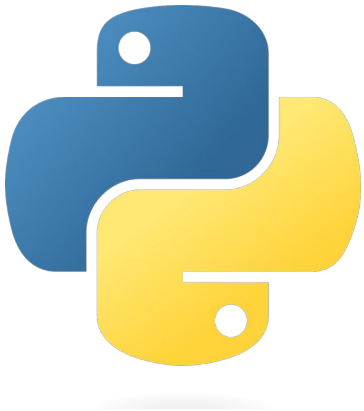


python

Nombres entiers et flottants

Python distingue deux types de nombres : les entiers et les décimaux à virgule flottante

I) Nombres entiers – « int »

```
>>> age = 16
>>> type(age)
<class 'int'>
```

Dans les langages de programmation classiques (C, C++, Java…), la taille des entiers est limitée par l'architecture matérielle des ordinateurs. En revanche, avec Python il n'existe aucune limitation sur la taille des entiers :

```
>>> 2**200
1606938044258990275541962092341162602522202993782792835301376
```

Les calculs avec des entiers sont toujours exacts, même s'ils sont compliqués.

Division entière : $13 = 2 \times 6 + 1$ avec $0 \leq 1 < 2$

```
>>> 13 / 2


>>> 13 // 2


>>> 13 % 2


```

6048592 est-il un multiple de 11 ?

```
>>> 6048592 % 11 == 0
True
```

II) Nombres flottants – « float »

```
>>> taille = 1.78
>>> type(taille)
<class 'float'>
```

```
>>> 51.
51.0
>>> 51
51
```

Attention, le séparateur décimal est le point et non la virgule !

```
>>> taille = 1,78
>>> type(taille)
<class 'tuple'>
```

Il suffit qu'il y ait un seul flottant dans une expression pour que Python renvoie un flottant, même si mathématiquement le résultat est entier :

```
>>> 2 * 1.5 + 10 - 3
10.0
```

De même lors d'une division :

```
>>> 4/2
2.0
```

Les grands nombres flottants sont affichés en notation scientifique :

```
>>> 2.0**200
1.6069380442589903e+60
```

Le codage des nombres flottants en binaire est beaucoup plus délicat que celui des nombres entiers. Une des conséquences est qu'il y a parfois des problèmes d'arrondis :

```
>>> 1/3 + 1/3 + 1/3 + 1/3 + 1/3 + 1/3
1.9999999999999998
>>> 6*1/3
2.0
>>> 1.1 + 2.2
3.3000000000000003
```

En conséquence, il faut être prudent quand on veut tester l'égalité de deux flottants :

```
>>> a = 1.1 + 2.2
>>> a == 3.3
False
>>> abs(a - 3.3) < 10**-15
True
```

Conversions avec int et float :

```
>>> a = 5.3
>>> type(a)
<class 'float'>
>>> b = int(a)
>>> type(b)
<class 'int'>
```

```
>>> int(5.3)


>>> int(5.9999)


>>> round(5.9999)


>>> int("5")


>>> int("007")


>>> float(5)


>>> float("5")


>>> float("5.6")
```

Variables – Affectations

I) Déclarer une variable, affectations

En Python, une variable est un nom qui permet de désigner une valeur.
L'association entre le nom et la valeur s'appelle une affectation et se fait avec le signe =

```
ville1 = 'Paris'
ville2 = ville1
ville1 = 'Lyon'
ville2 = 'Lyon'
nombre1 = 10
nombre2 = 2
arrondissement = nombre1 + nombre2
```

Pour mieux comprendre ce mécanisme d'affectation, on peut utiliser : Python tutor

Le nom d'une variable :
- ne peut contenir que des lettres (accents autorisés), des chiffres ou le blanc souligné _
- ne peut pas commencer par un chiffre (et doit donc commencer par une lettre ou par _)
- ne peut pas être un des mots-clés du langage Python. (if, elif, else, for, while, and, or, not, ...)

Python est sensible à la casse, ce qui signifie que les variables `Test`, `test` ou `TEST` sont différentes.

Barrer ci-dessous les noms de variables incorrects :
aire _ 1connu évident &vident Tourne-vis Glou,glou Génial! While numéro12

Choisir des noms explicites pour les variables (et aussi les fonctions) permet de rendre le code beaucoup plus lisible. Ce n'est donc pas une option !!

Conventions de nommage « camelCase » et « snake_case » :
Si une variable contient par exemple un nombre de participants, on l'appellera :
`nbreParticipants` (camelCase) ou encore `nbre_participants` (snake_case)

Python accepte les affectations multiples :

```
>>> a, b = -5, 4
>>> a, b = b, a
```

Affectations « augmentées » : `+=`, `-=`, `*=`, `/=`, `//=`, `%=`, `**=`, …
Tester les lignes suivantes :

```
>>> a = 2
>>> a += 4
>>> a /= 3
```

```
>>> b = "allez "
>>> b += "les bleus "
>>> b *= 3
```

II) Type d'une variable

Le « type » d'une variable désigne sa nature et est déterminé par Python à chaque affectation.

Les types que nous allons rencontrer cette année sont :
Types de base : int (nombre entier), float (nombre décimal), str (chaîne de caractère), bool (booléen),
Types évolués : list (listes Python), tuple (tuples), dict (dictionnaire)

La fonction « type() » permet de vérifier le type d'une variable :

```
>>> a = input()
45
>>> type(a)
<class 'str'>
```

```
if type(a) == int:
    print("C'est un entier")
elif type(a) == float:
    print("C'est un flottant")
```

Entrées – Sorties

I) Entrées avec « input »

Pour permettre à l'utilisateur du programme d'entrer une valeur en cours d'exécution, on utilise la fonction « input() » :

```
prénom = input('Quel est ton prénom ? ')
naissance = int(input('Quel ton année de naissance ? '))
print(prénom, 'tu as', 2020-naissance, 'ans ;)')
```

Attention :

- La valeur retournée par input est toujours une chaîne de caractère. Si on attend un nombre, il faut penser à convertir la réponse avec les fonctions int() ou float().
- Ne pas oublier de laisser un espace blanc en fin de message d'entrée pour que la réponse de l'utilisateur ne soit pas collée à ce message.

II) Sorties avec « print »

Pour afficher un message lors de l'exécution d'un programme, on utilise la fonction « print() » :

```
print('Hello world!')
```

Il est possible d'afficher plusieurs chaînes de caractères ou valeurs de variables :

```
print("cher", prénom, "tu auras l'an prochain", age + 1, "ans")
```

Code à essayer :

```
print()
print("Une ligne\nsur\nplusieurs lignes et avec\tune tabulation.")
```

Par défaut, Python passe à la ligne en fin de print, mais ce comportement peut être modifié :

```
print("Les chiffres sont : ", end="")
for i in range(10):
    print(i, end=", ")

Les chiffres sont : 0, 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

III) F-strings et formatage des variables

Le mécanisme des f-strings permet de simplifier l'affichage des variables :

```
>>> nom, âge = "Étienne", 16
>>> print(f"{nom} a {âge} ans")
Étienne a 16 ans
```

Attention au « f » avant le guillemet de début !

Les f-strings permettent également de formater l'affichage des variables :

```
>>> quotient = 1/3
>>> print(f"J'arrondi 1/3 au millième près : {quotient:4.3}")
J'arrondi 1/3 au millième près : 0.333
```

Explication : Ici, 4 désigne le nombre de chiffres à afficher et 3 le nombre de décimales voulu.
(La syntaxe complète de ces formatages est délicate à connaître par coeur. En revanche, il faut savoir la retrouver sur internet ;-)

Tests et expressions booléennes

I) Test « if-elif-else »

Syntaxe générale :

```
if <condition 1>:
    print("la condition 1 est vraie")
elif <condition 2>:
    print("la condition 1 est fausse, mais la condition 2 est vraie")
elif <condition 3>:
    print("les conditions 1 et 2 sont fausses, mais la condition 3 est vraie")
...
else:
    print("aucune des conditions précédentes n'est vraie")
print("fin du if, le script continue ici")
```

Exemples :

```
a = int(input())
b = int(input())
if b < a:
    a, b = b, a
print(a, "<", b)
```

```
if a % 2 ==0:
    print(a, "est pair")
else:
    print(a, "est impair")
```

```
if température < 0:
    print("c'est de la glace")
elif température < 100:
    print("c'est de l'eau")
else:
    print("c'est de la vapeur")
```

Ne pas oublier les deux points ni l'indentation, surtout si les tests sont imbriqués !
Lequel des 2 scripts ci-dessous est correct ?

```
if a >= 0:
    if a % 2 == 0:
        if a % 3 == 0:
            print("a est multiple de 6")
        print("a est multiple de 2")
        else:
            print("a est impair")
    elif a > -5:
        print("-5 < a < 0")
```

```
if a >= 0:
    if a % 2 == 0:
        if a % 3 == 0:
            print("a est multiple de 6")
        print("a est multiple de 2")
    else:
        print("a est impair")
elif a > -5:
    print("-5 < a < 0")
```

Lequel des 2 scripts ci-dessous est le plus efficace ? Pourquoi ?

```
if marque == "Renault":
    print("voiture française")
if marque == "Toyota":
    print("voiture japonaise")
if marque == "Wolkswagen":
    print("voiture allemande")
if marque == "Fiat":
    print("voiture italienne")
```

```
if marque == "Renault":
    print("voiture française")
elif marque == "Toyota":
    print("voiture japonaise")
elif marque == "Wolkswagen":
    print("voiture allemande")
elif marque == "Fiat":
    print("voiture italienne")
```

II) Expressions booléennes

Une expression booléenne est une expression qui est soit vraie, soit fausse. En Python, sa valeur est « True » ou « False ». (Attention à la majuscule)
Les conditions utilisées dans les if et elif sont des expressions booléennes.
Pour écrire de telles expressions, on peut s'appuyer sur les opérateurs suivants :

== > >= < <= != and or not

Attention : L'opérateur = est réservé à l'affectation ! Pour tester une égalité, on utilise ==

Exemples à tester :

```
>>> a = "nsi"
>>> a == "nsi"


>>> a = true      # avec un t minuscule
>>> a = True      # avec un T majuscule


>>> 1 + 2 == 3


>>> 1 > 5


>>> (12 % 2 == 0) and (12 % 3 == 0) \
    and (12 % 4 == 0)


>>> a, b, c = 1, 5, 12
>>> a < b < c
```

```
>>> age = 17
>>> majeur = (age >= 18)
>>> majeur


>>> not(majeur)


>>> type(majeur)


>>> 'AAA' == 'A'*3


>>> "répertoire" < "répertorié"


>>> "10" < "5"
```

Proposer des valeurs de `x`, `y` et `z` pour lesquelles l'expression ci-dessous est vraie :

```
>>> (x == y) and (x < z + 5 or not(x == y + z))
```

Comment peut-on écrire plus simplement les expressions ci-dessous ?

```
not(a < 10)
```

```
(a < 5) and (a <= 4)
```

```
(a < 5) or (a <= 4)
```

« Évaluation paresseuse » : Comment expliquer les réponses de la console Python ?

```
>>> x = 0
>>> (x != 0) and (1/x == 0.5)
False
>>> (1/x == 0.5) and (x != 0)
Traceback (most recent call last):
  File "<pyshell>", line 1, in <module>
ZeroDivisionError: division by zero
```

Boucles

Les boucles permettent de répéter plusieurs fois un même bloc d'instructions.
Quand on connaît à l'avance le nombre de répétitions voulu, on choisit en général une boucle for. Dans le cas contraire, on choisit une boucle while.

I) Boucle « for »

Syntaxe générale :

```
for <variable> in <séquence>:
    <bloc instructions>
```

Exemples :

```
for i in range(3):
    print(i)
```

```
for c in 'oui':
    print(c)
```

```
for nombre in [1, 4, 9, 16, 25, 36]:
    print(nombre, "est un carré parfait")
```

Ne pas oublier les deux points ni l'indentation !
Bien distinguer les deux codes ci-dessous :

```
for c in 'oui':
    print(c)
    print('---')
    print('***')
```

```
for c in 'oui':
    print(c)
print('---')
print('***')
```

Les 3 façons d'appeler « range » en fonction du nombre de paramètres :

```
range(10)           # retourne les entiers de 0 à 9
range(3, 10)        # retourne les entiers de 3 à 9
range(10, 3, -1)    # retourne les entiers de 10 à 4
```

Les boucles peuvent bien sûr être imbriquées :

```
for i in range(1,10):
    for j in range(0,i):
        print(i*j, end=' ')
    print()
```

Dès que le code devient un peu compliqué, penser à utiliser un débugueur pour exécuter ce code pas à pas !

II) Boucle « while »

Syntaxe générale :

```
while <condition>:
    <bloc instructions>
```

Exemples :

```
# Vérification du mot de passe :
mdp = ""
while mdp != "gé4ù@12W":
    mdp = input("Mot de passe ? ")
```

```
# Le nombre de bactéries double chaque
# jour et il y en 10 au départ:
nbre_bactéries = 10
nbre_jours = 0
while nbre_bactéries < 100:
    nbre_bactéries = nbre_bactéries * 2
    nbre_jours = nbre_jours + 1
print(nbre_jours)
```

Dans les deux cas ci-dessus, on ne peut pas utiliser un for car on ne sait pas à l'avance combien d'itérations seront nécessaires.
En revanche, les boucles for peuvent toujours être traduites en while :

```
for i in range(3):
    print("hello")
```

```
i = 0
while i < 3:
    print("hello")
    i = i+1
```

Attention aux boucles while infinies :

- Il faut penser à initialiser les variables dont dépend la condition de boucle.
- Il faut aussi penser à les modifier dans le corps de la boucle !

Exemple : Tester le code ci-dessous et corriger les 2 erreurs.

```
# On cherche si le caractère "u" est présent dans la variable texte
texte = "bonjour"
found = False
while (not found) and (i < len(texte)):
    if texte[i] == "u":
        found = True
print(found)
```

Fonctions

I) Fonctions prédéfinies

Exemples de fonctions déjà rencontrées :

```
>>> réponse = input("Votre prénom SVP ? ")
>>> entier = int(10.1)
>>> longueur_du_mot = len('informatique')
>>> chaîne = str(12)
```

A vous de proposer d'autres exemples de fonctions :

Une fonction a en principe un ou plusieurs paramètres en entrée et retourne une valeur en sortie.

```
>>> plus_petit = min(5, 8, 4, 1, 12)
>>> plus_petit
1
```

Les fonctions sont très importantes pour :
- organiser son code en petites unités plus simples et plus lisibles.
- ne pas avoir à retaper le même code plusieurs fois dans un même programme.
- pouvoir facilement utiliser le code écrit par d'autres, sans avoir besoin de le comprendre !

Une fonction doit toujours être appelée en écrivant les parenthèses, même s'il n'y a pas de paramètres en entrée :

```
>>> réponse = input()
>>> print()
```

On peut bien sûr combiner plusieurs appels de fonctions :

```
>>> double = abs(int(input())) * 2
```

II) Appel de fonctions définies dans un module externe

L'une des forces d'un langage comme Python est qu'il possède de très nombreuses bibliothèques de fonctions qui permettent de faire rapidement et simplement des choses très compliquées !
En Python, ces bibliothèques sont appelées « modules »

Les 3 façons d'appeler une fonction dans un module :

```
>>> import math
>>> math.sqrt(9)
3.0
```

```
>>> from math import sqrt
>>> sqrt(9)
3.0
```

```
>>> from math import *
>>> sqrt(9)
3.0
```

La méthode de droite permet d'importer d'un seul coup toutes les fonctions d'un module mais elle est à déconseiller fortement, notamment si on appelle plusieurs modules dans son script, car il arrive régulièrement que des fonctions faisant des choses différentes et étant dans différent modules aient le même nom !

A connaître dans le module « Math » :

```
sqrt(x)         # Racine carrée
floor(x)        # Arrondi à l'entier inférieur
ceil(x)         # Arrondi à l'entier supérieur
gcd(x,y)        # PGCD de x et y
sin(x), cos(x), tan(x), asin(x),...
pi              # Nombre pi avec la meilleure précision permise par un flottant
```

A connaître dans le module « Random » :

```
random()        # Flottant sur [0 ; 1[
randint(a,b)    # Entier entre a et b inclus
choice(liste)   # Élément au hasard dans la liste
```

III) Définir ses propres fonctions

Exemple :

```
def triple(nbre):            # Ne pas oublier les deux points, ni l'indentation !
    résultat = nbre * 3
    return résultat
```

Pour appeler cette fonction, on fait comme pour les fonctions prédéfinies :

```
>>> a = triple(7)
>>> a
21
```

```
>>> b = triple("7")
>>> b
'777'
```

IV) « Docstring » d'une fonction

En tout début de fonction, on prendra l'habitude d'écrire une « docstring » contenant :

- une explication sur ce que fait la fonction.
- la liste des paramètres d'entrée précisant les types possibles de ces paramètres et une brève description.
- les valeurs de sortie précisant les types possibles de ces valeurs et une brève description.

Les docstrings étant sur plusieurs lignes, elles sont encadrées habituellement par des triples guillemets.
On peut afficher la docstring d'une fonction dans la console Python avec la fonction help

V) Test d'une fonction avec « assert »

Pour tester nos fonctions, nous écrirons des « jeux de test » les plus complets possibles en nous appuyant sur l'instruction « assert » dont la syntaxe générale est :

```
assert <condition>, "message à afficher si la condition est fausse"
```

Si la condition est vérifiée, l'exécution du programme continue. Si elle ne l'est pas, le programme s'interrompt et affiche l'éventuel message d'erreur.

VI) Exemple avec docstring et assert :

```
def ordonner2nombres(a, b):
    """
    Compare les nombres a et b et les retourne en ordre croissant (a < b)
    Entrées : a et b (int ou float)
    Sorties : a et b (int ou float) par ordre croissant
    """
    if a <= b:
        return a, b
    else:
        return b, a

def ordonner3nombres(a, b, c):
    """
    Compare les nombres a, b et c et les retourne en ordre croissant (a < b < c)
    Entrées : a, b et c (int ou float)
    Sorties : a, b et c (int ou float) par ordre croissant
    """
    a, b = ordonner2nombres(a, b)
    a, c = ordonner2nombres(a, c)
    b, c = ordonner2nombres(b, c)
    return a, b, c

assert ordonner3nombres(1, 5, 7) == (1, 5, 7), "oups, ça ne marche pas !"
assert ordonner3nombres(1, 7, 5) == (1, 5, 7), "oups, ça ne marche pas !"
assert ordonner3nombres(5, 1, 7) == (1, 5, 7), "oups, ça ne marche pas !"
assert ordonner3nombres(5, 7, 1) == (1, 5, 7), "oups, ça ne marche pas !"
assert ordonner3nombres(7, 1, 5) == (1, 5, 7), "oups, ça ne marche pas !"
assert ordonner3nombres(7, 5, 1) == (1, 5, 7), "oups, ça ne marche pas !"
```

Listes Python

I) Qu'est-ce qu'une liste ?

Une liste est une structure de données séquentielle capable de contenir un nombre quelconque de données de types quelconques.
En Python, on écrit les listes entre crochets [ ], les items étant séparés par des virgules.

```
>>> chiffres = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> mélange = [True, "Vert", 10, "Vert", [1, 2, 3], 5.2]
>>> liste_vide = []
```

Le type des listes est `'list'`.

```
>>> type (chiffres)
<class 'list'>
```

Différentes façons d'initialiser une liste :

```
>>> l = [1, 5, -2, 4]              # En énumérant tous les éléments


>>> l = [1,2,3,4] + [5,6]          # En concaténant deux listes


>>> l = [0] * 10                   # Par répétition


>>> l = list("coucou")             # En transformant une séquence en liste
```

II) Accéder à un item d'une liste

On peut accéder à un item d'une liste via son « indice », qui est un entier compris entre 0 (inclus) et la longueur de la liste (exclue) :

```
>>> liste = ["chien", "chat", "souris", "oiseau", "oie", "cheval"]
# indices :     0        1        2         3       4        5
>>> len(liste)
6
>>> liste[0]
'chien'
>>> liste[3]
'oiseau'
```

Python accepte également des indices négatifs :

```
>>> liste = ["chien", "chat", "souris", "oiseau", "oie", "cheval"]
# indices nég : -6       -5       -4        -3      -2       -1
>>> liste[-1]
'cheval'
>>> liste[-4]
'souris'
```

On peut enfin modifier directement un item d'une liste :

```
>>> liste = list(range(5, 10))
>>> liste[0] = 3
>>> liste[-1] = 12
>>> liste
[3, 6, 7, 8, 12]
```

III) Parcourir une liste

Comparer les 2 méthodes de parcours d'une liste ci-dessous :

```
voyelles = list("aeiouy")
for c in voyelles:
    print(c)
```

```
voyelles = list("aeiouy")
for i in range(len(voyelles)):
    print(voyelles[i])
```

IV) Quelques opérations usuelles

```
>>> fruits = ["pomme", "poire", "orange", "fraise"]
>>> "poire" in fruits                          # teste la présence d'un item
True
>>> len(fruits)                                # nombre d'items
4
>>> min(nombres), max(nombres), sum(nombres)  # minimum, maximum et somme des items
(0, 8, 14)
>>> fruits.index("orange")                     # retourne l'index de l'item spécifié
2
>>> fruits.append("kiwi")                      # ajoute une valeur en fin de liste
>>> fruits
['pomme', 'poire', 'orange', 'fraise', 'kiwi']
>>> fruits.insert(2, "banane")                 # insère à l'endroit spécifié
>>> fruits
['pomme', 'poire', 'banane', 'orange', 'fraise', 'kiwi']
>>> fruits.remove('orange')                    # supprime l'élément dont la
>>> fruits                                     # valeur est spécifiée
['pomme', 'poire', 'banane', 'fraise', 'kiwi']
>>> del fruits[4]                              # supprime l'élément à la
>>> fruits                                     # position spécifiée
['pomme', 'poire', 'banane', 'fraise']
>>> a = fruits.pop(2)                          # même chose mais retourne en plus
>>> a                                          # la valeur de l'élément supprimé
'banane'
>>> fruits
['pomme', 'poire', 'fraise']
>>> fruits.reverse()                           # inverse l'ordre de la liste
>>> fruits
['fraise', 'poire', 'pomme']
>>> nombres = [5, 8, 1, 0]
>>> nombres2 = sorted(nombres)                 # renvoie une nouvelle liste triée
>>> nombres                                    # sans modifier la liste initiale
[5, 8, 1, 0]
>>> nombres2
[0, 1, 5, 8]
>>> nombres.sort()                             # trie la liste et ne renvoie rien
>>> nombres
[0, 1, 5, 8]
```

V) Slices : [début:fin:pas]

```
>>> liste = [1, 4, 7, 9, 12, 15, 17]
>>> liste[2:4]                                 # de l'indice 2, à 4 non compris
[7, 9]
>>> liste[2:]                                  # de l'indice 2, à la fin
[7, 9, 12, 15, 17]
>>> liste[:4]                                  # du début, à l'indice 4 non compris
[1, 4, 7, 9]
>>> liste[1:6:2]                               # de 1, à 6 non compris, par pas de 2
[4, 9, 15]
>>> liste[1:-1]                                # de 1, au dernier non compris
[4, 7, 9, 12, 15]
>>> liste[::-1]                                # à l'envers
[17, 15, 12, 9, 7, 4, 1]
```

Chaînes de caractères

I) Délimiter une chaîne de caractères

Une chaîne de caractère peut être délimitée de 4 façons différentes :
- par des apostrophes `'`
- par des guillemets `"`
- par des triples apostrophes `'''`
- par des triples guillemets : `"""`

```
print("Une chaîne avec des 'apostrophes'.")
print('Une chaîne avec des "guillemets".')
print('''Une chaîne contenant des 'apostrophes' et des "guillemets". ''')
print("""Une chaîne
sur plusieurs lignes""")
```

Le choix des apostrophes ou des guillemets dépend donc souvent du contenu de la chaîne.

II) Caractères d'échappement

Lorsque Python rencontre un anti-slash « \ » dans une chaîne de caractères, il ne l'affiche pas et le considère comme le début d'un caractère spécial à afficher.

Caractère spécial	Signification	Exemple
`\n`	Saut de ligne	`print("Une chaîne\nsur plusieurs\nlignes.")`
`\t`	Tabulation	`print("Une chaîne\tavec\tdes tabulations")`
`\"`	Guillemet	`print("Une chaîne avec des \"guillemets\".")`
`\'`	Apostrophe	`print('Une chaîne avec des \'apostrophes\'.')`
`\\`	Anti-slash	`print('Une chaîne avec trois anti-slashs \\\\\\.')`

L'anti-slash est appelé « caractère d'échappement ».

III) Fonctions et Opérations sur les chaînes

Longueur d'une chaîne :
```
>>> len("Assurancetourix")
15
```

Concaténer deux chaînes :
```
>>> prénom = "Roger"
>>> nom = "Federer"
>>> prénom + " " + nom
'Roger Federer'
```

Répéter une chaîne :
```
>>> rire = "Ah "*10
>>> rire
'Ah Ah Ah Ah Ah Ah Ah Ah Ah Ah '
```

Comparer 2 chaînes selon l'ordre des caractères dans la table Unicode :
```
>>> "A" < "B" < "a" < "b"
True
>>> "chant" > "champ"
True
```

Convertir un numéro Unicode en caractère et réciproquement :
```
>>> ord("a")
97
>>> chr(97)
'a'
```

IV) Méthodes applicables aux chaînes

Convertir en majuscules ou en minuscules :

```
>>> ville = "Paris"
>>> ville.upper()
'PARIS'
>>> ville.lower()
'paris'
```

Supprimer des espaces blancs (ou autres caractères) aux extrémités d'une chaîne :

```
>>> ligne = "   phrase  pleine    de   trous   "
>>> ligne.strip()
'phrase  pleine    de   trous'
>>> ligne.rstrip()
'   phrase  pleine    de   trous'
>>> ligne.lstrip()
'phrase  pleine    de   trous   '
```

Remplacer une sous-chaînes de caractères par une autre :

```
>>> a = "I love London"
>>> a.replace("London", "Paris")
'I love Paris'
```

Concaténer une liste de chaînes en une unique chaîne et réciproquement :

```
>>> "-".join(["a", "e", "i", "o", "u", "y"])
'a-e-i-o-u-y'
>>> 'a-e-i-o-u-y'.split("-")
['a', 'e', 'i', 'o', 'u', 'y']
>>> list("aeiouy")
['a', 'e', 'i', 'o', 'u', 'y']
```

V) Accéder aux caractères d'une chaîne

Avec un indice ou un slice :

```
>>> a = "J'aime NSI !"
>>> a[4]
'm'
>>> a[7:-2]
'NSI'
```

Avec une boucle for :

```
for c in "France":
    print(c)
```

Attention, une chaîne de caractère Python est « non mutable », ce qui veut dire que l'on ne peut modifier directement un des caractères de la chaîne. En revanche, on peut écraser la chaîne à modifier par une nouvelle valeur.

Tester le code ci-dessous :

```
>>> a = "premiere NSI"
>>> a[5] = 'è'
>>> a = "première NSI"
```

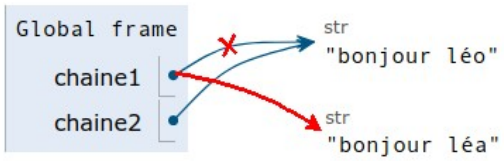
Types mutables – Types non-mutables

Tester les codes ci-dessous avec : Python tutor

I) Variables dont le type est non-mutable

La seule façon de modifier une variable de type `int`, `float`, `bool` ou `str`, est d'écraser l'ancienne valeur en lui affectant une nouvelle valeur. On dit que les types `int`, `float` et `str` sont « non mutables ».
En effet, si l'on tape par exemple : `chaine = "bonjour léo"`, puis que l'on veut renommer léo en léa, il faut refaire une affectation avec : `chaine = "bonjour léa"`. Nous n'avons pas modifié l'ancienne chaîne, nous l'avons remplacé par une nouvelle.

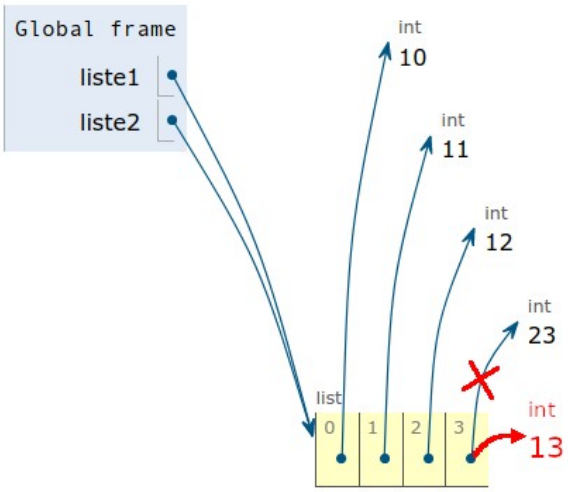
```
chaine1 = "bonjour léo"
chaine2 = chaine1
chaine1 = "bonjour léa"
```



II) Variables dont le type est mutable

En revanche, on peut modifier une variable de type `list` sans écraser son ancienne valeur. On peut par exemple modifier un de ses items ou utiliser une méthode comme : append, pop, sort…
Le type `list` est donc dit « mutable ».
Si par exemple, on tape : `liste1 = [10, 11, 12, 23]`, on peut remplacer le 23 par un 13 avec : `liste1[3] = 13`. La liste n'est donc pas une « valeur figée » que Python stocke en mémoire et qui n'est plus modifiable mais plutôt un « conteneur » qui pointe vers des valeurs que l'on peut réaffecter.

```
liste1 = [10, 11, 12, 23]
liste2 = liste1
liste1[3] = 13
```



C'est très puissant, mais cela a une conséquence importante à comprendre quand on veut créer une copie d'une liste qui soit indépendante de l'original. En effet, l' affectation `liste2 = liste1` ne suffit pas car elle crée une nouvelle référence `liste2` qui pointe vers la même liste concrète que `liste1`. Si l'on veut forcer la création d'un nouveau conteneur, on peut par exemple utiliser un slice : `liste2 = liste1[:]`

De même, lorsque l'on passe une liste en paramètre à une fonction, une nouvelle référence à cette liste est créée en local dans la fonction mais continue à pointer vers la liste originale. Si donc cette liste est modifiée dans le corps de la fonction, cette modification se répercute dans le programme appelant.

Cette année, nous rencontrerons 2 types mutables : list, dict

Tuples

I) Définition

Les « tuples » sont en quelque sorte des listes non mutables (c'est à dire non modifiables sans les écraser par une nouvelle valeur).
Contrairement aux listes, on ne peut modifier directement un item (`mon_tuple[0] = 1`), ni appliquer des méthodes comme : append, insert, remove…
Mais pour le reste, les tuples sont comme les listes.

Les tuples s'écrivent entre parenthèses (même si celles-ci ne sont pas toujours obligatoire) :

```
>>> jours_semaine = ("lun", "mar", "mer", "jeu", "ven", "sam", "dim")
>>> jours_semaine[0]


>>> jours_weekend = jours_semaine[-2:]


>>> len(jours_semaine)


>>> jours_semaine.index("mer")


>>> pièce = "pile", "face"
>>> type(pièce)
<class 'tuple'>
```

II) Quand utiliser des tuples plutôt que des listes ?

En fait, nous avons déjà utilisé des tuples sans le savoir :

- Affectation multiple :

```
>>> a, b, c = 0, 10, 5
```

- Renvoyer plusieurs valeurs en fin de fonction :

```
def dans_l_ordre(a, b):
if b < a:
    return b, a
else:
    return a, b

i, j = dans_l_ordre(6,3)
```

- Définir une liste de constantes qui n'ont aucune raison d'être modifiées dans le programme (un tuple est plus économe en ressources qu'une liste) :

```
>>> jours_semaine = ("lun", "mar", "mer", "jeu", "ven", "sam", "dim")
```

Listes Python avancées

I) Listes de listes

On peut créer des listes de listes :

```
>>> zoo = [ [4, "zèbres"], [2, "lions"], [3, "pumas"] ]
>>> zoo[0]
[4, 'zèbres']
>>> zoo[0][1]
'zèbres'
```

```
zoo = [ [4, "zèbres"], [2, "lions"], [3, "pumas"] ]
for espèce in zoo:
    print(f"Il y a {espèce[0]} {espèce[1]} dans le zoo")
```

Les listes de listes permettent notamment de stocker des tableaux à plusieurs dimensions
Par exemple, la table de multiplication ci-contre se traduit en Python par :

```
>>> table = [ [0, 0, 0, 0], [0, 1, 2, 3], [0, 2, 4, 6], [0, 3, 6, 9] ]
>>> table[2]
[0, 2, 4, 6]
>>> table[2][3]
6
```

*	0	1	2	3
0	0	0	0	0
1	0	1	2	3
2	0	2	4	6
3	0	3	6	9

II) Compréhensions de listes

Une « compréhension de liste » est une expression qui permet de construire une liste à partir de tout type itérable (liste, tuple, range, chaîne de caractères…). Le résultat obtenu est toujours une liste.

```
>>> liste1 = [i**2 for i in range(10)]
>>> liste2 = [1/i for i in range(10) if i%2 != 0]
```

Syntaxe générale :

```
new_list = [function(item) for item in list if condition(item)]
```

Comparaison avec une boucle for :

```
alphabet = []
for i in range(97, 123):
    alphabet.append(chr(i))
```

```
alphabet = [chr(i) for i in range(97, 123)]
```

Deviner quelles listes produisent les compréhensions de listes ci-dessous et écrire les boucles for équivalentes :

```
>>> l1 = [i for i in range(10)]


>>> l2 = [2 for i in range(10)]


>>> l3 = [v for v in "aeiouy"]


>>> l4 = [int(x) for x in '1 45 85 1 10 7 8'.split()]


>>> l5 = [n*2 for i in [[0, 'a'], [2, 'b'], [3, 'c']] for n in i]


```

Dictionnaires

I) Qu’est-ce qu’un dictionnaire ?

Un dictionnaire est une structure de données permettant d’associer des « clés » avec des « valeurs ».
Contrairement aux listes, on n'accède pas à un élément de dictionnaire par son indice mais par sa clé :

```
>>> stock = {"paire_chaussettes": 3, "chemise": 5, "pantalon": 4}
>>> stock
{'paire_chaussettes': 3, 'chemise': 5, 'pantalon': 4}
>>> stock["paire_chaussettes"]
3
>>> stock["paire_chaussettes"] = 5
>>> stock
{'paire_chaussettes': 5, 'chemise': 5, 'pantalon': 4}
```

Si on affecte une valeur à une clé qui n’existe pas, Python ajoute cette nouvelle clé au dictionnaire :

```
>>> stock["veste"] = 2
>>> stock
{'paire_chaussettes': 5, 'chemise': 5, 'pantalon': 4, 'veste': 2}
```

On peut supprimer une clé et sa valeur associée :

```
>>> del stock["pantalon"]
>>> stock
{'paire_chaussettes': 5, 'chemise': 5, 'veste': 2}
```

Le mot clé « `in` » permet de tester l'appartenance d'une clé à un dictionnaire mais pas d'une valeur :

```
>>> 'paire_chaussettes' in stock
True
```

Le type des dictionnaires est `'dict'`. C’est un type mutable.

```
>>> type(stock)
<class 'dict'>
```

Les clés doivent être de type non mutable (int, float, str, …) tandis que les valeurs peuvent être de n’importe quel type, même mutable :

```
>>> départements = {1: ("Ain", "Bourg-en-Bresse"), 2: ("Aisne", "Laon"), ...}
>>> dressing = {"chemise" : ["M", "Bleu", "tiroir du bas"], ...}
```

II) Parcourir un dictionnaire

Il y a 3 façons d’énumérer le contenu d’un dictionnaire :

```
for clé in stock.keys():                    # Énumération par clés
    print(clé)

paire_chaussettes
chemise
veste
```

```
for valeur in stock.values():               # Énumération par valeurs
    print(valeur)

5
5
2
```

```
for item in stock.items():                  # Énumération par tuples (clé, valeur)
    print(item)

('paire_chaussettes', 5)
('chemise', 5)
('veste', 2)
```

Fichiers textes

Python permet bien sûr de lire et modifier des fichiers contenant du texte.
Pour tester les exemples ci-dessous, ouvrir un éditeur de texte, copier-coller le contenu ci-dessous et enregistrer le fichier dans votre répertoire courant sous le nom « `bd.txt` » :

```
Astérix
Lucky luke
Blake et Mortimer
Les Schtroumpfs
Tintin
```

I) Ouvrir et fermer un fichier texte

Avant de pouvoir lire ou modifier un fichier contenant du texte, il faut l'ouvrir. A la fin, il faut le fermer :

Syntaxe 1 :

```
# ouverture du fichier :
fichier = open("bd.txt", "r")
# lecture du contenu :
texte = fichier.read()
# fermeture du fichier :
fichier.close()
```

Syntaxe 2 : (à préférer)

```
# ouverture du fichier :
with open("bd.txt", "r") as fichier:
    # lecture du contenu :
    texte = fichier.read()
# fermeture du fichier automatique
# en quittant l'indentation
```

Remarques :
- Pour que le script fonctionne, il doit être enregistré dans le même dossier que le fichier `bd.txt`
- Est-ce que le contenu de la variable texte correspond à ce que l'on attendait ?
- Le « `r` » en 2ème paramètre du open, signifie que l'on a ouvert le fichier en lecture seulement.

Les principaux mode d'ouverture d'un fichier :

Mode	Signification	Si le fichier existe	Si le fichier n'existe pas
`"r"`	Lecture seulement (read) mode par défaut	Le pointeur est placé au début du fichier.	Erreur
`"w"`	Écriture seulement (write)	Écrase le contenu du fichier	Crée un nouveau fichier.
`"a"`	Écriture seulement (append)	Le pointeur est placé en fin de fichier.	Crée un nouveau fichier.
`"r+"` `"w+"` `"a+"`	Lecture - Écriture	Voir ci-dessus	Voir ci-dessus

II) Lire un fichier texte

Lire l'intégralité du fichier et stocker le contenu dans une chaîne de caractères :

```
texte = fichier.read()
```

Lire l'intégralité du fichier et stocker le contenu ligne par ligne dans une liste :

```
liste = fichier.readlines()
```

Lire la ligne suivante du fichier dans une chaîne de caractères :

```
texte = fichier.readline()
```

Lire les n caractères suivants du fichier dans une chaîne de caractères :

```
texte = fichier.read(n)
```

On peut aussi parcourir le fichier avec une boucle for :

```
for ligne in fichier:
    print(ligne)
```

III) Écrire dans un fichier texte

Écrire une chaîne de caractères à la position courante du pointeur :

```
fichier.write(texte)
```

Écrire une liste de chaînes de caractères à la position courante du pointeur :

```
fichier.writelines(texte)
```

Remarques :

- Aussi bien en lecture qu'en écriture, les fonctions ci-dessus n'ajoutent pas ni n'enlèvent les caractères de fin de ligne « `\n` ». C'est donc au programmeur de les enlever ou de les ajouter selon ce qu'il souhaite.
- Dans certains cas, on peut avoir besoin de la méthode `.seek()` qui permet de déplacer le pointeur de fichier à une position définie. Si on a par exemple lu un fichier une première fois et que l'on veut le relire une deuxième fois sans le refermer, on peut remettre le pointeur de fichier en début de fichier avec : `fichier.seek(0, 0)` (voir doc Python…)

IV) Module os

Quand on travaille sur les fichiers, on peut avoir besoin de supprimer un fichier ou de le renommer. Le module « `os` » peut nous y aider :

```
import os         # Charge le module os
chdir(path)       # Change le répertoire en cours
getcwd()          # Renvoie le répertoire en cours
listdir(path)     # Renvoie la liste des fichiers et répertoires contenus dans path
mkdir(path)       # Crée un nouveau répertoire
remove(path)      # Supprime un fichier
rmdir(path)       # Supprime un répertoire vide
rename(src,dst)   # Renomme un fichier ou un répertoire
```

V) Gestion des erreurs d'exécution

Lors de l'exécution d'un programme, il peut arriver qu'une instruction pourtant correcte déclenche une erreur. Par exemple, votre programme est en train de travailler sur un fichier qui est sur une clé usb et l'utilisateur retire la clé usb brutalement !
Ces erreurs d'exécution sont appelées « exceptions » et ne sont pas toujours fatales, grâce à l'instruction `try-except` :

```
try:
    # on met dans le try l'opération susceptible de déclencher une erreur :
    r=a/b
except ZeroDivisionError:
    # Code exécuté uniquement si l'erreur est de type "ZeroDivisionError"
    print("impossible de diviser par 0")
else:
    # Code exécuté uniquement si il n'y a pas eu d'erreur
    print(r)
finaly:
    # Code exécuté que l'on soit passé par le except ou par le else
    print(a,b)
# retour de l'indentation = suite du programme
```

Fonctions avancées

I) « Portée » des variables

Tester pas à pas les scripts ci-dessous avec Python Tutor (liens ci-dessous) ou le débugueur de Thonny.

script 1 : (avec Python Tutor)

```
def fct():
    a = 3
    print("dans fct : a =", a)

a = 1
print("début programme principal : a =", a)
fct()
print("fin programme principal : a =", a)
```

```
début programme principal : a =
dans fct : a =
fin programme principal : a =
```

script 2 : (avec Python Tutor)

```
def fct():
    print("dans fct : a =", a)

a = 1
print("début programme principal : a =", a)
fct()
print("fin programme principal : a =", a)
```

```
début programme principal : a =
dans fct : a =
fin programme principal : a =
```

script 3 : (avec Python Tutor)

```
def fct(a):
    print("début fct : a =", a)
    a = 3
    print("fin fct : a =", a)

a = 1
print("début programme principal : a =", a)
fct(a)
print("fin programme principal : a =", a)
```

```
début programme principal : a =
début fct : a =
fin fct : a =
fin programme principal : a =
```

Pour comprendre les scripts ci-dessus :

- Les variables qui sont définies à l'intérieur d'une fonction par une affectation (cas notamment des paramètres d'entrées) ne sont connues qu'à l'intérieur de cette fonction. Dès que l'on sort de la fonction, ces variables sont supprimées et le programme appelant n'y a pas accès. On dira que ces variables sont « locales » à la fonction ou qu'elles ont une portée « locale ». (cf script 1 et 3)
- Les variables définies dans le programme appelant restent visibles dans la fonction si elles ne sont pas « écrasées » par une variable locale du même nom. On dira que ces variables sont « globales ». (cf script 2)
- Mais dans le script 1, on rencontre d'abord l'instruction `a = 1` dans le programme appelant. La variable `a` est donc globale. Puis, quand on rencontre l'instruction `a = 3`, on est désormais dans la fonction. Cette instruction ne modifie donc pas la variable globale `a` précédente mais crée une nouvelle variable `a` qui sera locale.

II) Cas des variables mutables

Tester pas à pas le script ci-dessous, toujours avec Python Tutor ou le débugueur de Thonny.

script 4 : (avec Python Tutor)

```
def fct(a):
    print("début fct : a =", a)
    a[0] = 3
    a[1] = 4
    print("fin fct : a =", a)

a = [1, 2]
print("début programme principal : a =", a)
fct(a)
print("fin programme principal : a =", a)
```

```
début programme principal : a =
début fct : a =
fin fct : a =
fin programme principal : a =
```

Pour comprendre le script ci-dessus :

- L'instruction `a = [1, 2]` crée une variable globale `a` qui est passée en paramètre à la fonction. A l'intérieur de la fonction, `a` désigne une nouvelle variable qui est locale mais comme une liste est de type mutable, les deux variables `a`, celle qui est locale et celle qui est globale pointent vers la même liste. Les modifications faites sur cette liste à l'intérieur de la fonction, se répercutent donc dans le programme appelant ! C'est ce que l'on appelle un « effet de bord », et si on ne peut l'éviter, il faut au moins le préciser clairement dans la docstring de la fonction.

III) Bilan

Vous verrez sur Internet des scripts utiliser des variables globales dans des fonctions, mais c'est une habitude risquée et donc à éviter ! Quand on programme une fonction, la bonne pratique est donc de passer en paramètre toutes les variables dont on va avoir besoin dans la fonction, puis de retourner à la fin toutes les variables que la fonction a modifié. Le but est que la fonction soit complètement indépendante des variables du programme appelant.

Exemple avec un script qui dessine un rectangle de « O » :

Utilisation de variables globales : Très très mal !

```
def ligne():
    return "O"*L + "\n"

def rectangle():
    return ligne()*H

L = int(input("Largeur : "))
H = int(input("Hauteur : "))
print(rectangle())
```

Utilisation de paramètres : Bien ;-)

```
def ligne(largeur):
    return "O"*largeur + "\n"

def rectangle(largeur, longueur):
    return ligne(largeur)*longueur

L = int(input("Largeur : "))
H = int(input("Hauteur : "))
print(rectangle(L, H))
```

Dans les fonctions de la version de droite, toutes les entrées sont passées en paramètre et toutes les sorties sont dans le return.