les chaînes de caractères <code>chaine1</code> et <code>chaine2</code>
<code>chaine[n:m]</code>	Retourne les caractères de <code>chaine</code> depuis la position <code>n</code> à <code>m</code>

Syntaxe	Description
chaîne * n	Retourne chaîne concaténée n fois avec elle-meme
len(chaîne)	Retourne la longueur de chaîne
chaîne.replace(a, b)	Renvoie chaîne avec les occurrences de a remplacées par b
chaîne.split(c)	Crée une liste à partir de chaîne en la séparant par rapport au caractère c
chaîne.strip()	"Nettoie" chaîne en supprimant les espaces et \n au début et à la fin
\n	Représentation du caractère 'nouvelle ligne'

Fonctions

```
def ma_fonction(toto, tutu=3):
    une_valeur = toto * 6 + tutu
    return une_valeur
```

Cette fonction :

- a pour nom `ma_fonction` ;
- a pour argument `toto` et `tutu` ;
- `tutu` est un argument optionnel avec comme valeur par défaut l'entier 3 ;
- `une_valeur` est une variable locale à la fonction ;
- elle retourne `une_valeur` ;

Conditions

```
if condition:
    instruction1
    instruction2
elif autre_condition:
    instruction3
elif encore_une_autre_condition:
    instruction4
else:
    instruction5
    instruction6
```

Opérateurs de conditions

Syntaxe	Description
<code>a == b</code>	Egalité entre a et b

Syntaxe	Description
<code>a != b</code>	Différence entre <code>a</code> et <code>b</code>
<code>a > b</code>	<code>a</code> supérieur (strictement) à <code>b</code>
<code>a >= b</code>	<code>a</code> supérieur ou égal à <code>b</code>
<code>a < b</code>	<code>a</code> inférieur (strictement) à <code>b</code>
<code>a <= b</code>	<code>a</code> inférieur ou égal à <code>b</code>
<code>cond1 and cond2</code>	<code>cond1</code> et <code>cond2</code>
<code>cond1 or cond2</code>	<code>cond1</code> OU <code>cond2</code>
<code>not cond</code>	négation de la condition <code>cond</code>
<code>a in b</code>	<code>a</code> est dans <code>b</code> (chaîne, liste, set..)

Inline ifs

```
parite = "pair" if n % 2 == 0 else "impair"
```

Exception, assertions

`try / except` permettent de tenter des instructions et d'attraper les exceptions qui peuvent survenir pour ensuite les gérer de manière spécifique :

```
try:
    instruction1
    instruction2
except FirstExceptionTime:
    instruction3
except Exception as e:
    print("an unknown exception happened ! :" + e.str)
```

Les assertions permettent d'explicitier et de vérifier des suppositions faites dans le code :

```
def une_fonction(n):
    assert isinstance(n, int) and is_prime(n), "Cette fonction fonctionne seulement pour des entiers premiers !"
```

Boucles

Syntaxe	Description
<code>for i in range(0, 10)</code>	Itère sur <code>i</code> de 0 à 9
<code>for element in iterable</code>	Itère sur tous les elements de <code>iterable</code> (liste, set, dict, ...)

Syntaxe	Description
<code>for key, value in d.items()</code>	Itère sur toutes les clefs, valeurs du dictionnaire <code>d</code>
<code>while condition</code>	Répète un jeu d'instruction tant que <code>condition</code> est vraie
<code>break</code>	Quitte immédiatement une boucle
<code>continue</code>	Passes immédiatement à l'itération suivante d'une boucle

Structures de données

Syntaxe	Description
<code>L = ["a", 2, 3.14]</code>	Liste (suite ordonnée d'éléments)
<code>S = { "a", "b", 3 }</code>	Ensemble (éléments unique, désordonné)
<code>D = { "a": 2, "b": 4 }</code>	Dictionnaire (ensemble de clé-valeurs, avec clés uniques)
<code>T = (1,2,3)</code>	Tuple (suite d'élément non-mutables)

Syntaxe	Description
<code>L[i]</code>	<code>i</code> -eme element d'une liste ou d'une tuple
<code>L[i:]</code>	Liste de tous les éléments à partir du <code>i</code> -eme
<code>L[i] = e</code>	Remplace le <code>i</code> -eme element par <code>e</code> dans une liste
<code>L.append(e)</code>	Ajoute <code>e</code> à la fin de la liste <code>L</code>
<code>S.add(e)</code>	Ajoute <code>e</code> dans le set <code>s</code>
<code>L.insert(i, e)</code>	Insère <code>e</code> à la position <code>i</code> dans la liste <code>L</code>
<code>chaine.join(L)</code>	Produit une string à partir de <code>L</code> en intercallant la string <code>chaine</code> entre les elements

Fichiers

Ouvrir et lire un fichier :

```
# Créé un contexte dans lequel le fichier
# est ouvert en lecture en tant que 'f',
# et met son contenu dans 'content'
```

```
with open("/un/fichier", "r") as f:
    content = f.readlines()
```

Ecrire dans un fichier :

```
# Créé un contexte dans lequel le fichier  
# est ouvert en ré-écriture complète et  
# écrit le contenu de 'content' dedans.
```

```
with open("/un/fichier", "w") as f:  
    f.write(content)
```

(Le mode 'a' (append) au lieu de 'w' permet d'ouvrir le fichier pour ajouter du contenu à la fin plutôt que de le ré-écrire)