Bases de Python

I. Constructions du langage Python

1. Affectations

Les instructions de la forme `variable = expression` sont des **affectations**.

Une telle instruction a le déroulement suivant :

1. l'expression est évaluée ;
2. la variable est affectée à la valeur obtenue.

Exemple

```
x = 2
# on affecte la variable x à la valeur 2
x = x + 1
# on évalue x + 1, qui vaut 3
# puis on affecte x à cette valeur
```

Une variable peut être utilisée pour stocker des valeurs qui varient au cours du temps, mais aussi pour décomposer un calcul de façon lisile. De manière générale, on essaiera de lui donner un nom qui indique son rôle.

Remarques

⋆ La notation n'est pas symétrique, à gauche on trouve la variable à affecter, et à droite l'expression à évaluer.
⋆ La notation n'a pas vraiment de rapport avec l'égalité en mathématiques.

2. Fonctions

Pour structurer son code et n'écrire qu'une fois une suite d'instructions que l'on veut exécuter plusieurs fois, en différents endroits.

2.1. Appel de fonction

Un **appel de fonction** se caractérise par l'**usage de parenthèses**[1] où l'on indique les arguments passés à la fonction. Mais on fait figurer des parenthèses même en l'absence d'arguments.

[1] En plus de l'utilisation usuelle en mathématiques.

```
f(2, 4) # appel de f avec deux arguments
g()     # appel de g sans argument
f       # pas de parenthèse,
        #     ce n'est pas un appel de fonction,
        #     mais la fonction elle-même
```

Si la fonction renvoie un résultat, il est en général intéressant de l'utiliser.

```
cos(1)      # pas très intéressant
x = cos(1)  # c'est mieux
autre_fonction("bonjour", cos(2))
            # on peut utiliser le résultat comme on veut
```

Remarques

⋆ Un appel de fonction déclenche l'exécution d'instructions. Cela consomme du temps.
⋆ On évitera donc de faire des appels de fonctions inutiles, comme par exemple pour calculer quelque chose… que l'on connaît déjà (ou que l'on pourrait déjà connaître).

2.2. Écriture de fonction

Définition La définition d'une fonction commence par **`def`**, suivi du nom et des *arguments* de la fonction puis de « : » qui annonce un bloc d'instructions, le *corps de la fonction*.

```
def f(a, b, c):      # déclaration de la fonction
  x = len(a) * b     # |
  c = c + 1          # | corps de la fonction
  return x - c       # |
```

Variables locales et arguments Toutes les variables utilisées pour une affectation dans le corps d'une fonction sont des variables locales.

Les arguments sont des variables locales particulières qui sont initialisées avec les valeurs passées lors de l'appel de fonction. Notons que ce sont des variables locales comme les autres, et que l'on peut modifier leur valeur.

Avec l'exemple précédent, lors de l'appel `f("Bonjour", 4, 24)`, le corps de la fonction est exécuté avec `a` qui est initialisé à `"Bonjour"`, `b` à `4` et `c` à `24`.

Caractère local des variables locales Les variables locales n'influent pas sur les variables déjà existantes. Leur existence et leurs valeurs sont *locales* à l'exécution de la fonction[2]. Dans le cas d'appels récursifs, chaque appel a son propre jeu de variables

[2] Heureusement, car sinon il serait difficile de contrôler ce qui se passe lors d'un appel de fonction.

locales.

```
>>> x = "Bonjour"
>>> a = 2
>>> f("Bonjour", 4, 2) # on va afficher le résultat renvoyé
25
>>> x                  # sa valeur n'a pas changé
'Bonjour'
>>> a                  # non plus
2
```

Valeur renvoyée L'exécution de l'instruction **`return`** (qui s'utilise sans parenthèse, ce n'est pas un appel de fonction) permet de renvoyer un résultat. En particulier, elle interrompt l'exécution de la fonction.

```
def fonction_sans_risque(x):
  suivant = x + 1
  return suivant
  truc_dangereux() # ne sera JAMAIS exécuté
```

3. Tests

L'instruction conditionnelle **`if`** est constituée d'une expression booléenne suivie d'un bloc d'instruction. Pour l'exécuter, on teste si l'expression est vraie (ou, plus précisément, on évalue l'expression et on teste si sa valeur est **`True`**) et si c'est le cas, on exécute le bloc d'instructions.

On peut ajouter, de façon **facultative**, une ligne **`else`**: suivi d'un bloc d'instructions destinées à être exécuté si le test est négatif (c'est-à-dire évalué à **`False`**).

```
if test:
  print("Le test est vrai.")
else:
  print("Le test est faux.")
```

Si l'on fait plusieurs tests à la suite, la construction **`elif`** `...`: est un raccourci pour **`else`**: **`if`** `...`:.

```
if n == 1:
  print("La valeur de n est 1")
elif n == 2:
  print("La valeur de n est 2")
else:
  print("La valeur de n est différente de 1 et de 2.")
```

Question 1 Quelle est la différence entre le test précédent et les suivants ?

```
if n == 1:
  print("La valeur de n est 1")

if n == 2:
  print("La valeur de n est 2")
else:
  print("La valeur de n est différente de 1 et de 2.")
```

4. Boucles

On dispose de deux instructions pour effectuer des boucles, c'est à dire répéter l'exécution d'instructions : **`for`** et **`while`**.

Les deux s'utilisent de manière similaire : on a une ligne d'entête terminée par « : » et suivi, les lignes suivantes, d'un bloc d'instructions indentées.

```
i = 0
while i > 1:
  i = (i * 9) // 10

for j in range(1, 10):
  print(j)
```

4.1. Boucles **`while`**

L'entête d'une boucle **`while`** comporte une expression booléenne (comme les **`if`**) qui sera évaluée à chaque itération avec le comportement suivant : si l'expression est vraie (ou, plus précisément, évaluée à **`True`**) alors le corps de la boucle est exécuté et recommence l'exécution de la boucle. Sinon, on arrête l'exécution de la boucle.

Pour éviter que la boucle ne se répète indéfiniment, il faut que la valeur de l'expression booléenne puisse changer de valeur. Cela implique la modification des valeurs des variables impliquées dans le test.

4.2. Boucles **`for`**

L'entête d'une boucle **`for`** est de la forme « **`for`** variable **`in`** itérable: ». Parmi les structures itérables, on rencontre fréquemment :

- les « range » de syntaxe `range(`*début*`,` *fin*`,` *pas*`)` (où les arguments *début* et *pas* sont facultatifs, le pas pouvant être négatif) qui parcourent toutes les valeurs commençant à *début* et s'arrêtant à *fin* (qui n'est pas traîtée) avec un pas de *pas* ;
- les tableaux,
- les chaînes de caractères,

⋆ les tuples,
⋆ les dictionnaires.

```
>>> for i in range(3, 0, -1):
...     print(i)
...
3
2
1
```

```
>>> for i in ["un", "deux", "trois"]:
...     print(i)
...
un
deux
trois
```

```
>>> for i in "abc":
...     print(i) # on itere lettre par lettre
...
a
b
c
```

```
>>> for i in (1, "a", True):
...     print(i)
...
1
a
True
```

```
>>> for cle in {"one": "un", "two": "deux", "three": "trois"}:
...     print(cle)  # on n'itère sur les clés
...
one
two
three
```

4.3. Sortie anticipée de boucle

Il est tout à fait possible d'utiliser un **return** à l'intérieur d'une boucle (si celle-ci est à l'intérieur du corps d'une fonction). Comme vu précédemment, l'exécution du **return** interrompt l'exécution de la fonction et donc, *a fortiori*, celle de la boucle.

Il est aussi possible d'interrompre une boucle à l'aide de l'instruction **break**.

```
>>> for i in range(10):
...     print(i)
...     if i == 4:
...         break
...
0
1
2
3
4
```

II. Types de données

Chaque donnée a un **type**, qui indique comme les utiliser et les manipuler : une donnée peut être un entier, une chaîne de caractère, un tableau de booléens, etc.

Le langage Python a un **typage dynamique** : chaque donnée a un type, mais celui-ci est déterminé *au dernier moment*, lors de l'exécution. Il ne peut donc dire à l'avance si une fonction est appelée avec un argument d'un mauvais type.

1. Types simples

Les types simples correspondent aux données de base, *atomiques*, que l'on ne peut décomposer. Les types simples sont principalement les nombres (entiers et flottants) et les booléens.

Nous parlerons des nombres dans un prochain chapitre, concentrons-nous sur les booléens.

1.1. Booléens

Il s'agit d'un type ayant deux valeurs possibles : **True** et **False**. Une façon usuelle d'obtenir un booléen est d'effectuer une comparaison :

```
>>> "bonjour" == "bonsoir"
False
>>> 1 <= 2
True
```

On note que la comparaison se note avec deux signes « égal » `==` alors que l'affectation se note avec un seul signe.

Les opérations sur les booléens sont la négation **not**, la conjonction **and** et la disjonction **or** :

```
>>> not True
False
>>> True and False
```

```
False
>>> True or False
True
```

L'évaluation de la conjonction et de la disjonction est paresseuse : on n'évalue le second opérande que si nécessaire.

```
>>> 1 / 0 == 3 # erreur
Traceback (most recent call last):
File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 1 == 1 or 1 / 0 == 3
True
```

C'est très utile pour les parcours de tableaux où l'on peut écrire des conditions de `while` de la forme :

```
while i < len(tab) and tab[i] != 2:
```

Notons enfin qu'un booléen est une donnée comme une autre, qui peut être manipulée, affectée à une variable, etc.

```
>>> x = 1 == 2
>>> x
False
>>> x == True
False
```

Question 2 Si x est une variable booléenne, déterminer la valeur de `x == True` en fonction de celle de x.

2. Types composés

Les principaux types composés que l'on va croiser sont les chaînes de caractères, les *tuples*, les tableaux et les dictionnaires.

Itérables Ces types sont *itérables* : on peut les utiliser dans une boucle `for`.

- ⋆ Pour une chaîne de caractères, on va parcourir les caractère un par un.
- ⋆ Pour un dictionnaire `d`, en faisant `for cle in d: ...`, on va parcourir les *clés* du dictionnaire. Pour parcourir les couples *clé*/*valeur*, on peut écrire

```
for cle, valeur in d.items(): ...
```

- ⋆ Pour les tableaux et les *tuples*, on visite bien sûr les éléments un par un.

Structure linéaire Avec les chaînes de caractères, les tableaux et les tuples, on peut accéder aux différents éléments de ces à l'aide de leur position (en partant de 0), indiquée entre crochets. Ce sont des structures *linéaires* :

```
>>> s = "bonjour"
>>> s[1]
'o'
```

Avec une structure linéaire, on peut extraire des données à l'aide des *slices* :

```
>>> s[2:5] # de l'indice 2 inclus à l'indice 5 exclu
'njo'
>>> s[2:]  # à partir de l'indice 2
'njour'
>>> s[:5]  # jusqu'à l'indice 5 exclu
'bonjo'
```

et on peut faire des concaténations à l'aide de l'opérateur `+` :

```
>>> "bon" + "jour"
'bonjour'
```

Les dictionnaires ne sont pas des structures linéaires, puisque les éléments sont indexés par des clés qui ne sont pas nécessairement des entiers.

Modification éventuelle du contenu

- ⋆ On ne peut pas modifier le contenu des chaînes de caractères et des *tuples*.

```
>>> s[1] = "a"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'str' object does not support item assignment
```

- ⋆ Les tableaux et dictionnaires peuvent, eux, être modifiés, à la fois en contenu et en structure.

```
>>> t = [1; 2; 3]
>>> t[1] = 4 # modification, contenu
>>> t
[1; 4; 3]
>>> t.append(5) # modification, ajout en queue
>>> t
[1; 4; 3; 5]
>>> t.pop() # modification, retrait en queue
5
>>> t
[1; 4; 3]
```