MEEF

Python

pour le professeur de mathématiques

Python

Centre Universitaire
MAYOTTE

*

Sommaire

Partie I

Les variables

i Il est souvent intéressant de pouvoir en conserver et réutiliser une valeur, par exemple le résultat d'un calcul. C'est le principe des variables : mémoriser une valeur.

Objectifs

- Déclarer et utiliser les variables
- Connaître les règles de nommage des variables

Affectation

En Python, les variables fonctionnent comme des étiquettes. Quand on assigne une valeur à une variable, on pose une étiquette avec un nom sur une valeur.

C'est-à-dire que la valeur existe quelque part en mémoire et qu'on vient lui attacher une étiquette. On peut aisément placer plusieurs étiquettes sur une même valeur, mais aussi retirer une étiquette pour la placer sur une autre valeur.

L'affectation se fait à l'aide du signe **=**

```
ma_variable = valeur
```

Mécanismes d'affectation

Affectation

```
1 a = 1
```

Réaffectation

```
a = 2
```

Double affectation

```
a = b
```

Source : http ://foobarnbaz.com/2012/07/08/understanding-python-variables/

Les types de variables

Il existe plusieurs type de variables. Parmi les plus courants, on retrouve :

Variable	Type	Remarques
`a = 2`	`int` : Les nombres entiers relatifs	
`a = 3.5`	`float` : Les nombres réels	Le séparateur est le . et non la ,
`a = "Bonjour"`	`str` : Les chaines de caractères	Entourer par des guillemets (`"`) ou des apostrophes (`'`) ou des triples guillemets (`"""`) ou des triples apostrophes (`'''`)
`a = True`	`bool` : Les booléens True ou Fasle	

Le nom des variables

Obligations :

- Ne pas contenir un signe opératoire + - * / %/ ni le caractères # (*Il est réserver aux commentaires*)
- Doit commencer par une lettre
- Ne doit pas être un mot clé Python (`if`, `in`, `as`, ...)

Préconisations :

- le nom décris le contenu variable et est constituer d'au moins 3 caractères.
- le nom est entièrement écrit en minuscules et les mots sont séparés par des espaces soulignés (underscores _). Exemple : ma_variable
- le nom doit permettre de comprendre le rôle de la variable, éviter les caractères spéciaux et accentuer

Sucre syntaxique

Comme dans d'autres langages certains opérateurs sont introduits pour raccourcir les processus de réaffectation des variables.

Exemple

Plutôt que d'écrire

```
1 i = i + 1
```

on préférera le raccourci suivant :

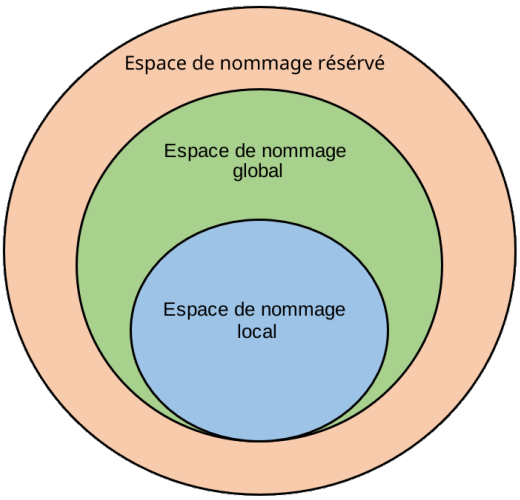
```
1 i += 1
```

Opérateur	Équivalent
a += b	a = a + b
a -= b	a = a - b
a *= b	a = a * b
a **= b	a = a ** b
a /= b	a = a / b
a //= b	a = a // b
a %= b	a = a % b

Espace de nommage

Quand vous faites référence à une variable, Python cherche la référence à l'objet correspondant dans l'espace de nommage. Un espace de nommage est donc simplement une collection de noms associés à des références d'objet. C'est ce qui fait la correspondance entre le nom de la variable et ce à quoi elle se réfère.
Il existe de nombreux espaces de noms :

- L'espace des nommage réservé de Python (built-in) : créé quand on lance l'interpréteur.
- L'espace de nommage global : créé quand on lance l'interpréteur.
- L'espace de nommage locaux : espace lié à un module, une fonction ou une classe



Partie II

Entrée - Sortie

i Dans cette partie, nous verrons comment :
- afficher du texte,
- permettre à l'utilisateur de saisir du texte dans le terminale.

• Sortie

•• Affichage du texte

En Python, pour afficher du texte, on utilise l'instruction **`print`**.
Si l'on veut afficher une chaine de caractère, on la placera entre guillemets :

Exemple

Pour afficher à l'écran le texte **`Bonjour`**.

Code

```
1 print("Bonjour")
```

Affichage

```
1 Bonjour
```

i En effet, **`print`** en anglais signifie « imprimer ».

Exemple

Chaque ligne du programme représente une instruction.

Code

```
1 print("Cours Python")
2 print("Pour  le prof de maths)
```

Affichage

```
    print("Cours Python")
    print("Pour  le prof de maths)
```

Formated string

Il peut être utile de placer un élément variable dans une chaine de caractère.
Par exemple, le résultat d'un calcul. Ainsi le code ci-dessous

Code

```
print(f"2 + 2 = {2+2}")
```

Affichage

```
2 + 2 = 4
```

L'instruction entre les accolades est évaluée avant l'affichage.

Code

```
nom = "John"
print(f"Bonjour {nom} !")
```

Affichage

```
Bonjour John !
```

Les erreurs

Quand on écrit un programme informatique, il faut être très rigoureux car un simple caractère mal placé peut perturber l'ordinateur du robot. Si le format de votre code n'est pas bon, il ne va pas aller plus loin et va simplement vous indiquer que vous avez commis une erreur. Quand ça arrive, il faut d'abord tenter de comprendre le message d'erreur et ensuite relire attentivement la zone de code concernée, en particulier pour s'assurer qu'il ne manque pas de symbole tel qu'une parenthèse ou un guillemet. Voici quelques exemples de messages d'erreur.
Attention aux parenthèses ! Si on oublie une parenthèse, cela produit une des erreurs suivantes :

```
    print("Bonjour"
    SyntaxError: unexpected EOF while parsing
```

```
    print "Bonjour")
    SyntaxError: invalid syntax
```

Attention aux guillemets !
Si on ne place pas correctement les guillemets, cela produit une des erreurs suivantes :

```
    print("Bonjour)
    SyntaxError: EOL while scanning string literal
```

```
    print(Bonjour)
    NameError: name 'Bonjour' is not defined
```

```
    print(Bonjour tout le monde)
    SyntaxError: invalid syntax
```

Entrée

Dans certains cas, il peut être nécessaire de demander des informations à l'utilisateur.

L'instruction `input`

L'instruction **`input`** permet à l'utilisateur de saisir des données au clavier.

Exemple

```
1 prenom = input("Entrer votre prénom :")
2 print(f"Bonjour {prenom}")
```

Si l'utilisateur saisie **John**, l'affichage sera :

```
1 Bonjour John
```

i La chaine de caractères donnée en argument de la fonction **`input`** (**`"Entrer votre nom:"`**) est facultative mais est utile à l'utilisateur pour savoir ce qu'on attend.

La conversion

i L'instruction **`input`** considère les entrées utilisateurs comme des chaines de caractères. Il peut donc être utile de changer le type de la variable.

Exemple

Exécutons le code suivant.

```
1 nombre = input("Entrer un nombre")
2 nombre = nombre * 5
3 print(nombre)
```

Si l'utilisateur entre **3**
L'affichage sera :

```
1 33333
```

? Pourquoi l'interpréteur affiche **33333** et non **15** comme attendu ?

L'interpréteur Python considérant la variable **`nombre`** comme une chaine de caractère, il va interpréter l'instruction **`nombre * 5`** comme **`répéter 5 fois la valeur de la variable nombre`**.
De fait si l'on veut que le programme fonctionne correctement, il faut donc changer le type de la variable. Dans notre exemple, nous allons utilisant l'instruction **`float`** pour transformer la saisie en nombre à virgule flottante. Ce qui donnerait :

```
nombre = float(input("Entrer un nombre"))
nombre = nombre * 5
print(nombre)
```

Voici donc différentes instructions permettant de convertir des variables.

Instruction	Description
int	convertir en nombre entier
float	convertir en nombre flottant
str	convertir en chaine de caractères

! Certaines conversions ne sont pas possibles.

Par exemple l'instruction `int("a")` renverra une erreur `ValueError: invalid literal for int() with base 10: 'a'`.

Partie III

Importation des modules et packages

i Les modules et les packages sont un ou des fichiers contenant un ensemble de fonctions, de classes et de variables prédéfinies

• Importation classique

L'importation d'un module de manière classique se fait par l'instruction d'importation.

Exemple

Par exemple le module `turtle`, on utiliser l'instruction `import turtle` Ensuite on peut utiliser les instructions du module `turtle` comme par exemple la fonction `forward`.

⚠ Avec ce type d'import, il faudra préfixer l'instruction du nom du module

```
1 import turtle
2
3 turtle.forward(100)
```

• Importation avec alias

Pour éviter de devoir préfixer toutes les instructions avec le nom du module, on peut utiliser un alias à l'aide de l'instruction `as`.On peut ainsi raccourcir le nom du module

Exemple

Toujours avec l'exemple du module `turtle`

```
1 import turtle as tl
2
3 tl.forward(100)
```

• Importation tout ou une partie d'un module

On peut aussi choisir d'importer seulement quelques instructions d'un module, dans ce cas on utilisera les instruction `from module import instruction`.

Exemple

Toujours avec l'exemple du module `turtle`

```
1 from turtle import forward
2
3 forward(100)
```

Il est aussi possible d'importer de cette façon toutes les instructions d'un module avec `*`.

⚠ Cette méthode n'est pas conseillée.

Exemple

```
1 from turtle import *
2
3 forward(100)
```

Partie IV

Les listes (type `list`)

Définition

Définition

En informatique, un tableau est une structure de données représentant une séquence finie d'éléments auxquels on peut accéder efficacement par leur position, nommée **indice**, dans la séquence.

i Python implemente les tableaux dans le type `list`.
De ce fait, les tableaux seront appellés des liste.

Objectifs

1. Créer un tableau vide ou non,
2. Déterminer la longueur d'un tableau : combien de valeurs comporte-t-il ?
3. Accéder à une valeur depuis son index.
4. Ajouter / supprimer un élément .
5. Parcourir les éléments

Représentation

En python, les tableaux sont contenus dans des crochets [...]
Les éléments sont séparés par des virgules ,

[élement1 , élement2 , élement3]

Initialisation d'un `tableau`

Tableau vide

Il est possible de créer un tableau vide

```
# Créer un tableau vide
mon_tableau = []
```

Tableau non vide

Il est aussi possible de l'initialiser avec des valeurs

```
# Créer un tableau à partir de valeurs
>>> mon_tableau = ["Alan", "Ada", "Grace", "Ian"]
["Alan", "Ada", "Grace", "Ian"]
```

Enfin si l'on veut, il est aussi possible de l'initialiser avec les mêmes valeurs

```
# Créer un tableau contenant 5 zéros
>>> mon_tableau = [0]*5
[0, 0 , 0, 0, 0]
```

Accéder à un élément

Pour accéder à un élément, on utilise son **`index`** (sa position dans la liste)

⚠ L'index d'une liste commence à 0.

`Tableau` : ["Alan", "Ada", "Grace", "Ian"]
indices : 0, 1, 2, 3

L'instruction pour accéder à un élément est construite à partir du nom du tableau et de l'index (*la position*) de l'élément souhaité entre crochets.

Exemple

Le tableau s'appelle **`mon_tableau`** et l'élément **`Ada`** est en position **1**.
On accède donc à cet élément avec l'instruction :
mon_tableau[1]

```
>>> mon_tableau = ["Alan", "Ada", "Grace", "Ian"]
>>> print(mon_tableau[1])
"Ada"
```

Les index négatifs

? Comment accéder aux derniers éléments d'un tableau

Pour atteindre le dernier élément d'un tableau, il faudrait connaître le nombre total d'éléments du tableau (ne utilisant la fonction `len`) et lui soustraire 1.

Exemple

On considère une liste `notes` qui contient les notes obtenues à différents contrôles par un élève. Pour afficher à la dernière note obtenue, on doit en pratique avoir le code suivant :

```
1 notes = [12, 14, 11, 16, 17, 15, 18, 15, 19, 14, 17]
2 nb_notes = len(notes)
3 print(notes[nb_notes - 1])
```

La variable `nb_notes` ne servant pas par la suite, on pourrait aussi écrire :

```
1 notes = [12, 14, 11, 16, 17, 15, 18, 15, 19, 14, 17]
2 print( notes[len(notes) - 1])
```

Pour alléger l'écriture, Python propose une raccourci, on peut écrire plus directement :

```
1 notes = [12, 14, 11, 16, 17, 15, 18, 15, 19, 14, 17]
2 print( notes[-1])
```

• Modification d'un élément

On peut modifier les éléments de la liste en affectant une nouvelle valeurs.

Exemple

```
1 >>> mon_tableau = ["Alan", "Ada", "Grace", "Ian"}
2 >>> mon_tableau[1] = "Ada Lovelace"
3 >>> print(mon_tableau)
4 ["Alan", "Ada Lovelace", "Grace", "Ian"]
```

• Ajouter des éléments

i Il existe plusieurs façons d'ajouter des éléments à un tableau donné.
On peut :

- ajouter un élement,
- étendre un tableau à partir d'un second tableaux,
- concatener 2 tableaux pour en former un troisième.

•• Ajouter un élément

Pour ajouter un **seul élément**, on utilise la méthode `append`. La méthode `append` modifie sur place l'objet tableau qui l'appelle.

```
mon_tableau = ["Alan", "Ada", "Grace", "Ian"]
mon_tableau.append("George")
print(mon_tableau)
["Alan", "Ada", "Grace", "Ian", "George"]
```

•• Ajout par extension de liste

Pour ajouter un tableau d'élément, on utilise la méthode **`extend`**. La méthode **`extend`** modifie sur place l'objet tableau qui l'appelle.

```
mon_tableau = ["Alan", "Ada", "Grace", "Ian"]
tableau_2 = ["Charles", "Guido"]
mon_tableau.extend(tableau_2)
print(mon_tableau)
["Alan", "Ada", "Grace", "Ian", "Charles", "Guido"]
```

•• Ajout par concaténation

Il est aussi possible de concaténer 2 listes pour en créer une troisième.

```
tableau_1 = ["Alan", "Ada", "Grace", "Ian"]
tableau_2 = ["Charles", "Guido"]
mon_tableau = tableau_1 + tableau_2
print(mon_tableau)
["Alan", "Ada", "Grace", "Ian", "Charles", "Guido"]
```

• Parcourir un tableau

Il existe deux manières de parcourir les éléments d'une liste :

- Parcourir une liste à partir des **`index`** des éléments
- Parcourir une liste par itération sur les éléments

•• Parcours à partir de l'index des éléments

i En python, les tableaux (list) fonction taille **len()** donnant le nombre d'éléments contenus dans la liste.
Associer à la fonction **`range()`**, on obtient un itérable composé de tous les index des éléments de la liste.

Exemple

```
mon_tableau = ["Alan", "Ada", "Grace", "Ian", "Charles", "Guido"]
for index in range(len(mon_tableau)):
    print(mon_tableau[index])
```

Dans cet exemple, la longueur de **`mon_tableau`** est 6.
L'instruction **`range(len(mon_tableau))`** va être successivement évaluée par **`range(6)`** puis par un itérable de longueur 6 qui contiendra les valeurs **`0, 1, 2, 3, 4`** et **`5`**.
Ainsi la variable index prendra successivement l'ensemble de ces valeurs.

•• Itération des éléments

i Comme pour tout les itérables, il est possible de parcourir un tableau en itérant les éléments du tableau, avec les instructions `for v in iterable`.

.

Exemple

```
1 mon_tableau = ["Alan", "Ada", "Grace", "Ian", "Charles", "Guido"]
2 for elmt in mon_tableau:
3     print(elmt)
```

Dans cet exemple, la variable `elmt` prendra successivement les valeurs contenues dans la variable `mon_tableau`, c'est à dire `"Alan"`, `"Ada"`, `"Grace"`, `"Ian"`, `"Charles"` et `"Guido"`

❝ Il est souvent utile de bien nommée la variable créée dans la boucle `for`. Ainsi pour le code précédent, il aurait été préférable d'écrire :

```
1 mon_tableau = ["Alan", "Ada", "Grace", "Ian", "Charles", "Guido"]
2 for informaticien in mon_tableau:
3     print(informaticien)
```

Partie V

Boucle

i Les boucles vont nous permettre de répéter un bloc d'instruction.
Il existe deux types de boucle :

- Les boucles itératives
- Les boucles conditionnelles

Les boucles itératives

On utilise les boucles itératives quand on peut déterminer à l'avance le nombre de répétitions nécessaires.
En python, ce type de boucle utilise un itérable.
A chaque itération, Python va attirbuer la valeur d'un élément de l'itérable à la variable de boucle à chaque fois que le corps de la boucle est exécutée.

Le bloc d'instructions à répéter est délimité par `l'indentation` (*un décalage de quatre espaces*).

Définition `itérable`

Un itérable est un type de données informatique qui peut être parcouru séquentiellement, c'est-à-dire qu'il est possible de parcourir chaque élément de l'itérable de manière séquentielle, un par un.

• L'instruction `for`

Les instructions "for" en Python permettent de répéter un bloc de code pour chaque élément d'un itérable.
Elles sont souvent utilisées pour parcourir des séquences de données, telles que des listes, les objets de type `range` ou des chaînes de caractères.

for variable **in** iterable **:**
    bloc d'instructions à répéter

Exemple

Voici un exemple d'utilisation de l'instruction **`for`** en Python :

Code

```
liste = [2, 3, 5, 7, 11, 13, 17]

# boucle qui parcourt chaque élément de l
for element in liste:
    # affichage de l'élément courant
    print(element)
```

Affichage

```
2
3
5
7
11
13
17
```

Dans cet exemple, la boucle **`for`** parcourt chaque élément de la liste nommée **`liste`** et affiche chaque élément.
La variable **`element`** prend successivement la valeur de chaque élément de la liste, et le bloc d'instruction (**`print(element)`**) à l'intérieur de la boucle est exécuté pour chaque élément.

• L'instruction `range`

Définition

`Objet de type range`

La fonction **`range()`** en Python permet de générer une séquence de nombres allant d'un début à une fin, avec un pas spécifié. Elle est souvent utilisée en conjonction avec les boucles **`for`** pour parcourir une plage de nombres.

! La borne de fin n'est pas inclue dans l'objet **`range`** produit

Exemple

Dans cet exemple, l'instruction **`range`** va générer un itérable contenant les nombre 0, 1, 2, 3 et 4.
La variable **`"i"`** prend successivement la valeur de chaque élément de l'objet **`range`**.

Code

```
# génération de la séquence de nombres de
for i in range(5):
    print(i)
```

Rendu

```
0
1
2
3
4
```

Partie VI

Les instructions conditionnelles

i Un instruction conditionnelle est composée d'un test puis d'un bloc d'instructions qui sera executé, ou non, en fonction de la validité du test.

• Syntaxe

En Python, le test commence par le mot clef **if** suivi d'une condition à valeur booléenne (`True` ou `False`) et se termine par le symbole **:**
Le bloc d'instructions qui suit s'exécute si et seulement si le test a pour valeur `True`

• 3 cas de figure

S'il n'y a qu'un seul cas à distinguer, on utilisera :

```
if condition:
    bloc_d_instructions
```

S'il n'y a que deux cas à distinguer, on utilisera le couple `if` et `else` :

```
if condition:
    bloc_d_instructions_1
else:
    bloc_d_instructions_2
```

S'il y a plus de deux cas, on utilisera `elif` pour ajouter des conditions.

```
if condition_1:
    bloc_d_instructions_1
elif condition_2:
    bloc_d_instructions_2
elif condition_3:
    bloc_d_instructions_3
```

• Opérateurs booléens

Il existe plusieurs type de variables. Parmi les plus courants, on retrouve :

Test	Sens	Remarques
a == 2	Test d'égalité	Le signe **=** étant réservé à l'affectation on utilise ici le double égal **==**
a > 3.5	supérieur	
a >= 3.5	supérieur ou égal	
a <	inférieur	
a <=	inférieur ou égal	
"B" in "Bonjour"	Test d'appartenance	Nécessite un itérable

Partie VII

Les fonctions

i Une fonction en informatique est une portion réalisant un tâche bien précise et qui pourra être utilisée une ou plusieurs fois dans la suite du programme.

Nous avons déjà rencontré diverses fonctions prédéfinies : `print()`, `input()`, `range()`

On verra que définir c'est propres fonction avec Python, c'est très simple. Voici deux exemples :

Fonction sans paramètre

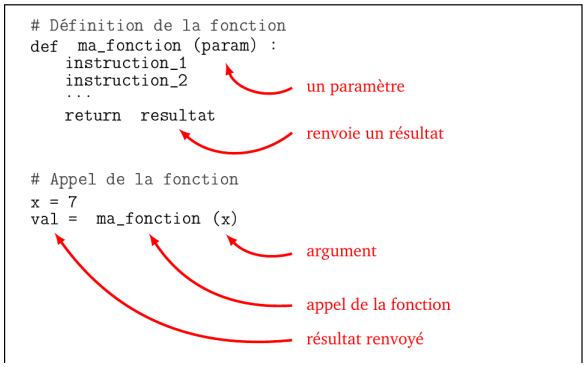
```
1 def dit_bonjour():
2     print("Bonjour le monde !")
```

Fonction avec paramètre

```
1 def carre(nombre):
2     return nombre**2
```

Définir une fonction

Pour définir une fonction on utilise le mot clé **`def`**.
Les instructions sont regroupées dans un bloc indenté.
Le mot **`return`** (optionnel) indique la fin de la fonction et précède les valeurs renvoyées par la fonction.



Source : Apprendre Python au lycée

• Appeler une fonction

⚠ Ces instructions ne sont exécutées que si j'appelle la fonction.

Par exemple, ce code n'affiche rien. Il sert uniquement à définir une fonction nommée `dit_bonjour`.

```
def dit_bonjour():
    print("Bonjour le monde !")
```

Pour exécuter la fonction, il faut l'appeler plus tard dans le programme.

```
dit_bonjour()
```

•• Paramètre optionnel valeur par défaut

Objectif : Permettre aux utilisateurs d'appeler une fonction en passant ou en omettant de passer les arguments relatifs aux paramètres possédant des valeurs par défaut.

Dans la déclaration d'une fonction, on peut préciser des valeurs par défaut pour nos paramètres.
Comme leur nom l'indique, ces valeurs seront utilisées par défaut lors d'un appel à la fonction si aucune valeur effective (si aucun argument) n'est passée à la place.

i On placera **toujours** les paramètres avec valeurs par défaut à la fin afin que le arguments passés remplacent en priorité les paramètres sans valeur.

```
import turtle as tl

def triangle(cote, couleur="red"):
    tl.begin_fill(couleur)
    tl.forward(cote)
    tl.left(120)
    tl.forward(cote)
    tl.left(120)
    tl.end_fill
```

Dans cet exemple :

- **triangle(100)** : produira un triangle **rouge** de 100 px de côté puisque rien .
- **triangle(200, "green")** : produira un triangle **vert** de 100 px de côté puisque la couleur est préciser par l'utilisateur.

•• Nombre arbitraire de paramètres

Objectif : Créer une fonction dont on ne connait pas le nombres de paramètres qui seront entrés par l'utilisateur.

La syntaxe ***args** (remplacez argements par ce que vous voulez) permet d'indiquer lors de la définition d'une fonction que la fonction peut accepter un nombre quelconque de paramètres.
Ces paramètres pourront être parcourue dans une boucle.

```
def somme(*args):
    res= 0
    for i in args:
        res += i
    return res
```

Dans cet exemple :

- **somme(1,2)** : renverra 3.
- **triangle(1,2,4,5)** : renverra 12