Variables et types de données

I. Les variables

1) Définition d'une variable

Définition 1 : *Variable*

Une variable est un espace mémoire élémentaire qui associe un nom (l'identifiant) à une valeur. Elle peut être modifiée au cours de l'exécution d'un programme. Une variable se caractérise par :

- ⋆ son nom : il sert d'identifiant
- ⋆ sa valeur : désigne l'information stockée dans la variable
- ⋆ son type : associé à la valeur, désigne la catégorie de donnée enregistrée dans la variable
- ⋆ sa portée : détermine jusqu'où l'on peut accéder à cette variable. Elle peut être de portée locale (variable définie dans une fonction) ou de portée globale (variable accessible dans tout le programme)

Remarque : Il existe certaines règles à respecter concernant le nommage des variables. Le nom d'une variable

- ⋆ ne doit contenir que des chiffres `0-9`, des lettres majuscules `A-Z`, des lettres minuscules `a-z` et le caractère underscore `_` (appelé aussi « tiret du 8 », situé sur la touche 8 du clavier français)
- ⋆ ne doit pas commencer par un chiffre `0-9`
- ⋆ ne doit pas être un mot réservé à Python (comme `for`, `if`, `while`, `def`, `True`, ...)

De plus, Python est sensible à la casse : les variables `Poisson` et `poisson` sont différentes.

Définition 2 : *Affectation d'une variable*

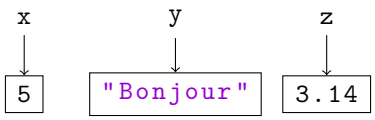
L'affectation d'une variable consiste à **créer** ou **modifier** la variable en lui attribuant une nouvelle valeur. La syntaxe d'affectation est la suivante :

```
nom_variable = valeur
```

On peut supprimer une variable de la mémoire avec la syntaxe : `del nom_variable`

```
x = 5
y = "Bonjour"
z = 3.14
```

FIGURE 1 – Visualisation de la mémoire après une affectation



Remarque : Contrairement aux mathématiques, le symbole `=` ne signifie pas une égalité, mais une affectation : on stocke dans `x` la valeur `5`.

L'ordre est important dans la syntaxe d'affectation : le nom de la variable doit être à gauche du `=` et la valeur à droite.

Variantes : affectation multiple et affectation simultannée

On peut définir plusieurs variables en même temps, en leur affectant **la même valeur** : on parle d'affectation multiple. Par exemple :

```
x = y = z = 0
```

On peut aussi définir plusieurs variables en même temps, en leur affectant **des valeurs différentes** : on parle d'affectation simultannée. Par exemple :

```
x, y, z = 1, 2, 3
```

2) Types de données

Les variables en Python sont typées c'est-à-dire qu'elles sont classées en différentes catégories en fonction des données qu'elles contiennent. Les types les plus courants sont :

Type	Nom	Exemples
`int`	Entiers	`4, 0, -5`
`float`	Flottants	`4.52, 2.0, -5.4`
`bool`	Booléens	`True, False`
`str`	Chaines de caractères	`"Bonjour !", 'Comment ça va ?'`
`list`	Listes	`[4, 3, -2, 2.0], ["a", "e", "i", "o", "u", "y"]`
`dict`	Dictionnaires	`{"pommes": 3, "poires": 2, "bananes": 0}`
`tuple`	Tuples	`(1, 2), (3, 2, 0)`

```
a = 10              # int
b = 2.5             # float
nom = "Alice"       # str
admis = True        # bool
L = [4, 2, 1]       # list
```

Remarque : La fonction `type` de Python permet de connaitre le type d'une variable :

```
>>> x = False
>>> type(x)
<class 'bool'>
```

Conversion de type

Pour chaque type de donnée, il existe une fonction de conversion permettant de transformer un objet quelconque en un objet de ce type (si c'est possible). La fonction de conversion porte le même nom que le type qui lui est associé : `int`, `float`, `bool`, `str`, `list`, `dict`, `tuple`.

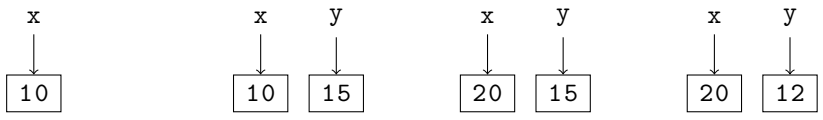
```
>>> int("32")
32
>>> int(2.72)
2
>>> str(32)
"32"
>>> float("3.14")
3.14
>>> list((1, 2, 3))
[1, 2, 3]
```

3) Ordre d'exécution et affectations successives

Lorsqu'un programme est exécuté, les instructions sont interprétées de haut en bas, ligne par ligne. L'ordre dans lequel on écrit les lignes est donc essentiel. Une variable peut être réaffectée : elle peut donc changer de valeur au cours d'un programme (d'où le nom de « variable »).

```
1 x = 10
2 y = x + 5
3 x = x * 2
4 y = y - 3
5 print(x)       # affiche 20
6 print(y)       # affiche 12
```

FIGURE 2 – Visualisation de la mémoire au cours du programme



Ligne 1 Ligne 2 Ligne 3 Ligne 4

Incrémentation d'une variable

Il existe des raccourcis de syntaxes pour modifier une variable. Par exemple, pour augmenter de 1 une variable numérique `x`, on peut écrire `x += 1` au lieu de `x = x + 1` : on parle d'incrémentation de la variable `x`.
Plus généralement, pour chacune des opérations sur les types numériques, on dispose d'un opérateur analogue :

Opérateur	Équivalent à
`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`

Problème : échange de variables

Un problème classique en programmation est celui de l'échange de deux variables. Étant donnés deux variables `a` et `b`, quelle suite d'instructions permet de mettre la valeur de `a` dans la variable `b` et la valeur de `b` dans la variable `a` ?

Le programme suivant ne fonctionne pas :

```
a = b        # a = 2 et b = 2
b = a        # cette ligne ne change rien
```

⋆ Solution 1 : Avec une variable auxiliaire

```
aux = a        # on retient la valeur de a avant de la changer
a = b          # on change a qui vaut maintenant 2
b = aux        # on change b qui vaut maintenant 1
```

⋆ Solution 2 : Avec une affectation simultanée

```
a, b = b, a
```

II. Les types numériques

1) Les entiers (`int`)

Définition 3

Le type entier se nomme `int` en Python (cela vient de « integer » en anglais qui signifie entier). Il permet de représenter (presque) tous les nombres entiers relatifs en les notant de la façon habituelle : `1`, `2`, `-5`, `0`.

Nous pouvons bien sûr demander au processeur d'effectuer des opérations simples sur les entiers : Python se comporte alors comme une calculatrice.

Les opérations possibles sur les entiers sont les opérations arithmétiques usuelles :

- ⋆ l'addition : $+$
- ⋆ la soustraction : $-$
- ⋆ la multiplication : $*$
- ⋆ la puissance : $**$ ou la fonction `pow`

```
>>> 1 + 1
2
>>> 4 - 1
3
>>> 32 * 3
96
>>> 3**3
27
>>> pow(4, 3)
64
```

Théorème 1 : *Division euclidienne*

Soit $a, b \in \mathbb{Z}$ avec $b \neq 0$. Il existe un unique couple d'entiers $(q, r) \in \mathbb{Z}^2$ tels que

$$\begin{cases} a = bq + r \\ 0 \leqslant r < |b| \end{cases}$$

Le nombre q est appelé le quotient de la division euclidienne de a par b et le nombre r est appelé le reste de la division euclidienne de a par b.

Nous pouvons rajouter aux opérations usuelles, les deux opérations **indispensables** suivantes :

- ⋆ la division entière ou quotient de la division euclidienne : $//$
- ⋆ le modulo ou reste de la division euclidienne : $\%$

```
>>> a = 43
>>> b = 10
>>> q = a // b
>>> q
4
>>> r = a % b
>>> r
```

```
3
>>> b * q + r
43
```

2) Les flottants (`float`)

Définition 4

Le type flottant se nomme `float` en Python. Il permet de représenter les nombres à virgule flottante, c'est-à-dire les nombres décimaux en les notant avec un point à la place de la virgule : `1.5`, `-3.25`, `2.0`

Les opérations possibles sur les flottants sont :

- ⋆ l'addition : $+$
- ⋆ la soustraction : $-$
- ⋆ la multiplication : $*$
- ⋆ la puissance : $**$ ou la fonction `pow`
- ⋆ la division : $/$

```
>>> 1.2 + 3.4
4.6
>>> 5.34 - 1.34
4.0
>>> 1.2 * 21.23
25.476
>>> 2.3**3.2
14.372392707920499
>>> pow(2.3, 3.2)
14.372392707920499
>>> 3.23 / 2.1
1.538095238095238
```

Remarque :

- ⋆ On peut bien entendu mélanger des entiers et des flottants dans une opération mais, dans ce cas, le nombre entier est automatiquement transformé en le nombre flottant qui lui correspond mathématiquement : par exemple `2` est transformé en `2.0`. Le résultat est donc toujours un flottant :
- ⋆
```
>>> 2 + 3.2
5.2
>>> 11 / 1.1
10.0
```

- ⋆ On ne peut absolument pas prévoir si le résultat d'un calcul sur les flottants sera exact ou approché. Même pour des calculs très simples ! Il faut donc toujours considéré le calcul comme approché si l'on ne veut pas avoir de surprise :

```
>>> 0.1 + 0.1 + 0.1 - 0.3
5.551115123125783e-17
```

- ⋆ L'opération division (/) peut aussi être utilisée avec des entiers uniquement. Dans ce cas aussi les nombres entiers sont convertis en flottants et le résultat est toujours un flottant et approché :

```
>>> 3 / 2
1.5
>>> 4 / 2
2.0
>>> 1 / 3
0.3333333333333333
```

Bonne pratique

Si on sait à l'avance qu'une division entre deux entiers doit donner un entier, alors il vaut mieux utiliser la division entière `//` que la division simple `/`. En effet, en utilisant `/`, on aura un flottant en résultat donc tous les calculs qui suivront sera approchés et non exacts !
Par exemple :

```
>>> a = 512
>>> b = a // 2      # est bien meilleur que b = a / 2
```

III. Les booléens (`bool`)

Définition 5

Le type booléen se nomme `bool` en Python. Il permet de représenter les valeurs de vérité : `True` (vrai) et `False` (faux).

Remarque : Il faut faire attention aux majuscules ! Écrire `true` à la place de `True` ne fonctionnera pas :

```
>>> true
Traceback (most recent call last):
    File "<console>", line 1, in <module>
NameError: name 'true' is not defined. Did you mean: 'True'?
```

On peut créer des expressions booléennes à l'aide des symboles :

Symbole Python	Signification	Mathématique
`==`	est égal à	$=$
`!=`	est différent de	$\neq$
`<=`	est inférieur ou égal à	$\leqslant$
`<`	est strictement inférieur à	$<$
`>=`	est supérieur ou égal à	$\geqslant$
`>`	est strictement supérieur à	$>$

et des connecteurs logiques :

Symbole Python	Signification	Mathématique
`and`	et	ET
`or`	ou	OU
`not`	négation	NON

```
>>> 1 + 1 == 2
True
>>> 1 + 1 != 2
False
>>> 4 <= 8
True
>>> 7 > 7
False
>>> 1 + 2 == 3 and 4 >= 3
True
>>> (1 + 2 + 3 == 6 or 4 == 8) and 1 > 0
True
>>> not(3.1 > 3)
False
```

Remarque :

⋆ Le symbole pour tester une égalité est == et non =. En effet, le symbole = sert à l'affectation d'une variable. Par exemple :

```
>>> x = 1     # Création de la variable x et affectation
>>> x == 2    # Test d'égalité, x vaut toujours 1
False
>>> x = 2     # Nouvelle affectation, x vaut maintenant 2
>>> x == 2    # Test d'égalité, x vaut toujours 2
True
```

⋆ Les exemples précédents sont des expressions booléennes. Il est possible de créer une variable booléenne en lui affectant le résultat d'une expression booléenne. Par exemple :

```
1 x = 12
2 y = x**2
3 test = (y - 3 * x > 0) and (y % x == 0)     # test = True
```

IV. Les types conteneurs

En Python, les chaines de caractères (`str`), les listes (`list`), les dictionnaires (`dict`) et les tuples (`tuple`) sont des conteneurs : ils permettent de regrouper plusieurs données en une seule variable. Ils ont donc des points communs mais également des différences (sinon il n'y aurait qu'un seul type).

Chacun de ces types de données (à l'exception des tuples) fera l'objet d'un chapitre dédié. Dans cette

partie, nous allons brièvement voir les points communs des types conteneurs.

Points communs des conteneurs

⋆ Chacun de ces types a un délimiteur qui lui est propre :

- Les chaines de caractères sont délimitées par des guillemets : `""` ou `''`
- Les listes sont délimitées par des crochets : `[]`
- Les dictionnaires sont délimités par des accolades : `{}`
- Les tuples sont délimités par des parenthèses : `()`

⋆ La fonction `len` permet d'obtenir le nombre d'éléments de l'objet :

```
>>> len("Bonjour !")
9
>>> len(["a", "e", "i", "o", "u", "y"])
6
>>> len({"pommes": 3, "poires": 2, "bananes": 0})
3
>>> len((1, 2, 3, 4))
4
```

⋆ Chacun de ces types est indexable : on peut accéder à un élément de l'objet à l'aide d'un indice ou d'une clé (pour les dictionnaires).

```
objet[indice/clé]
```

⋆ Chacun de ces types est itérable : on peut parcourir ses éléments à l'aide d'une boucle `for`.

```
for e in objet:
    ...
```

V. Scripts

1) Définition

Définition 6 : *Script*

Un script est un programme, écrit dans l'éditeur, qui, une fois exécuté, réalise une tâche précise. En général, un script commence par demander à l'utilisateur du programme certaines informations nécessaires à la réalisation de la tâche, puis réalise un certain nombre de calculs pour enfin afficher le résultat sous forme de phrase.

Schéma d'un script

```
## Initialisation
   # demande d'informations à l'utilisateur à l'aide de la fonction input

## Corps du programme :
   # affectations
   # instructions conditionnelles
   # instructions itératives

## Conclusion :
   # affichage des résultats à l'aide de la fonction print
```

2) La fonction `input`

La fonction `input` est une fonction qui permet de demander une information à l'utilisateur d'un programme. Cette information **doit toujours** être enregistrée dans une variable car il est complètement inutile de demander quelque chose à l'utilisateur si on ne retient pas sa réponse. Cette variable sera alors par défaut de type chaine de caractères (`str`) et ce quelle que soit la réponse de l'utilisateur.

```
>>> input("Quel est votre nom ? ")
Quel est votre nom ?
```

Le programme est en pause : Python attend une réponse à la question.

```
Quel est votre nom ? Paul
'Paul'
```

La réponse n'a pas été enregistrée ! Il faut toujours stocker la réponse dans une variable :

```
>>> nom = input("Quel est votre nom ? ")
Quel est votre nom ? Paul
>>> nom
'Paul'
>>> type(nom)
<class 'str'>
```

Ainsi, la syntaxe à retenir pour demander une information à l'utilisateur est

```
reponse = input(question)
```

où `question` est une chaine de caractère contenant la consigne indiquant à l'utilisateur ce qu'il doit écrire et `reponse` est une variable de type `str` qui enregistrera la réponse de l'utilisateur.

Quelle que soit la réponse de l'utilisateur, celle-ci est enregistrée par Python sous la forme d'une **chaine de caractères** ! Par exemple :

```
>>> age = input("Quel est votre age ? ")
Quel est votre age ? 18
>>> age
'18'
>>> type(age)
<class 'str'>
```

Une bonne pratique consiste à convertir **immédiatement** la variable en `int` ou en `float` lorsqu'on attend un nombre :

```
>>> age = int(input("Quel est votre age ? "))
Quel est votre age ? 18
>>> age
18
>>> type(age)
<class 'int'>
>>> taille = float(input("Quel est votre taille ? "))
Quel est votre taille ? 1.72
>>> taille
1.72
>>> type(taille)
<class 'float'>
```

3) La fonction `print`

La fonction `print` est une fonction qui permet d'afficher une chaine de caractères ou la valeur d'une variable. Il existe beaucoup de façons différentes de l'utiliser. En voici quelques unes :

```
>>> print("Bonjour")
Bonjour
>>> x = 33
>>> print("x")              # Affiche de la lettre x
x
>>> print(x)                # Affiche la valeur de la variable x
33
>>> print(2*x)
66
>>> print("J'ai", x, "euros.")
J'ai 33 euros.
```

Le dernier exemple est une phrase dynamique : la phrase affichée change en fonction de la valeur de la variable `x`. Par opposition, une phrase statique est une phrase qui sera affichée telle qu'elle est écrite.

Afficher une phrase dynamique

Imaginons que nous ayons trois variables `nom = "Hugo"`, `prenom = "Victor"` et `age = 83` et que nous voulions afficher la phrase `"Bonjour Victor Hugo. Tu as 83 ans."`.
Voici quatre solutions (la dernière étant la plus simple d'utilisation) :

- ⋆ **Utiliser plusieurs paramètres dans la fonction `print` :**

```
>>> print("Bonjour", prenom, nom, "! Tu as", age, "ans.")
Bonjour Victor Hugo. Tu as 83 ans.
```

Python met automatiquement des espaces entre les différentes parties. Pour modifier ce comportement, on peut ajouter un argument optionnel à la fonction `print` :

```
>>> print("Bonjour", prenom, nom, "! Tu as", age, "ans.", sep='')
BonjourVictorHugo! Tu as83ans.
>>> print("Bonjour", prenom, nom, "! Tu as", age, "ans.", sep='-')
Bonjour-Victor-Hugo-! Tu as-83-ans.
```

- ⋆ **Utiliser le typage (conversion de type) et la concaténation :**

```
>>> print("Bonjour " + prenom + nom + " ! Tu as " + str(age) + " ans.")
Bonjour Victor Hugo. Tu as 83 ans.
```

Nous devons alors gérer nous même les espacements entre les morceaux au risque d'avoir un résultat qui n'est pas satisfaisant :

```
>>> print("Bonjour" + prenom + nom + "! Tu as" + str(age) + "ans.")
BonjourVictorHugo! Tu as83ans.
```

- ⋆ **Utiliser la fonction `format` :**

```
>>> print("Bonjour {} {} ! Tu as {} ans.".format(prenom, nom, age))
Bonjour Victor Hugo. Tu as 83 ans.
```

Les accolades représentent les endroits où les valeurs des variables vont être placées. Entre les parenthèses après `format`, on écrit le nom de ces variables dans l'ordre d'apparition dans la phrase.

- ⋆ **Utiliser les f-string (sous Python 3.6 et supérieur) :**

```
>>> print(f"Bonjour {prenom} {nom} ! Tu as {age} ans.")
Bonjour Victor Hugo. Tu as 83 ans.
```

Ce qui est entre accolades ne sera pas affiché directement mais évalué puis affiché. C'est la façon la plus simple et logique de procéder.

Remarque : On peut encore améliorer les exemples précédents en remarquant qu'on n'est pas sûr que `age` soit supérieur ou égal à `2` auquel cas on aura une faute d'orthographe au mot `"ans"`. Pour résoudre ce problème, on peut définir une variable `s` qui vaudra `"s"` si `age` est au moins égal à `2` et qui vaudra `""` sinon :

```
nom = input("Entrez votre nom : ")
prenom = input("Entrez votre prénom : ")
age = int(input("Entrez votre âge : "))

if age >= 2:
    s = "s"
else:
    s = ""
print(f"Bonjour {prenom} {nom} ! Tu as {age} an{s}.")
```

puis après exécution :

```
Entrez votre nom : Hugo
Entrez votre prénom : Victor
Entrez votre âge : 1
Bonjour Victor Hugo ! Tu as 1 an.
```

Alors qu'il est **indispensable** d'enregistrer le résultat de la fonction `input` dans une variable, il ne faut **surtout pas** le faire avec la fonction `print`. En effet, cette fonction se contente d'afficher quelque chose à l'écran mais **elle ne renvoie aucune valeur** !
Voici ce qu'il se passe si on commet cette erreur :

```
>>> a = 12
>>> b = print(a)  # Pas d'erreur : on a l'impression que b = 12
12
>>> b             # Pas d'erreur mais aucune valeur n'est affichée
>>> type(b)       # b est une variable de type NoneType : elle est vide
<class 'NoneType'>
```

La variable `b` ne vaut pas 12 : elle ne vaut rien et n'a même pas de type. Faire un calcul avec `b` renverra donc une erreur.