Programmation avec Python

Spécialité numérique et sciences informatiques en classe de première

Lycée Rabelais

Saint Brieuc
2023-2024

Partie I
Programmation avec Python

Chapitre 1

Valeurs et types

1 Python, machine à évaluer

Python est en premier lieu une machine à calculer, ou plutôt une machine à *évaluer* : lorsque Python rencontre une *expression*, c'est-à-dire une écriture qui produit une valeur, il commence par déterminer cette valeur.
Pour s'en rendre compte, il suffit d'écrire des expressions dans une *console*.

Python

```
>>> 5 - 3
2

>>> 11 / (1 + 2)
3.6666666666666665

>>> 30 / 15
2.0
```

On se rend compte que les valeurs rencontrées ne sont pas présentées de la même manière :
`5 - 3` a la valeur `2` alors que `30 / 15` a la valeur `2.0`.
Pour y voir plus clair, on peut appeler la fonction `type` qui

- en entrée prend une expression ;
- renvoie le *type* de l'expression.

Python

```
>>> type(2)
<class int>

>>> type(2.0)
<class float>
```

Il y a donc au moins deux types de valeurs, le type `int` et le type `float`.
En fait il existe une multitude de types prédéfinis selon la nature de la valeur à représenter et nous allons les passer en revue.

2 Le type int

Il sert à représenter les *entiers relatifs* (*integer* signifie « entier » en Anglais).
Le type `int` dispose des opérations `+` (addition), `-` (soustraction) et `*` (multiplication).

Python

```
>>> 3 + 2
5

>>> 2 * 3
6

>>> 3 - 2 * 2
-1

>>> 10_000 # on utilise des _ pour séparer les
 ↪  chiffres
```

Les parenthèses sont utilisées comme en mathématiques, pour indiquer une *priorité opératoire*. Cependant les crochets et les accolades sont réservés à un autre usage.

Python

```
>>> (3 + 4) * 5
35
```

On dispose également de *deux opérations très pratiques* : soient `a` et `b` deux `int`, et `b` non nul, alors on

- `a // b` est le *quotient* de la *division euclidienne* de `a` par `b` ;
- `a % b` est le *reste*.

Python

```
>>> 64 // 10 # 64, c'est 6 * 10 + 4
6

>>> 64 % 10
4

>>> 22 // 7 # 22, c'est 3 * 7 + 1
3

>>> 22 % 7
1
```

On dispose de l'opération d'*exponentiation* (opération puissance), notée `**`.
Attention : Cette opération peut produire un résultat *non-entier*, de type `float` (voir partie suivante).

Python

```
>>> 2 ** 3
8

>>> 10 ** 4
10000
```

```
>>> 2 ** (-1)
0.5
```

Pour finir, la *division décimale* peut être effectuée sur des entiers, mais elle renvoie un résultat de type `float`.

3 Le type float

Il sert à représenter les *nombres à virgule flottante* (*to float* : flotter en Anglais). Ce sont (en gros) des nombres décimaux. PYTHON comprend et utilise la notation scientifique : `2.35e6` vaut $2,35 \times 10^6$, c'est-à-dire 2 350 000.

Python

```
>>>  2 / 7
0.2857142857142857 # c'est une valeur approchée

>>> 1 / 100_000
1e-05

>>> 1.2e-4
0.00012
```

On peut pratiquer sur les `float` toutes les opérations vues avec les `int`. Pour des fonctions plus compliquées telles le cosinus ou l'exponentielle, on fait appel au module[1] `math` :

Python

```
>>> from math import *
>>> pi
3.141592653589793
```

[1] Un module est un ensemble de *fonctions* et/ou de *constantes* que l'on peut importer.

```
>>> cos(pi / 3)
0.5000000000000001

>>> exp(2)
7.38905609893065

>>> log(2)
0.6931471805599453

>>> exp(log(2))
2.0
```

`exp` est la *fonction exponentielle* et `log` la *fonction logarithme népérien*[2] notée ln en France.

4 Le type str

Il sert à représenter les *chaînes de caractères* (`str` est l'abréviation de *string*, qui veut dire chaîne en anglais). Lorsqu'on écrit une valeur de type `str`, on peut utiliser les symboles `'`, `"` ou même `'''` (suivant que la chaîne contient des apostrophes, ou des guillemets).

Python

```
>>> 'Bonjour.'
'Bonjour'

>>> 'J'aime Python.'
SyntaxError

>>> "J'aime Python."
```

[2] Voir le programme de mathématiques de terminale scientifique.

```
"J'aime Python."

>>> "Je n'aime pas qu'on m'appelle "geek"."
SyntaxError

>>> """Je n'aime pas qu'on m'appelle "geek"."""
'Je n\'aime pas qu\'on m\'appelle "geek".'
```

La dernière évaluation produit une valeur correcte. PYTHON utilise simplement `\'` pour écrire les apostrophes qui sont à l'intérieur de la valeur.

Le symbole `+` sert à *concaténer 2 chaînes*, c'est-à-dire à les mettre bout à bout.

Python

```
>>> 'Yes' + 'No'
'YesNo'

>>> 'No' + 'Yes'
'NoYes'
```

On peut même multiplier un `str` par un `int` :

Python

```
>>> 3 * 'Aïe ! '
'Aïe ! Aïe ! Aïe ! '
```

PYTHON évalue `3*'Aïe !'` comme `'Aïe ! ' + 'Aïe ! ' + 'Aïe ! '`.
Voici deux types très utiles que nous étudierons en détail plus tard.

5 Le type list

Une valeur de type `list` est une… liste ordonnée de valeurs. Celles-ci peuvent être du même type ou non.

Python

```
>>> [] # liste vide
[]

>>> [1, 4, 5] # liste comportant 3 int
[1, 4, 5]

>>> [2.0, -4, 'Bonjour', 'Coucou']
[2.0, -4, 'Bonjour', 'Coucou']
```

L'intérêt de ce type est de rassembler plusieurs valeurs au sein d'une seule, qui pourra ensuite être *parcourue*.

6 Le type dict

Celui-ci sert à établir des *associations* du type *clé : valeur*.

Python

```
>>> {} # dictionnaire vide
{}

>>> {'France': 'Paris', 'Canada': 'Ottawa',
 ↪  'Pays-Bas': 'La Haye'}
{'France': 'Paris', 'Canada': 'Ottawa', 'Pays-Bas':
 ↪  'La Haye'}
```

Tout comme le type `list`, ce type sert à *structurer les données*. L'exemple précédent fait correspondre des capitales à des pays.

7 Le type bool

Il sert à représenter les *valeurs booléennes*, valant `True` (vrai) ou `False` (faux).

Python

```
>>> False
False

>>> True
True
```

Cela peut paraître un peu pauvre, c'est trompeur : les *expressions logiques* sont des écritures dont la valeur est un booléen. Lorsque PYTHON les rencontre, il les évalue pour trouver soit `True` soit `False`.

Python

```
>>> 3 >= 2 # évalue si 3 est supérieur ou égal à 2
True

>>> 3 + 5 == 2 # évalue si 3 + 5 vaut 2
False

>>> 1 in [3, 4, 1, 5] # évalue si 1 est un élément de
 ↪  la liste
True
```

Attention

Pour *tester* si deux valeurs sont égales, on utilise `==` (et pas `=`).

Ce type dispose d'*opérations logiques* : `or` (ou), `and` (et) et `not` (non).

Python

```
>>> True and False # vrai que si les 2 sont vrais
False

>>> True or False # faux que si les 2 sont faux
True
```

```
>>> not (3 < 1) # contraire
True

>>> (2 < 1) or (3 >= 0)
True
```

Chapitre 2

Variables et affectations

1 Le symbole =

En mathématiques, le symbole $=$ a plusieurs significations :

- dans $2 + 2 = 4$, on peut comprendre $=$ comme un opérateur d'évaluation : $2 + 2$, cela « donne » 4 ;
- dans $\mathcal{P} = 2 \times (\ell + \mathrm{L})$, on peut considérer que $=$ sert à définir ce qu'est le périmètre d'un rectangle de dimensions ℓ et L ;
- dans $3x + 2 = 4x + 5$, le $=$ sert à convenir que les 2 membres ont la même valeur et on cherche s'il existe un ou des nombres x qui satisfont l'égalité (appelée équation) ;
- *et cætera*.

En Python, le symbole = n'a qu'un seul sens : il sert à l'*affectation*.

2 L'affectation

Il s'agit de « stocker » une valeur dans un endroit de la mémoire auquel Python donne un nom[1]. Voici un exemple d'affectation :

```
a = 2
```

[1] En réalité c'est plus compliqué mais cela ne nous intéresse pas pour le moment.

- `2` est créee en tant que *valeur* de type `int` ;
- la *variable* `a` est créée ;
- `a` est « attachée » à la valeur `2` ;
- par extension `a` est également de type `int`.

Au cours d'un programme la valeur associée à une variable peut changer…D'où le nom de *variable*.

Définition : affectation

> Lors d'une affectation
>
> - d'abord PYTHON évalue ce qu'il y a à droite du symbole `=` ;
> - si cette valeur n'existe pas déjà en mémoire, elle est créée ;
> - ensuite il affecte cette valeur à la variable qui figure à gauche du symbole `=` ;
> - si la variable n'existe pas déjà, elle est créée automatiquement ;
> - le type de la variable, c'est le type de la valeur qu'on lui affecte.

Que fait le programme suivant ?

Python

```
x = 0
x = x + 1
print(x)
```

- il crée une variable `x` de type `int` valant `0` ;
- il évalue `x + 1`, trouve `1` et affecte cette valeur à `x` ;
- évalue `x`, trouve 1 et donc affiche `1`.

À retenir

En mathématiques, $x = x + 1$ est une équation sans solution.

En PYTHON, l'instruction `x = x + 1` sert à augmenter la valeur de `x` de 1 (on dit aussi *incrémenter*).

2.1 Affectations multiples

PYTHON permet d'affecter plusieurs valeurs à plusieurs variables en même temps.

Python

```
>>> a, b = 10, 2
>>> a
10

>>> b
2

>>> a, b = b, a # permet d'échanger a et b
>>> a
2

>>> b
10
```

2.2 Notation condensée

On est souvent amené à écrire des instructions telles que `a = a + 1` ou `b = b / 2`. Cela peut être lourd quand les variables ne s'appellent pas `a` ou `b` mais `rayon_sphere` ou `largeur_niveau`. On peut utiliser les notation suivantes :

Python

```
>>> rayon_sphere = 3.4
>>> rayon_sphere /= 2 # rayon_sphere = rayon_sphere /
 ↪  2
>>> rayon_sphere
1.7

>>> largeur_niveau = 19
>>> largeur_niveau += 1 # largeur_niveau =
 ↪  largeur_niveau + 1
>>> largeur_niveau
20
```

On dispose également de `*=`, `//=`, `%=`, `-=` et `**=`.

3 Le cas des variables de type str ou list

3.1 Les str

Les valeurs de type `str` sont composées de caractères *alphanumériques*. On peut accéder à chacun d'eux de la manière suivante :

Python

```
>>> chaine = 'Bonjour !'
>>> chaine[0]
'B'
>>> chaine[5]
'u'
```

Voici comment PYTHON représente la chaîne précédente :

i	0	1	2	3	4	5	6	7	8
chaine[i]	B	o	n	j	o	u	r		!

On a parfois besoin de connaître la longueur (*length* en anglais) d'une chaîne de caractères :

Python

```
>>> chaine = 'onzelettres'
>>> len(chaine)
11
```

On peut aussi accéder facilement au dernier (ou à l'avant dernier) caractère d'une variable de type `str` :

Python

```
>>> a = "M'enfin ?!"
>>> a[-1]
'!'

>>> a[-2]
'?'
```

3.2 Les list

Cela se passe un peu comme pour les `str`.

Python

```
>>> lst = [3, 4, 8]
>>> lst[1] # élément d'indice 1 de lst
4

>>> lst[-1] # dernier élément de lst
8

>>> len(lst) # longueur de la liste
3
```

Lorsqu'on essaie d'accéder à un élément dont l'indice est supérieur ou égal à la longueur de la liste, on obtient une erreur. Dans l'exemple précédent si on évalue `lst[3]` on obtient :

```
IndexError: list index out of range
```

Un chapitre est consacré à l'étude détaillée des `list`.

Chapitre 3

Entrées-sorties et transtypage

1 Entrées-sorties

Lorsqu'on écrit un programme en PYTHON, on a très souvent besoin

- qu'il affiche des informations à l'écran ;
- qu'il demande des informations à l'utilisateur·trice.

1.1 la fonction print

C'est elle qui sert à afficher à l'écran :

Python

```python
print("Bonjour") # affiche bonjour
print(4) # affiche 4
print(2 + 4) # évalue 2 + 4 et affiche 6
a = 3
print(a) évalue a et affiche 3
```

On peut s'en servir de diverses manières :

Python

```python
age = 16
print("Vous avez", age, "ans.")
```

Dans ce premier cas, on a donné 3 paramètres à `print` :

- le `str "Vous avez"` ;
- la variable `age` ;
- le `str "ans."`.

On peut aussi utiliser ce qu'on appelle une « f-string » de la manière suivante :

Python

```
age = 16
print(f"Vous avez {age} ans.") # noter le f au début
```

Dans ce cas, PYTHON construit le `str` à afficher en y reportant la valeur de `age`. C'est cette méthode que nous utiliserons par la suite.

Enfin, `print` revient à la ligne par défaut, mais on peut changer cela :

Python

```
print("Bonjour",end="") # aucun retour à la ligne
print("à tous",end="!") # aucun retour et ! à la fin
```

Ce programme affiche `Bonjour à tous!` sans revenir à la ligne.

1.2 La fonction input

C'est elle qui est chargée de récupérer des informations que l'utilisateur·trice entre au clavier.

Python

```
nom = input("Entrez votre nom : ")
```

Le programme précédent

- affiche `Entrez votre nom :` ;
- attend que l'utilisateur·trice tape quelque chose, puis valide avec la touche d'entrée ;

– stocke le message tapé dans la variable `nom`.

Important

> La valeur renvoyée par la fonction `input` est *toujours* de type `str`. Cela va donc nous amener à faire du *transtypage*.

2 Transtypage

Définition : transtypage

> C'est l'action de changer le type d'une valeur ou d'une variable en un autre type *compatible*.

Voici un exemple :

Python

```
a = 2 # a est un int
b = float(a) # la valeur 2 est convertie en float et
 ↪  stockée dans b
print(b) # affiche 2.0
```

En voici un autre dont nous nous servirons beaucoup

Python

```
a = '12' # a est un str
b = int(a) # '12' est converti en l'entier 12
print(b + 1) # évalue b + 1 à 13 et affiche 13
```

Attention

> Le transtypage n'est pas une action anodine et peut produire des erreurs lorsque la valeur à transtyper est incompatible avec le nouveau type qu'on veut lui donner.

Python

```
a = "Salut"
b = int(a) # impossible à réaliser

ValueError: invalid literal for int() with base 10:
'Salut'
```

Par ce message d'erreur, PYTHON nous indique que la valeur `'Salut'` ne peut pas être envisagée comme l'écriture en base 10 d'un entier.

3 Bilan

Dès l'écriture de programmes simples, le recours aux fonctions `print`, `input` et au transtypage sont inévitables, comme le montre le programme suivant :

Python

```
age = input("Entrez votre âge : ") # on obtient un str
age = int(age) # on convertit age en int
nouveau = age + 10 # qui nous permet de calculer
print(f"Dans 10 ans, vous aurez {nouveau} ans.")
```

Il existe d'autres actions de transtypage que `str` → `int` :

- `int` → `str`;
- `float` → `int`, qui induit souvent une perte d'information car la partie décimale de la valeur disparaît;
- `str` → `list` qui transforme par exemple `"abc"` en `["a", "b", "c"]`.

Chapitre 4

Tests et conditions

« Ceci n'est pas un test ! »

1 Des outils pour comparer

Ce sont les *opérateurs de comparaison* :

Opérateur	Signification	Remarques
<	strictement inférieur	Ordre usuel sur `int` et `float`, lexicographique sur `str`…
<=	inférieur ou égal	Idem
>	strictement supérieur	Idem
>=	supérieur ou égal	Idem
==	égal	« avoir même valeur » *Attention :* deux signes =
!=	différent	
`is`	identique	être le même objet
`is not`	non identique	
`in`	appartient à	avec `str`, `list` et `dict`
`not in`	n'appartient pas à	avec `str` et `list` et `dict`

Python

```
>>> a = 2 # crée une variable de type int avec la
 ↪  valeur 2
>>> a == 2 # a vaut-elle 2 ?
```

```
True

>>> a == 3 # a vaut-elle 3 ?
False

>>> a == 2.0 # a vaut-elle 2.0 ?
True

>>> a is 2.0 # a est-elle la valeur 2.0 ?
False

>>> a != 100 # a est-elle différente de 100 ?
True

>>> a > 2 # a est-elle supérieure à 2 ?
False

>>> a >= 2 # a est-elle supérieure ou égale à 2 ?
True
```

Python

```
>>> a = 'Alice'
>>> b = 'Bob'
>>> a < b # a est il avant b dans l'ordre
 ↪  lexicographique ?
True

>>> 'ce' in a # 'ce' est-il une sous-chaîne de 'Alice'
 ↪  ?
True

>>> 'e' in b # 'e' est-il une sous-chaîne de 'Bob' ?
```

```
False

>>> liste = [1, 10, 100]
>>> 2 in liste # 2 est-il un élément de liste ?
False
```

Ces opérateurs permettent de réaliser des tests basiques. Pour des tests plus évolués on utilisera des « mots de liaison » logiques.

2 Les connecteurs logiques

- `and` permet de vérifier que 2 conditions sont *vérifiées simultanément*.
- `or` permet de vérifier qu'*au moins une* des deux conditions est vérifiée.
- `not` est un opérateur de *négation* très utile quand on veut par exemple vérifier qu'une condition est fausse.

Voici les tables de vérité des deux premiers connecteurs :

and	True	False
True	True	False
False	False	False

or	True	False
True	True	True
False	True	False

À ceci on peut ajouter que `not True` vaut `False` et vice-versa.

Python

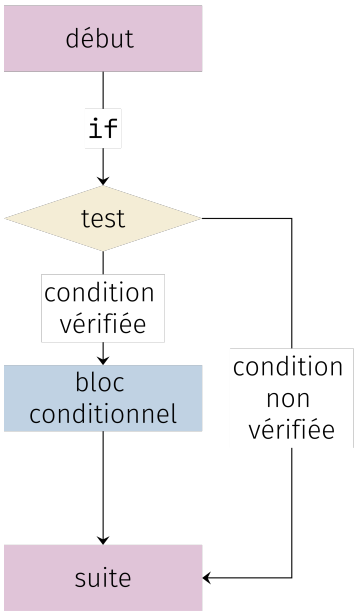
```
>>> True and False
False

>>> True or False
True

>>> not True
False
```

Python

```
>>> resultats = 12.8
>>> mention_bien = resultats >= 14 and resultats < 16
>>> print(mention_bien)
False
```

3 if, else et elif

Voici le schéma de fonctionnement d'un test `if` :



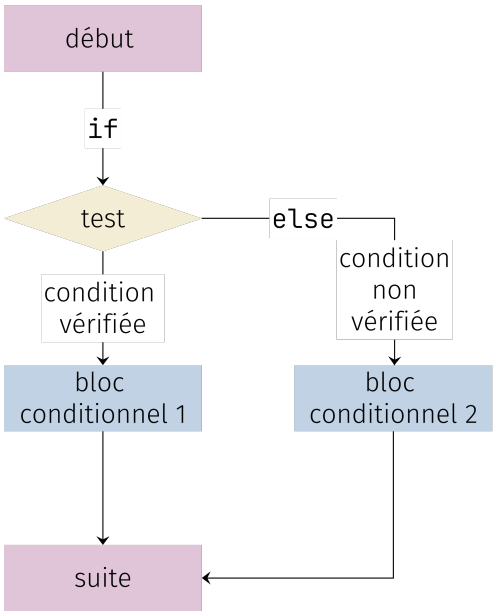
Attention : Un bloc conditionnel doit être *tabulé* par rapport à la ligne précédente : il n'y a ni `DébutSi` ni `FinSi` en PYTHON, ce sont les tabulations qui délimitent les blocs.

Python

```
phrase ='Je vous trouve très joli'
reponse = input('Etes vous une femme ?(O/N) : ')
if reponse == 'O':
```

```
    phrase += 'e' # remarquer la tabulation de cette
    ↪ ligne
phrase +='.'
print(phrase)
```

Voici le schéma de fonctionnement d'un test `if...else` :



Python

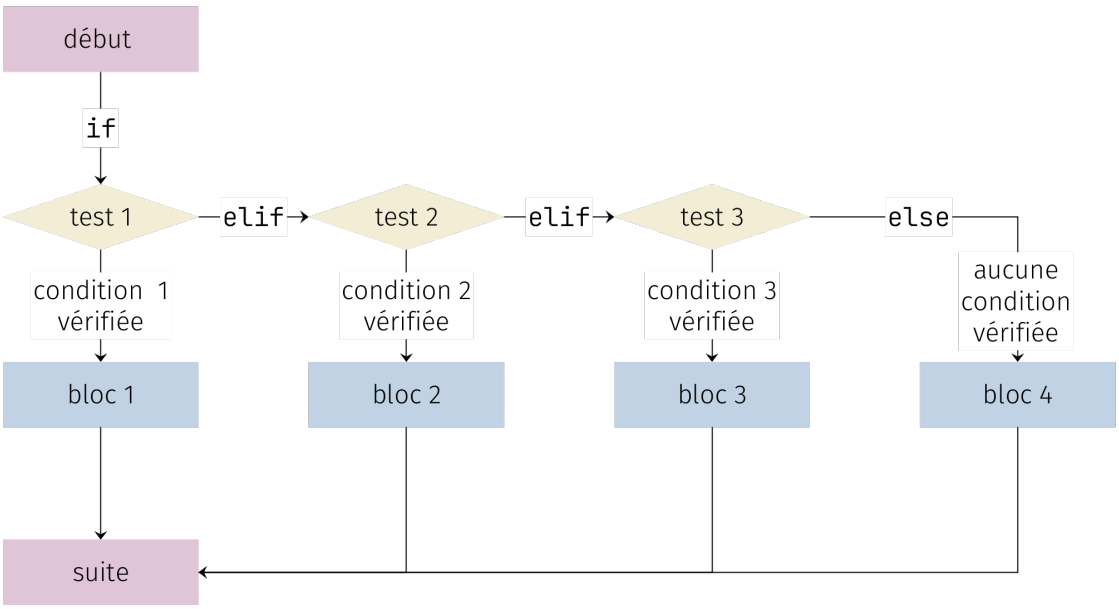
```
print('Bonjour')
age = int(input('Entrez votre age : '))
if age >= 18:
    print('Vous etes majeur')
else:
    print('Vous etes mineur.')
print('Au revoir.')
```

Voici un exemple de fonctionnement d'un test `if...elif...` :

Python

```
print('Bonjour')
prenom = input('Entrez un prénom : ')
if prenom == 'Robert':
    print("Robert, c'est le prénom de mon
    ↪  grand-père.")
elif prenom == 'Raoul':
    print("Mon oncle s'appelle Raoul.")
elif prenom == 'Médor':
    print("Médor, comme mon chien !")
else:
    print("Connais pas")
print('Au revoir.')
```

Et voici un schéma décrivant son fonctionnement :



On peut bien sûr inclure autant de `elif` que nécessaire.

4 Exercices

Exercice 1

Écrire un script qui demande son âge à l'utilisateur puis qui affiche `'Bravo pour votre longévité.'` si celui-ci est supérieur à 90.

Exercice 2

Écrire un script qui demande un nombre à l'utilisateur puis affiche si ce nombre est pair ou impair.

Exercice 3

Écrire un script qui demande l'âge d'un enfant à l'utilisateur puis qui l'informe ensuite de sa catégorie :

- trop petit avant 6 ans ;
- poussin de 6 à 7 ans inclus ;
- pupille de 8 à 9 ans inclus ;
- minime de 10 à 11 ans inclus ;
- cadet à 12 ans et plus ;

Exercice 4

Écrire un script qui demande une note sur 20 à l'utilisateur puis vérifie qu'elle est bien comprise entre 0 et 20. Si c'est le cas rien ne se produit mais sinon le programme devra afficher un message tel que `'Note non valide.'`.

Exercice 5

Écrire un script qui demande un nombre à l'utilisateur puis affiche s'il est divisible par 5, par 7 par aucun ou par les deux de ces deux nombres.

Exercice 6

En reprenant l'exercice du chapitre 1 sur les numéros de sécurité sociale, écrire un script qui demande à un utilisateur son numéro de sécurité sociale, puis qui vérifie si la clé est valide ou non.

Exercice 7

Écrire un script qui résout dans **R** l'équation du second degré $ax^2+bx+c=0$.
On commencera par `from math import sqrt` pour utiliser la fonction `sqrt`, qui calcule la racine carrée d'un `float`.
On rappelle que lorsqu'on considère une équation du type $ax^2+bx+c=0$

- si $a=0$ ce n'est pas une équation de seconde degré ;
- sinon on calcule $\Delta = b^2-4ac$ et
  - Si $\Delta<0$ l'équation n'a pas de solutions dans **R** ;
  - Si $\Delta=0$ l'équation admet pour unique solution $\frac{-b}{2a}$;
  - Si $\Delta>0$ l'équation admet 2 solutions : $\frac{-b-\sqrt{\Delta}}{2a}$ et $\frac{-b+\sqrt{\Delta}}{2a}$.

Pour vérifier que le script fonctionne bien on pourra tester les équations suivantes :

- $2x^2+x+7=0$ (pas de solution dans **R**) ;
- $9x^2-6x+1=0$ (une seule solution qui est $\frac{1}{3}$) ;
- x^2-3x+2 (deux solutions qui sont 1 et 2).

Exercice 8

L'opérateur `nand` est défini de la manière suivante : si `A` et `B` sont deux booléens alors

`A nand B` vaut `not (A and B)`

Construire la table de vérité de `nand` en complétant :

A	B	A and B	not (A and B)
`False`	`False`		
`False`	`True`		
`True`	`False`		
`True`	`True`		

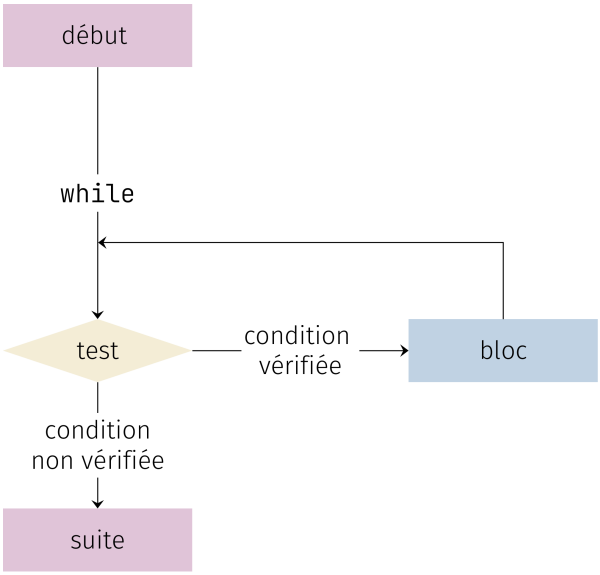
Chapitre 5
Boucles

« Tant que tu n'y arrives pas recommence. »

On s'intéresse dans ce chapitre aux *structures itératives*, plus communément appelées *boucles*.

1 La boucle while

Voici son schéma de fonctionnement :



La boucle `while` exécute un bloc d'instructions conditionnel *tant que* une condition est vérifiée.
Dès que la condition n'est plus vérifiée, le bloc conditionnel n'est plus exécuté.

Python

```
reponse=''
print('Bonjour !')
while reponse !='n':
    reponse = input('Voulez-vous continuer ? (o/n) :
     ↪  ')
print('Au revoir.')
```

La boucle `while` doit être utilisée avec soin : si la condition est toujours vérifiée, le programme ne s'arrêtera pas :

Python

```
while True:
    print('Au secours !')
```

Voici un exemple typique d'utilisation de la boucle `while` :

On place un capital de 2000 euros sur un compte à intérêts annuels de 2%. On aimerait savoir au bout de combien de temps, sans rien toucher, le solde du compte dépassera 2 300 €.

Python

```
solde = 2000  # solde initial
n = 0  # nombre d'annees
while solde <= 2300:  # condition de boucle
    n += 1  # augmente le compteur d'annees
    solde *= 1.02  # actualise le solde
print(' Il nous faudra ', n, 'ans.')  # affichage
 ↪  final
```

2 La boucle for ... in range(...)

Commençons par examiner un nouveau type : `range` (plage de valeurs)

Python

```
>>> a = range(10)
>>> type(a)
<class range>

>>> print(list(a))
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

Si `range(10)` ressemble beaucoup à la liste `[0,1,...,9]`, la finalité de `range(10)` est d'être un *itérateur*, c'est-à-dire une objet dont on peut parcourir le contenu pour créer une boucle :

Python

```
for i in range(10):
    print(i)
```

La syntaxe complète de `range` est : `range(<debut>,fin,<increment>)`.

Par défaut, si ce n'est pas précisé, `debut=0`, et `increment=1`.

`range(<debut>,fin,<increment>)` renvoie la plage de valeurs suivantes :

- On part de la valeur de début, appelons la `val`

- Tant que `val < fin` :
  - ajouter `val` à la plage
  - ajouter `increment` à `val`

Ainsi, `range(2,52,10)` renvoie la plage de valeurs `2,12,22,32,42`, mais `range(2,53,10)` renvoie la plage de valeurs `2,12,22,32,42,52`.

Très souvent, on se contente d'utiliser une instruction du type `range(n)`, où `n` est de type `int`.

Voici un exemple : Calculons $1 + 2 + ... + 100$:

Python

```
somme = 0
for i in range(1, 101):
    somme += i
print(somme)
```

3 La boucle for ... in ...

On peut généraliser le paragraphe précédent à toute *variable itérable*, c'est extrêmement puissant : les `str`, les `list` et les `dict` sont des types itérables.

Voici des exemples :

Comptons le nombre de voyelles d'une chaîne de caractères :

Python

```
voyelles = 'aeiouy'  # ensemble de voyelles
phrase = input('Entrez une phrases sans accents :
 ↪  ').lower()  # phrase mise en minuscules
compteur = 0  # comptera les voyelles
for lettre in phrase:  # on parcourt la phrase
    if lettre in voyelles:  # est-ce une voyelle ?
        compteur += 1  # si oui on comptabilise
print('Nombre de voyelles : ', compteur)  # affiche le
 ↪  nombre
```

Faisons la moyenne d'une liste de notes :

Python

```
liste_notes = [12, 11.5, 13, 18, 13, 11, 9]
moyenne = 0
for note in liste_notes:
    moyenne += note
moyenne /= len(liste_notes)
print(moyenne)
```

Pour le dernier exemple on utilise le type `dict`. Soit `a` une variable de ce type :

- `a.keys()` renvoie la liste des clés (des indices du dictionnaire).
- `a.values()` renvoie la liste des valeurs prises par le dictionnaire.

Voici un second programme de moyenne :

Python

```
resultats = {'EPS': 12, 'maths': 15, 'info': 18}
moyenne = 0
for note in resultats.values():
    moyenne += note
moyenne /= len(resultats)
print(moyenne)
```

4 Quelle boucle utiliser ?

Si la boucle dépend d'une condition particulière on préfèrera la boucle `while`.
Si le nombre d'itérations de la boucle est connu on préfèrera une boucle `for`.
On peut utiliser une boucle `for` sur toute *structure itérative*, par exemple une variable de type `range`, `str`, `list` ou, dans une certaine mesure, `dict`.

5 Exercices

Exercice 9

Calculer à l'aide d'un script la somme des carrés des 1000 premiers entiers non nuls.

Exercice 10

Calculer à l'aide d'un script la somme des carrés des 1000 premiers multiples de 3 non nuls.

Exercice 11

Écrire un script qui demande une phrase à l'utilisateur, puis affiche la phrase en rajoutant des tirets.
Exemple : on entre `'Salut à toi'` le script affiche `'S-a-l-u-t- -à--t-o-i-'`.

Exercice 12

Calculer à l'aide d'un script le nombre n à partir duquel la somme $1^2 + 2^2 + \ldots + n^2$ dépasse un milliard.

Exercice 13

Écrire un script qui demande une phrase et compte le nombre d'occurrences de la lettre « a » dans celle-ci.

Exercice 14

Programmer le jeu du "plus petit plus grand" :

- L'ordinateur choisit un nombre entier au hasard compris entre 0 et 100.
  Au début du script, importer la fonction `randint` du module `random` avec `form random import randint`.
  Pour obtenir un entier au hasard, utiliser `randint(0,100)`.
- L'utilisateur propose un nombre, l'ordinateur répond « gagné » , « plus

petit » ou « plus grand » .

- Le programme continue tant que l'utilisateur n'a pas gagné.

Exercice 15

On considère la suite s définie par :

$$\begin{cases} s_0 &= 1000 \\ s_{n+1} &= 0,99s_n + 1 \quad \text{pour tout } n \in \mathbf{N} \end{cases}$$

- Écrire un script calculant les premiers termes de s (vous décidez le nombre de termes).
- Utiliser ce script pour conjecturer la limite de s.
- Modifier ce script pour obtenir le plus petit entier n tel que l'écart entre s_n et sa limite soit inférieur ou égal à 10^{-4}.

Exercice 16

φ (lettre phi, équivalent du « f » en grec) est défini par :

$$\varphi = 1 + \cfrac{1}{1 + \cfrac{1}{1 + \cfrac{1}{\ddots}}}$$

Sur papier, fabriquer une suite par récurrence commençant ainsi :

- 1
- $1 + \frac{1}{1}$
- $1 + \cfrac{1}{1 + \frac{1}{1}}$
- Et cætera (trouver une relation simple pour calculer le terme suivant à partir du terme actuel).

Programmer un script qui calcule successivement les termes de cette suite (aller jusqu'à 10000^{e}).

Comparer avec la valeur exacte de φ, qui est $\frac{1+\sqrt{5}}{2}$.

Exercice 17

Écrire un script qui détermine si un entier est premier ou pas.

Chapitre 6
Listes

Le type `list` permet de stocker des valeurs dans un ordre précis.

Python

```
# on crée une liste avec 3 valeurs
lst = ['bonjour', 3.14, True]
```

Une valeur de type `list` est itérable :

- on peut accéder à un élément de la liste, par exemple `lst[0]` ;
- on peut parcourir une liste.

Pour accéder à un élement d'une liste situé à un endroit précis, on doit connaître son *indice* : le premier élément d'une liste a l'indice zéro, le deuxième l'indice 1, *et cætera*.

Python

```
# on crée une liste avec 3 valeurs
>>> lst = ['bonjour', 3.14, True]
# premier élément : indice 0
>>> lst[0]
'bonjour'

# deuxième élément
>>> lst[1]
3.14
```

1 Opérations de base

1.1 Créer une liste

- `lst = list()` crée une liste vide ;
- `lst = []` fait la même chose ;
- `lst = ['a', 7, True]` crée une liste composée de 3 éléments.

Une liste peut contenir des éléments de plusieurs types mais en pratique on évite cela.

1.2 Modifier un élément

Le type `list` est *mutable* : on peut changer un ou des éléments d'une liste *sans changer la liste elle-même*.

Python

```
>>> lst = [2, 3, 4, 1]
# on change le deuxième élément
>>> lst[1] = 10
>>> lst
[2, 10, 4, 1]
```

1.3 Ajouter un élément en fin de liste

On reprend l'exemple précédent

Python

```
>>> lst.append(7) # ajoute 7 à la fin de la liste
>>> lst
[2, 10, 4, 1, 7]
```

Remarque

> `lst = lst + [7]` a le même effet que `lst.append(7)` : on crée une « mini-liste » `[4]`, on concatène les 2 listes et on remet le résultat dans `lst`.
> En pratique la première méthode est la plus simple et aussi la plus rapide.

1.4 Insérer un élément à une place donnée

Pour une liste `lst` valant `[2, 10, 4, 1]`, si on veut insérer la valeur 5 à l'indice 1 on écrira :

```
lst.insert(1, 5)
```

et `lst` vaudra `[2, 5, 10, 4, 1]`

La syntaxe est `lst.insert(indice, valeur)`

1.5 Retirer un élément à une position donnée

Si une liste `lst` a pour valeur `[3 ,7, 1]` et qu'on veut supprimer son deuxième élément alors on écrit :

```
del lst[1]
```

Ensuite, `lst` aura la valeur `[3,1]`.

1.6 Retirer une valeur précise

Pour retirer une valeur *qui appartient à une liste* on procède ainsi :

Si `lst` a la valeur `[1, 2, 5, 4, 2, 3]` alors l'instruction

```
lst.remove(2)
```

Supprime la *première occurrence* de 2 dans `lst`.
Après cela, `lst` a la valeur `[1, 5, 4, 2, 3]`.

1.7 Concaténer des listes

On peut procéder de 2 manières :

- `lst1.extend(lst2)` ajoute les éléments de la liste `lst2` à la fin de `lst1`;
- `lst1 = lst1 + lst2` crée une liste avec les éléments de `lst1` et ceux de `lst2`, puis replace le résultat dans `lst1`.

En pratique la première méthode est plus rapide.

1.8 Longueur d'une liste

La fonction `len`

- prend en entrée une liste `lst`;
- renvoie la longueur de cette liste.

Ainsi `len([2, 3, 4])` vaut 3.

1.9 Divers

`lst.sort()` trie la liste dans l'ordre croissant.

`lst.reverse()` met les éléments dans l'ordre inverse.

2 Opérations avancées

2.1 Copier une liste (mauvaise méthode)

Python

```
>>> lst1 = [5, 6, 8]
>>> lst2 = lst1
>>> lst1[0] = 10
>>> lst1
[10, 6, 8]

>>> lst2
[10, 6, 8]  # problème : lst2[0] a changé aussi !
```

Ce comportement « étrange » vient du fait que le type `list` est *mutable*. Nous allons expliquer cela plus tard dans ce chapitre.

2.2 Copier une liste (bonne méthode)

Python

```
>>> lst1 = [5, 6, 8]
>>> lst2 = lst1[:] # on copie tous les éléments de
 ↪  lst1 dans lst2
>>> lst1[0] = 10
>>> lst1
[10, 6, 8]

>>> lst2
[5, 6, 8] # Ouf !
```

2.3 Extraire une sous-chaîne

Soit `lst` une liste de longueur n et p et q deux entiers tels que $0 \leqslant p < q \leqslant n$. Alors

- `lst[p:q]` est la liste composée des éléments `lst[p]`, ..., `lst[q-1]` ;
- `lst[:q]` signifie `lst[0:q]` ;
- `lst[p:]` signifie `lst[p:n]`.

Exemple

Si `lst` vaut `[2, 5, 3, 4, 9, 2, 5]` alors

- `lst[2:6]` vaut `[3, 4, 9, 2]` ;
- `lst[3:]` vaut `[4, 9, 2, 5]` ;
- `lst[:2]` vaut `[2, 5]`.

3 Parcourir une liste

3.1 Parcours selon les indices

Définition : parcours selon les indices

Soit `lst` une liste de longueur `n`.

Alors ses éléments sont `lst[0]`, ..., `lst[n-1]` et on parcourt la liste en

- considérant un entier `i` qui joue le rôle d'*indice* ;
- faisant parcourir à `i` la plage de valeurs `range(n)` ;
- considérant les `lst[i]`.

Exemple : un parcours selon les indices

On affiche les élements d'une liste grâce à un parcours par les indices.

```
lst = [54, 65, 123]
n = len(lst) # n vaut 3
```

```
for i in range(n): # range(3), c'est 0, 1, 2
    print(lst[i])
```

Le parcours d'une liste par les indices est *crucial* si lors du parcours, on veut savoir à quelle place on se trouve dans la liste. C'est le cas quand on veut déterminer si une liste est triée dans l'ordre croissant ou non : il faut regarder si chaque élement est plus petit que le suivant dans la liste.
C'est aussi le cas quand on veut par exemple déterminer l'indice de la première apparition d'une valeur dans une liste.

3.2 Parcours selon les éléments

C'est plus simple que le parcours selon les indices mais on pert un peu d'information car pendant le parcours, on ne sait pas à quelle place on se trouve dans la liste.

Définition : parcours selon les éléments

> Le parcours des éléments d'une liste `lst` s'effectue à l'aide d'une simple boucle `for x in lst`. `x` prend alors successivement les valeurs de chacun des éléments de `lst`, dans l'ordre.

Exemple : un parcours selon les éléments

```
lst = [54, 65, 123]
for x in lst:
    print(x)
```

3.3 Bilan

On peut toujours utiliser un parcours de liste selon les indices. On peut toujours transformer un parcours selon les éléments en un parcours selon les indices. Le contraire est faux si l'on a *absolument* besoin de savoir quels sont les indices des éléments que l'on examine lors du parcours.

À éviter absolument

Les codes comme celui-ci :

```
for i in lst:
    print(i)
```

On a l'impression que `i` est un indice mais c'est une valeur, et l'expérience prouve que dans 90% des cas, une erreur du type `lst[i]` survient, alors que...`i` n'est pas un indice ici !

De même les codes comme celui-là :

```
for x in range(len(lst)):
    print(lst[x])
```

Là encore il y a beaucoup de chances que l'indice `x` soit pris pour une valeur.

Conseil

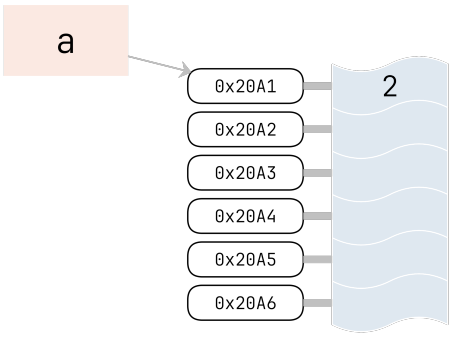
Réserver les noms de variables `i`, `j` et `k` pour les indices et `x`, `y` et `z` pour les éléments.

4 Mutabilité

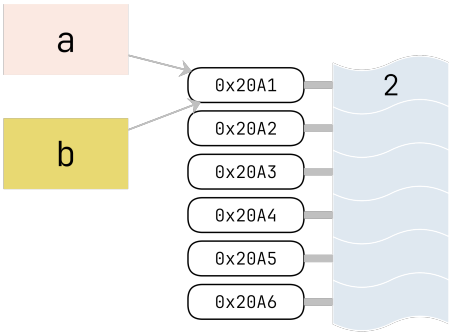
Examinons la différence entre un type *non mutable* tel que `int` et le type `list`, qui est *mutable*.

4.1 Variables de type non-mutable

```
a = 2
```



La valeur 2 est stockée en mémoire et une variable **a** est créée, associée à cette valeur.

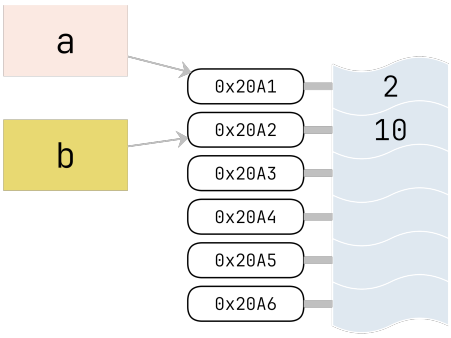



```
b = a
```

Une deuxième variable **b** est créée, avec pour valeur 2 également. Elles partagent la même adresse-mémoire.

```
b = 10
```

La valeur 10 est stockée dans une autre adresse mémoire (car la valeur 2 sert toujours pour **a**) et associée à **b**.

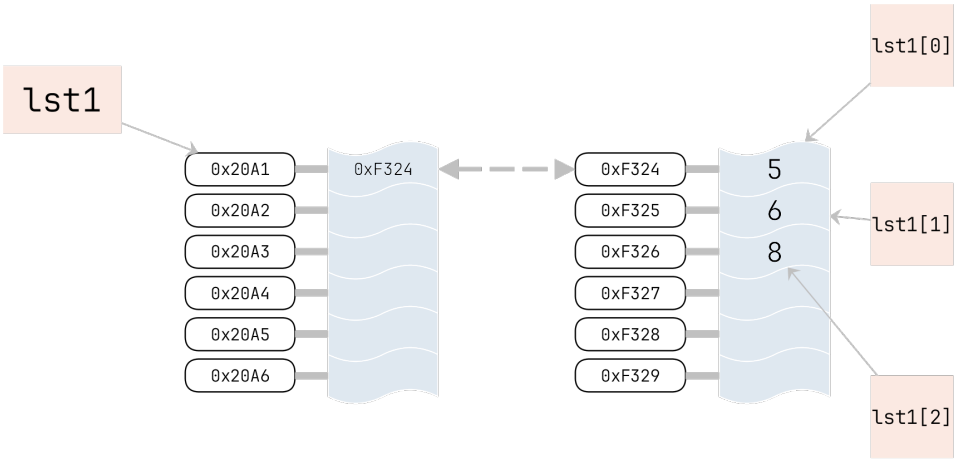


4.2 Variables de type mutable

Copier une liste (mauvaise méthode)

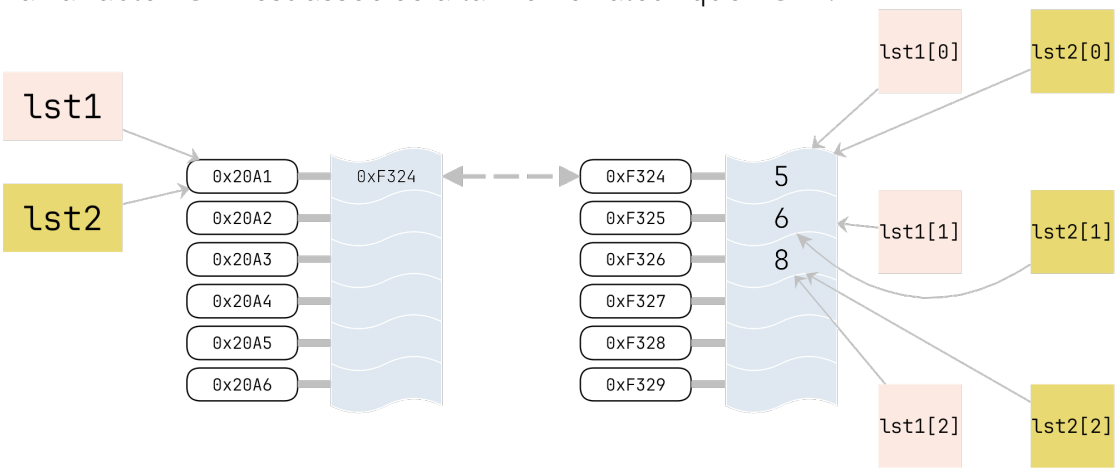
```
lst1 = [5, 6, 8]
```

Les élements 5, 6 et 8 sont stockés en mémoire et `lst1` contient l'adresse du début de la plage mémoire à laquelle ces valeurs sont stockées.



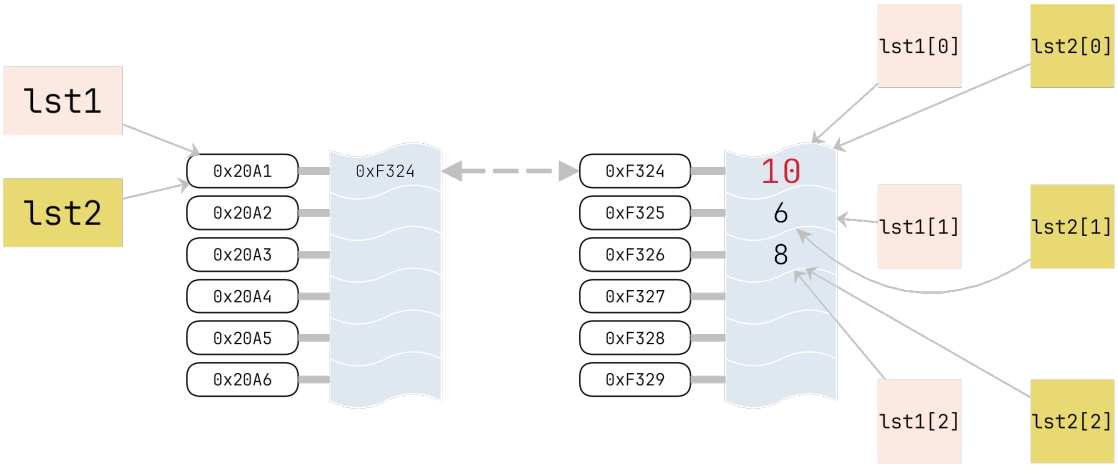

```
lst2 = lst1
```

La variable `lst2` est associée à la même valeur que `lst1`.




```
lst1[0] = 10
```

La valeur de `lst1[0]` est changée, elle l'est donc aussi pour `lst2`.

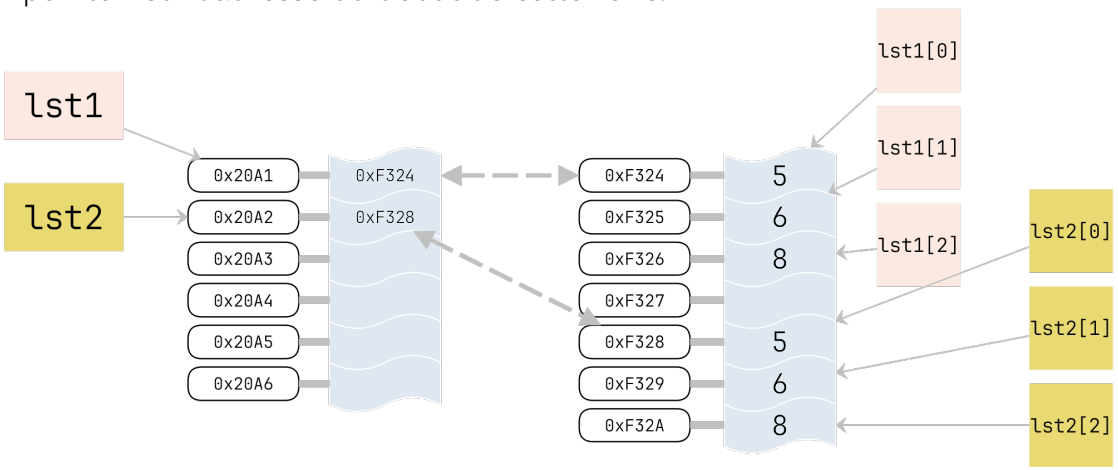


Voilà donc pourquoi lorsqu'on écrit `lst2 = lst1`, tout changement dans `lst1` se reflète aussi dans `lst2` et vice-versa.

Copier une liste (bonne méthode)

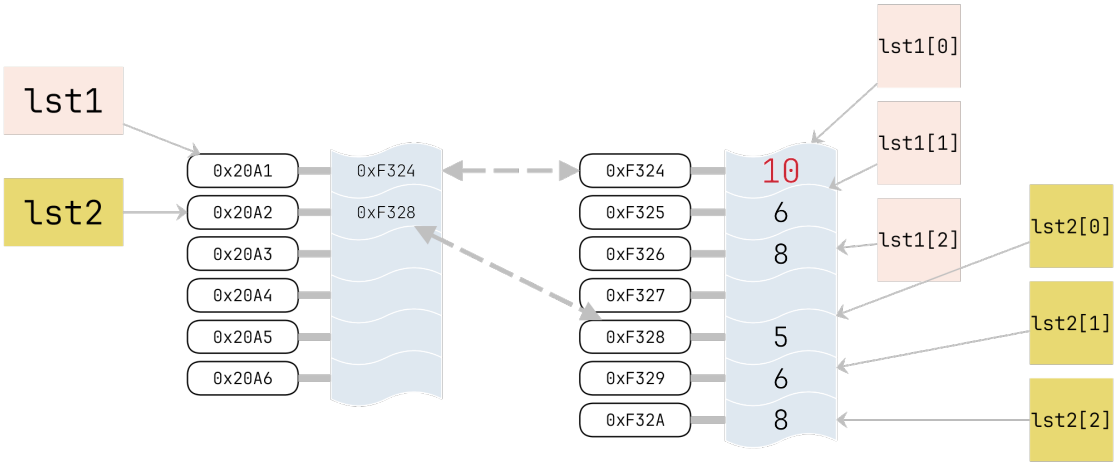
```
lst2 = lst1[:]
```

Les éléments de `lst1` sont recopiés dans une autre zone mémoire, et `lst2` « pointe » sur l'adresse du début de cette zone.




```
lst1[0] = 10
```

Le changement n'affecte pas `lst2`.


lst1
lst2
0x20A1
0x20A2
0x20A3
0x20A4
0x20A5
0x20A6
0xF324
0xF328
0xF324
0xF325
0xF326
0xF327
0xF328
0xF329
0xF32A
10
6
8
5
6
8
lst1[0]
lst1[1]
lst1[2]
lst2[0]
lst2[1]
lst2[2]

Chapitre 7
Fonctions

« Quelle est la fonction de ce chapitre ? »

1 Exemples de fonctions

1.1 Un objet déjà connu

Nous avons déjà rencontré des fonctions *côté utilisateur* :

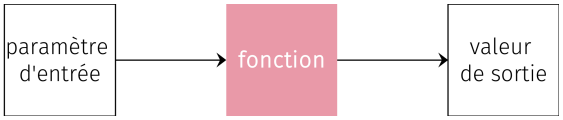
- `input`
  - prend en entrée une chaîne de caractères ;
  - renvoie la chaîne de caractère saisie par l'utilisateur.

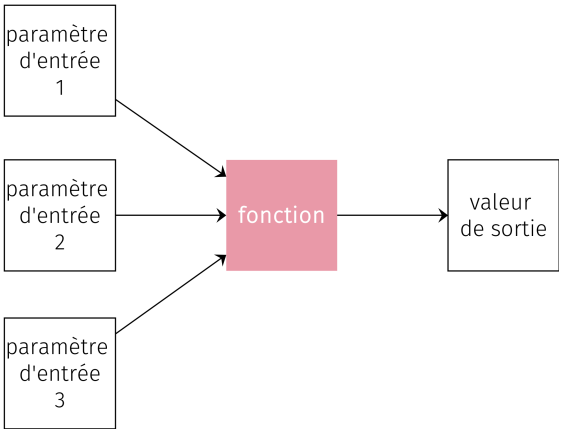
  On peut noter ceci `input(chaine: str) -> str`
- `len`
  - prend en entrée une liste ;
  - renvoie le nombre d'éléments de cette liste.

  On peut noter cela `len(lst: list) -> int`

1.2 De multiples formes

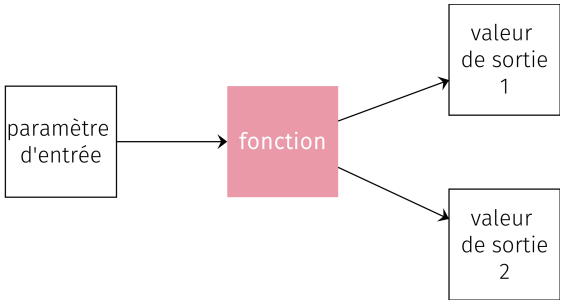
Les deux exemples précédents rentrent dans la catégorie représentée à droite.

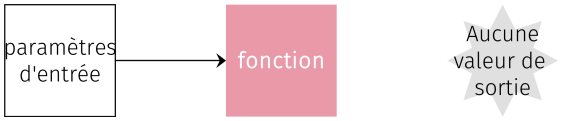




Certaines fonctions sont comme à gauche.
Par exemple `max(20,3,10)` renvoie 20.

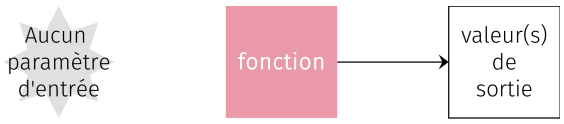
D'autres fonctions sont comme à droite.
On verra des exemples plus tard.





D'autres encore sont comme à gauche.
Par exemple `print("salut")` ne renvoie rien mais affiche `salut` à l'écran.

D'autres suivent le schéma ci-contre.
Par exemple dans le module `time`, la fonction `time` ne prend aucun paramètre d'entrée mais renvoie l'heure qu'indique l'horloge de l'ordinateur.
On peut par exemple l'utiliser pour stocker une heure précise en tapant `maintenant = time()`.



Enfin certaines suivent ce schéma. Par exemple dans le module `pygame`, `pygame.display.flip` ne prend aucun paramètre d'entrée, ne renvoie aucune valeur, mais actualise la fenêtre graphique.
On l'appelle donc en tapant `pygame.display.flip()`.

Il est possible de créer de nouvelles fonctions. On parle alors de fonctions *côté concepteur*.

Il faut donc définir rigoureusement ce qu'est une fonction.

2 Définition de la notion de fonction

Définition : fonction

Une *fonction* est un « morceau de code » qui représente un *sous-programme*.
Elle a pour but d'effectuer une tâche *de manière indépendante*.

Exemple

On veut modéliser la fonction mathématique f définie pour tout nombre réel x par

$$f(x) = x^2 + 3x + 2$$

On écrira alors

```
def f(x : float) -> float:
    return x ** 2 + 3 * x + 2
```

Pour évaluer ce que vaut $f(10)$ et affecter cette valeur à une variable, on pourra désormais écrire `resultat = f(10)`.

Que fait la fonction `mystere` ?

```
def mystere(a : float, b : float) -> float:
    if a <= b:
        return b
    else:
        return a
```

La fonction `mystere` :

- prend en entrée deux paramètres de type `float` `a` et `b` ;
- renvoie le plus grand de ces deux nombres.

La réponse que l'on vient de formuler s'appelle *la spécification* de la fonction *f*.

Définition : fonction

Donner la spécification d'une fonction `f` c'est

- préciser le(s) type(s) du (des) paramètre(s) d'entrée (s'il y en a) ;
- indiquer sommairement ce que fait la fonction `f` ;
- préciser le(s) type(s) de la (des) valeur(s) de sortie (s'il y en a).

3 Anatomie d'une fonction

Python

```
def f(lst: list) -> int
    mini = lst[0]
    n = len(lst)
    for i in range(n):
        lst[i] < mini:
        mini = lst[i]
    return mini
```

La fonction `f`

– prend en entrée une liste (sous entendu d'entiers) ;

– renvoie le plus petit entier de cette liste.

3.1 Paramètre formel

```
def f(lst: list) -> int
    mini = lst[0]
    n = len(lst)
    for i in range(n):
        lst[i] < mini:
        mini = lst[i]
    return mini
```

paramètre formel

Le paramètre d'entrée est *formel* : *le nom de cette variable n'existe qu'à l'intérieur de la fonction*. Si ce nom de variable existe déjà à l'extérieur de la fonction, *ce n'est pas la même variable*.

Le type du paramètre d'entrée peut être spécifié. Ce n'est pas obligatoire mais très fortement recommandé pour « garder les idées claires » .

```
def f(lst: list) -> int
    mini = lst[0]
    n = len(lst)
    for i in range(n):
        lst[i] < mini:
        mini = lst[i]
    return mini
```

type du paramètre formel

3.2 Variables locales

```
def f(lst: list) -> int
    mini = lst[0]
    n = len(lst)
    for i in range(n):
        lst[i] < mini:
        mini = lst[i]
    return mini
```

variables locales

Toutes les variables *créées* dans une fonction n'existent *que dans cette fonction*. Elles ne sont pas accessibles depuis l'extérieur de la fonction. On dit que ce sont des *variables locales*.

3.3 valeur de sortie

Le type de la valeur de sortie peut être précisé, c'est également recommandé.

```
def f(lst: list) -> int
    mini = lst[0]
    n = len(lst)
    for i in range(n):
        lst[i] < mini:
        mini = lst[i]
    return mini
```

type de la valeur renvoyée

4 En pratique

4.1 Des exemples

Python

```
1   def f(x : float) -> float:
2       return x ** 2 + 3 * x + 2
3
4   print(f(1)) # Affiche 6
```

Le programme commence à la ligne 4 !
Les 2 premières lignes servent à définir la fonction `f`, elles ne sont exécutées que lorsqu'on évalue `f(1)`.

Python

```
def f(x : float) -> float:
    return x ** 2 + 3 * x + 2

print(x) # Provoque une erreur
```

L'erreur vient du fait que la variable `x` *n'est pas définie*. Le « `x` qu'on voit dans

la fonction `f` » est un paramètre formel et n'existe que dans `f`.

Python

```python
def f(x : float) -> float:
    a = 2
    return x + a

print(a) # Provoque une erreur
```

L'erreur vient du fait que la variable `a` *est locale* : elle n'est définie que durant l'exécution de `f`.

Python

```python
def f(x : float) -> float:
    a = 2
    return x + a

print(f(4)) # Affiche 6
print(a) # Provoque une erreur
```

C'est encore la même erreur : une fois `f(4)` évaluée, `a` n'existe plus.

Python

```python
def f(x : float) -> float:
    a = 2
    return x + a

a = 3
print(f(4)) # Affiche 6
print(a) # Affiche 3 et pas 2
```

La variable `a` définie dans la fonction `f` n'est pas la même que celle qui est

définie à la ligne 5.
Celle définie à la ligne 2 est *locale*.
La variable `a` de la ligne 5 est appelée *globale*.

Python

```python
def f(x : float) -> float:
    return x + a

a = 3
print(f(4)) # Affiche 7
```

À retenir

> Une fonction a le droit d'*accéder en lecture* à une variable globale, mais n'a pas *a priori* le droit d'en modifier la valeur.

4.2 À éviter autant que possible

Python

```python
1  def f(x : float) -> float:
2      global a
3      a = a + 1
4      return x + a
5
6  a = 3
7  print(f(4)) # Affiche 8
8  print(a) # Affiche 4
```

À la ligne 2, on signale à Python que `f` a la droit de modifier la variable globale `a`. C'est fortement déconseillé : sauf si on ne peut pas faire autrement, une fonction ne doit pas modifier les variables globales.

Chapitre 8

Écriture en compréhension

1 Écritures simples

Jusqu'à présent, pour construire des listes on a souvent :

- créé une liste `lst` vide ;
- construit une boucle `for` ou `while` ;
- peuplé la liste avec `lst.append`.

Exemple

```
lst = []
for i in range(10):
    lst.append(i*i)
print(lst)
```

Ce programme affiche `[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]`
C'est la liste des carrés des 10 premiers entiers naturels.
En mathématiques, l'ensemble des carrés des 10 premiers entiers naturels se note

$$\{i^2 \mid i \in \mathbf{N},\, i < 10\}$$

C'est une écriture en *compréhension*.
On peut faire la même chose en PYTHON :

```
lst = [i*i for i in range(10)]
```

Évidemment, c'est plus rapide que la méthode précédente... Et on peut faire bien plus ! On peut utiliser une liste pour en construire une autre, par exemple en ajoutant 1 à chacun des éléments :

Python

```
>>> lst1 = [2, -1, 3, 4, 7]
>>> lst2 = [x + 1 for x in lst1]
>>> lst2
[3, 0, 4, 5, 8]
```

Dans le même esprit, on peut construire une liste dont les élements sont ceux de la première, mais avec une conversion de type :

Python

```
>>> lst1 = ['2', '0', '13']
>>> lst2 = [int(x) for x in lst1]
>>> lst2
[2, 0, 13]
```

Ou encore fabriquer la liste des initiales à partir d'une liste de prénoms :

Python

```
>>> lst1 = ['Fred', 'Titouan', 'Tinaïg']
>>> lst2 = [prenom[0] for prenom in lst1]
>>> lst2
['F', 'T', 'T']
```

2 Écritures avec conditions

Il est possible d'utiliser `if` en compréhension : mettons dans `lst2` le double de chaque élément de `lst1` supérieur à 10 (dans l'ordre de parcours).

Python

```
l>>> lst1 = [8, 0, 11, 10, 3, 15]
>>> lst2 = [2 * x for x in lst1 if x > 10]
>>> lst2
[22, 30]
```

Il est possible d'utiliser `if ... else ...` en compréhension, mais à ce moment là il faut écrire les conditions au début : créons une nouvelle liste en remplaçant tous les nombres négatifs de `lst1` par zéro.

Python

```
>>> lst1 = [8, -10, 11, -4, -3, 15]
>>> lst2 = [(x if x > 0 else 0) for x in lst1]
# les parenthèses sont facultatives
>>> lst2
[8, 0, 11, 0, 0, 15]
```

Créons une liste contenant les indices des éléments de `lst1` qui sont strictement positifs.

Python

```
>>> lst1 = [8, -10, 11, -4, -3, 15]
>>> lst2 = [i for i in range(len(lst1)) if lst1[i] >
 ↪  0]
>>> lst2
[0, 2, 5]
```

3 Les écritures en compréhension imbriquées

0 0 0 0
0 0 0 0
0 0 0 0

Si on veut représenter ce « tableau de nombres » par une liste, on peut écrire cet liste de 3 lignes comportant chacune 4 éléments :

`lst = [[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]`.

Cependant il est plus pratique d'écrire

```
lst=[[0 for j in range(4)] for i in range(3)]
```

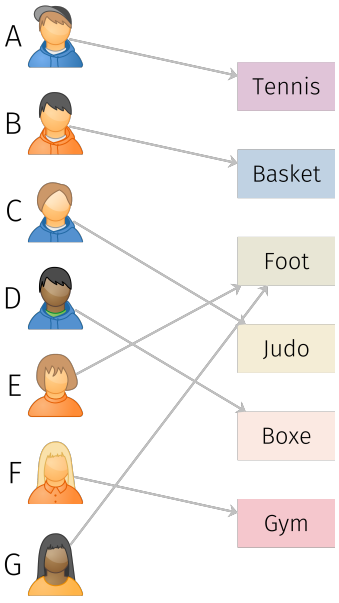
4 Pour conclure

On peut combiner toutes les techniques que nous venons de voir. Par exemple on peut créer une liste de listes de listes avec des conditions, *et cætera*. La seule limite, c'est l'imagination et la capacité à écrire en PYTHON !

Chapitre 9
Dictionnaires

1 Un nouveau type

On demande à des jeunes quel est leur sport préféré, les résultats sont présentés sur la figure ci-contre.
Un sport peut être cité par *plusieurs* jeunes, en revanche chaque jeune ne peut citer qu'*un seul* sport.
On pourrait utiliser une ou plusieurs listes pour représenter ces données mais il y a mieux : le *dictionnaire*. La variable `sport` est de type `dict` :



Python

```
sport = {'A': 'Tennis', 'B': 'Basket',
         'C': 'Judo', ' D': 'Boxe',
         'E': 'Foot', 'F': 'Gym',
         'G': 'Foot'}
```

Définition : dictionnaire

Un dictionnaire est un ensemble d'*éléments*.
les éléments sont des couples de la forme *clé : valeur*.

La syntaxe est :
`variable = { cle1 : valeur1, cle2 : valeur2, ...}`

Les valeurs peuvent être de n'importe quel type. Les clés peuvent être

- des `bool`, des `int`, des `float`;
- des `str`...
- mais pas des `list`!

On peut tout de même utiliser des `tuples` en guise de clés : les `tuples` ressemblent aux `list` mais sont *non mutables*.

`a = (1, 2, 3)` est un exemple de `tuple`.

2 Opérations sur les dictionnaires

2.1 Accéder à une valeur par sa clé

Pour connaître le sport préféré de `'A'`, c'est simple :

Python

```
>>> sport['A']
'Tennis'
```

2.2 Créer de nouveaux couples clé : valeur

Contrairement aux listes, il n'y a pas de méthode `append`.
Pour intégrer l'information « le sport préféré de H est le Rugby » on écrira sim-

plement :

Python

```
>>> sport['H']='Rugby'
```

2.3 Créer un dictionnaire vide et le peupler

On peut partir d'un dictionnaire vide et remplir ses valeurs au fur et à mesure :

Python

```
>>> d = dict()
>>> d['bonjour'] = 'hello'
>>> d['crayon'] = 'pencil'
>>> d['se prélasser'] = 'to bask'
```

2.4 Supprimer un élément du dictionnaire

`del d['crayon']` supprime l'élément `'crayon': 'pencil'`.

fusionner 2 dictionnaires

```
>>> d1 = {"anglais": "bread",
          "français": "pain",
          "slovaque": "chlieb"}
>>> d2 = {"allemand": "brot", "italien": "pane"}
>>> d1.update(d2) # fusionne d2 dans d1
>>> d1
{"anglais": "bread", "français": "pain", "slovaque":
 ↪  "chlieb", "allemand": "brot", "italien": "pane"}
```

2.5 Parcourir l'ensemble des clés d'un dictionnaire

Python

```
for cle in d1.keys():
    print(cle)
```

Ce script affiche

```
anglais
français
slovaque
allemand
italien
```

2.6 Parcourir l'ensemble des valeurs d'un dictionnaire

Python

```
for valeur in d1.values():
    print(valeur)
```

Ce script affiche

```
bread
pain
chlieb
brot
pane
```

2.7 Précisions

`d1.keys()` et `d1.values()` ressemblent à des listes mais n'en sont pas![1]

[1] Ce sont des *itérateurs*, sctructures destinées à être parcourues.

Pour avoir par exemple la liste des clés de `d1` on écrira :

```
list(d1.keys())
```

ou bien en *compréhension* (ce qui revient au même mais peut s'avérer utile)
`[k for k in d1.keys()]`

2.8 Erreurs de clé

```
print(d1['suédois'])
```

Ce script produit une erreur :
`KeyError : 'suédois'`

Exemple : Utilisation d'un dictionnaire

> On veut créer un tableau de 10 × 10 cases avec la valeur 0 dedans.
>
> On peut bien sûr créer cela avec une liste de listes (en compréhension) mais on peut également utiliser un dictionnaire :
>
> ```
> {(x, y) : 0 for x in range(1, 11) for y in range(1,
> 11)}
> ```
>
> Avantages :
>
> - plus simple à manipuler : on écrit `d[x, y]` au lieu de `d[x][y]` ;
> - on n'est pas obligé de faire commencer les indices à zéro.
>
> Inconvénients :
>
> - prend plus de place en mémoire (on s'en fiche un peu) ;
> - plus flexible entraîne plus de possibilité d'erreurs !

3 Utilisation des dictionnaires

Typiquement, pour stocker des *données structurées* :

```
reseau = {'nom'        : 'local',
          'ip'         : '192.168.1.0',
          'masque'     : '255.255.255.0',
          'passerelle' : '192.168.1.254'}
```

On utilise fréquemment des listes de dictionnaires, ou bien des dictionnaires de listes.

Table des matières