



1 Principes de base

- ▶ Python peut fonctionner en **mode interactif** : une **console** (ou *shell*) affiche une *invite de commande* (reconnaissable aux *triples chevrons* `>>>`). On saisit des **instructions** qui, une fois **validées** par un appui sur la touche , sont **aussitôt exécutées**, et leur **éventuel résultat affiché**.
- ▶ Python sait **exécuter des programmes (ou scripts) enregistrés dans des fichiers portant l'extension .py**.
- ▶ On réserve l'utilisation de la console aux tests rapides : **un code qui dépasse quelques lignes sera enregistré et exécuté depuis un fichier** (les modifications sont simplifiées).
- ▶ **On met une instruction par ligne** : des lignes courtes sont plus *lisibles*.
- ▶ **Les instructions s'enchaînent ligne à ligne, séquentiellement**, de la première à la dernière, sauf si l'ordre d'exécution est modifié par une *structure de contrôle* (voir plus loin les boucles et les tests).

2 Les variables

- ▶ Une **variable** permet de **mémoriser une information**, qu'on pourra ensuite réutiliser grâce à son nom. Ce nom **doit** débiter par une lettre, et peut comporter d'autres lettres, des chiffres ou le caractère « _ » (touche ).
- ▶ Lorsqu'on mémorise une valeur à l'aide d'une variable, on parle d'**affectation** : l'instruction Python `taille = 175.2` **affecte la valeur 175.2 à la variable nommée taille**.
- ▶ Dans un **algorithme**, l'affectation s'écrit souvent $a \leftarrow 2$. Cette ligne, et l'instruction Python `a = 2`, se traduisent en français par « la variable *a* prend la valeur 2 ».



Une variable ne doit JAMAIS porter un nom déjà réservé par Python



- ▶ Les variables peuvent « **changer de valeur** » au cours du temps. Elles permettent de mémoriser des nombres mais aussi du texte, par exemple : `vente = "Ça coûte 10 €"` ; `print(vente)`. Notez les 3 guillemets de part et d'autre du texte : ils le délimitent. L'ensemble se nomme une **chaîne de caractères**.
- ▶ On peut « **faire des choses** » avec des variables, notamment calculer :
 - les opérations habituelles sont disponibles (utiliser les parenthèses lorsque c'est nécessaire). Notez que l'addition permet de « coller » des chaînes de caractères (on dit « concaténer » : `print("Ça" + "va ?")`);
 - la puissance est représentée par deux étoiles `**` : `a = 2**10` ; `print(a)` calculera 2^{10} puis affichera 1024;
 - la division standard donne un résultat *décimal* : `b = 9/4` ; `print(b)` calculera puis affichera 2.25;
 - la division *entière* existe. L'opérateur associé est le « double slash » `//` : `c = 9//4` ; `print(c)` affichera 2;
 - le *reste de la division entière* est donné par l'opérateur « modulo » `%`. On a la relation $a = (a // b) \times b + (a \% b)$: avec $a = 7$ et $b = 2$, on a $7 = 2 \times 3 + 1$, et `a // b` donnera 3 tandis que `a % b` donnera 1.
- ▶ Ces exemples montrent des données de différentes *natures* : on parle de **type d'une donnée** et, par extension, de la **variable** qui la désigne. Entre autres, il y a le type entier `int`, flottant (ou décimal) `float`, et chaîne de caractères `str`.
- ▶ **Optionnel** On peut convertir une variable d'un type à un autre. Ainsi :
 - l'instruction `texte = str(3.14)` mémorisera la chaîne de caractères "3.14" via la variable nommée `texte`;
 - l'instruction `entier = int(3.14)` mémorisera la valeur entière 3 via la variable nommée `entier`;
 - l'instruction `entier = int("3")` mémorisera la valeur entière 3 via la variable nommée `entier`;
 - l'instruction `dec = float("3.14")` mémorisera la valeur décimale 3.14 via la variable nommée `dec`.
- ▶ **Optionnel** Un programme peut interagir avec l'utilisateur. L'instruction `nom = input("Saisir votre nom : ")` mémorisera dans la variable `nom` une *chaîne de caractères* censée contenir le nom de l'utilisateur.

Exemples :

```

1 >>> valeur = 10 // 3 + 7 % 5      # Ceci est un commentaire, utile aux humains. Python les ignore.
2 >>> print(valeur)                 # Affichera 5
3 >>> valeur = valeur - 3           # Écriture surprenante, non ?
4 >>> print(valeur)                 # Affichera 2

```

L'instruction ligne 3 ressemble à une équation qui n'aurait *aucune* solution. En informatique, elle signifie « *récupère la valeur désignée par la variable `valeur`, retranche lui 3, puis affecte le résultat à la variable `valeur`* ».

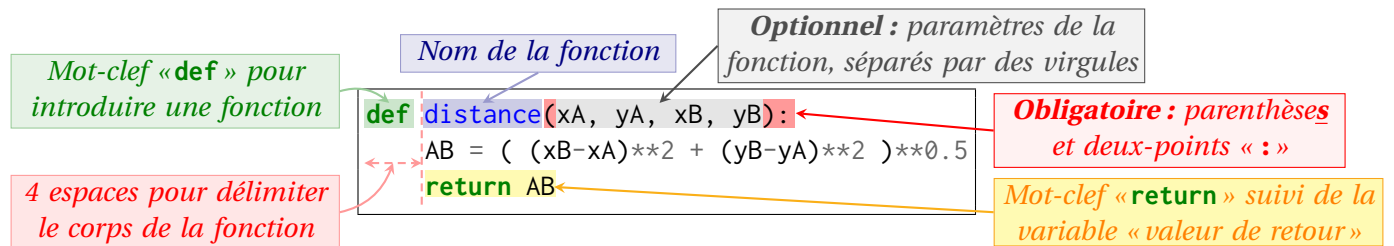
3 Étendre les fonctionnalités de Python

Python ne connaît de base qu'un petit nombre d'opérations et fonctions (par exemple, il ignore comment calculer un sinus). Pour étendre ses capacités, on dispose de *bibliothèques* de fonctions additionnelles, ou **modules**. Pour importer un module, la syntaxe est `from module import *`. On remplacera le plus souvent **module** par :

- ▶ **math** : donne accès aux *fonctions mathématiques*, comme `sqrt` (racine carrée), `sin` (sinus), la constante `pi`, etc.;
- ▶ **random** : donne accès aux *fonctions aléatoires*. En particulier :
 - `random` : chaque fois que la commande `alea = random()` sera exécutée, la variable `alea` désignera une valeur *décimale* prise au hasard entre 0 *inclus* et 1 *exclus*.
 - `randint` : chaque fois que la commande `alea = randint(1,6)` sera exécutée, la variable `alea` désignera une valeur *entière* prise au hasard entre 1 et 6 *inclus* (valeurs à adapter selon les besoins).
- ▶ **turtle** : donne accès à des fonctions permettant de réaliser simplement des *tracés*.

4 Structure de contrôle n°1 : la fonction

Il est fréquent que des portions de codes se répètent : au lieu de recopier le code, **on définit une fonction**, qu'on peut ensuite **appeler** chaque fois que c'est utile. La définition d'une nouvelle fonction doit respecter les règles suivantes :



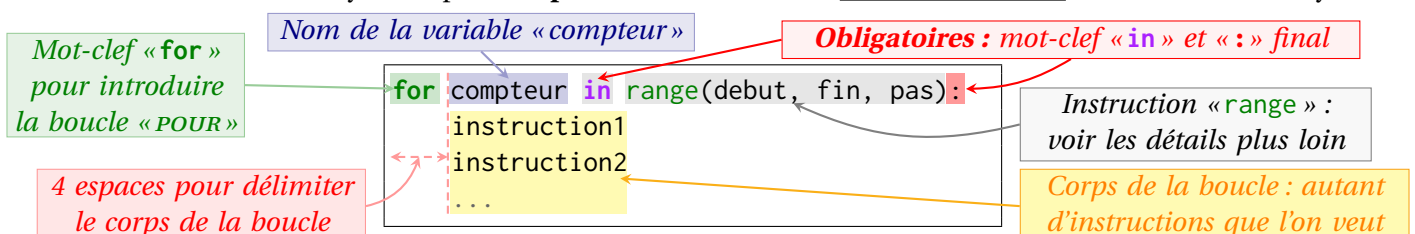
- ▶ La déclaration d'une fonction est la ligne débutant par **def**, suivi d'une espace, puis du **nom** de la fonction, puis de **parenthèses** qui entourent les *éventuels* noms de ses *arguments*, séparés par des virgules s'il y en a plusieurs.
- ▶ Le **corps** de la fonction suit, avec un décalage de 4 espaces pour l'ensemble des instructions qui le composent. Il prend fin lorsque le décalage de 4 caractères s'arrête (sauter une ligne vide pour plus de lisibilité!).
- ▶ Si la fonction doit **renvoyer une valeur**, utiliser le mot-clé **return** suivi de la **valeur de retour** (souvent un nom de variable). Il ne devrait y avoir qu'au plus un **return** par fonction (utiliser une variable, qu'on renvoie à la fin).
- ▶ **Appeler** ou **exécuter** une fonction se fait en utilisant son nom suivi de parenthèses : `ma_fonction()` (insérer les éventuels arguments entre les parenthèses). **Une fonction doit être définie avant qu'on y fasse appel!**

Exemple : voici une définition de fonction pour un usage *purement* mathématique.

```
def f(x):
    return x**2-3*x+4
resultat = f(-2)           # La variable nommée « resultat » mémorisera la valeur 14
```

5 Structure de contrôle n°2 : la boucle « POUR »

Une fonction permet de « factoriser » le code : c'est une sorte de raccourci. Mais si l'on veut exécuter 10 fois de suite les mêmes instructions, il faudra recopier les 10 appels de fonctions : cela reste fastidieux! On peut faire mieux avec une **boucle « POUR »** : un moyen simple de **répéter des instructions un nombre défini de fois**. En voici la syntaxe :



L'instruction « **range** » admet 3 variantes, selon le nombre de paramètres, **tous entiers**, qu'on lui passe. Exemples :

► **for** compteur **in range**(5):

La variable nommée compteur prendra successivement les valeurs 0, 1, 2, 3 et 4 : 5 est donc la **première valeur qui NE sera PAS prise** par la variable compteur.

► **for** compteur **in range**(-3, 2):

compteur prendra successivement les valeurs -3, -2, -1, 0, 1 : -3 est donc la **première valeur PRISE** par compteur, 2 est la **première valeur qui NE sera PAS prise** par compteur.

► **for** compteur **in range**(8, 1, -2):

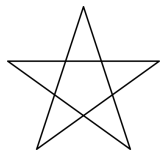
compteur prendra successivement les valeurs 8, 6, 4 et 2 : 8 est donc la **première valeur PRISE** par compteur, 1 est la **première valeur qui NE sera PAS prise** par compteur, et les valeurs changent par sauts de longueur -2.

⚠ Avec « **for** compteur **in range**(1, 8, 2): », compteur prendra successivement les valeurs 1, 3, 5 et 7.

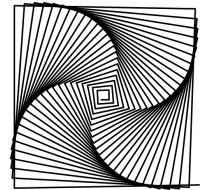
Exemples :

Les codes ci-dessous produisent les tracés à leurs droites.

```
from turtle import *
for i in range(5):
    left(144)
    forward(300)
```

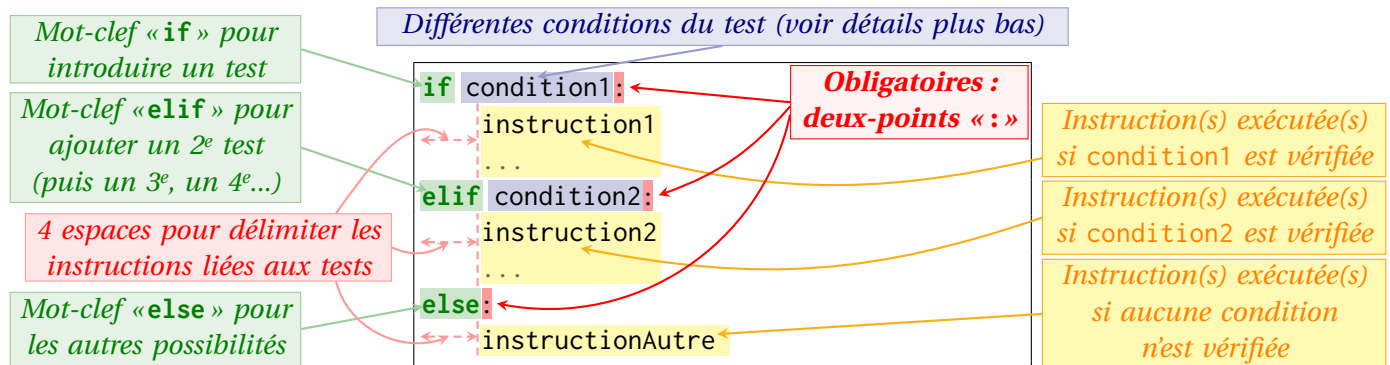


```
from turtle import *
speed(0)
for i in range(20, 480, 5):
    forward(i)
    left(91)
```



6 Structure de contrôle n°3 : tests et structures conditionnelles

Un test (ou *structure conditionnelle*) permet de **modifier le flux d'exécution d'un programme** (qui, autrement, s'exécute *séquentiellement*, de sa première ligne à sa dernière ligne). Il déclenche l'exécution d'un bloc d'instructions si une condition est vérifiée. Plusieurs tests peuvent s'enchaîner. Voici la structure générale d'un test.



Les conditions sont le cœur du test : elles impliquent la comparaison entre une variable (ou le résultat d'un appel de fonction) et une valeur, ou entre deux variables. **Les opérateurs de comparaison essentiels sont :**

== pour l'égalité (il s'agit bien d'un « *double égal* »!) : $a == 2$ sera vrai si la variable nommée a désigne la valeur 2.

!= pour la différence : $a != 2$ sera vrai si la variable nommée a désigne une valeur *autre* que 2;

> pour la supériorité stricte, **>=** pour la supériorité large : $a >= 2$ sera vrai si $a \geq 2$;

< pour l'infériorité stricte, **<=** pour l'infériorité large : $a <= 2$ sera vrai si $a \leq 2$.

Python comprend des tests tels que $2 < a <= 3$, et on peut combiner des tests avec les opérateurs logiques :

not pour le contraire : **not** ($a > 2$) sera vrai lorsque $a \leq 2$;

and pour «et» : ($a > 2$) **and** ($a < 3$) sera vrai lorsque $2 < a < 3$, et sera donc équivalent au test $2 < a < 3$;

or pour «ou» : ($a < 2$) **or** ($a > 3$) sera vrai lorsque $a < 2$ ou $a > 3$, soit $a \in]-\infty; 2[\cup]3; +\infty[$.

Optionnel Le résultat d'un test est une valeur appelée *booléen* (de type **bool**). Il n'existe que deux valeurs booléennes : **True** («vrai») et **False** («faux»). Évidemment **not True** donne **False**, et donc **not True == False** donne **True** 😊!
Attention aux surprises néanmoins : $0.1 * 3 == 0.3$ donnera **False** (pourquoi ? réponse en spécialité NSI 😊!)

Optionnel On teste l'appartenance à une chaîne de caractères avec le mot-clef **in** : **"BA" in "ABBA"** vaudra **True** !

Exemple : voici une définition de fonction reposant sur un test. Elle donne l'issue d'un examen selon le résultat.

```

1 def issue_examen(moyenne):
2     if moyenne < 8:
3         return ""refusé""
4     elif 8 <= moyenne < 10:
5         return ""au rattrapage""
6     else:
7         return ""admis""

```

On utilise cette fonction de la façon suivante :

```

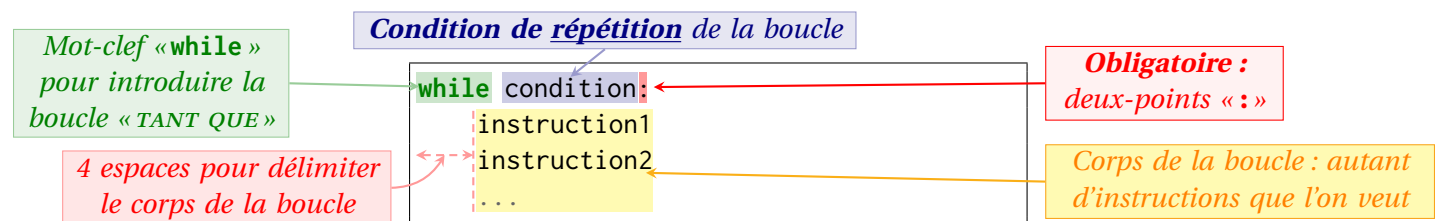
resultat = "Vous êtes " + issue_examen(11.5)
print(resultat) # Affichera « Vous êtes admis »

```

On pourrait ajouter d'autres tests pour tenir compte des mentions (ces tests complémentaires devraient prendre place à l'intérieur de la clause **else**, afin de n'être effectués que s'ils sont réellement utiles).

7 Structure de contrôle n°4 : la boucle « TANT QUE »

Si la boucle « POUR » permet de répéter des instructions un nombre de fois *connu à l'avance*, la boucle « TANT QUE » **répètera des instructions TANT QU'une condition reste vraie**. La boucle sera quittée dès que la condition ne sera plus vérifiée : **une boucle « TANT QUE » implique donc un test!** En voici la structure.



La condition peut être n'importe quel test convenablement rédigé (y compris un test composé, faisant appel à des opérateurs logiques) : tant qu'elle reste vraie, les instructions contenues dans le corps de la boucle seront répétées (éventuellement indéfiniment, si l'on n'y prend pas garde!). Il est donc *important* que le test de cette condition porte sur une variable qui *évolue* (qui désigne une valeur qui change) au cours des passages dans la boucle!

Optionnel On peut quitter une boucle « de force » avec le mot-clef **break** (mais c'est une pratique considérée comme *inélégante*, il est donc préférable de l'éviter).

Exemple : la **conjecture de COLLATZ**. Prenez un nombre entier positif, et appliquez lui le traitement suivant :

- s'il est pair, vous le divisez par 2;
- s'il est impair, vous le multipliez par 3 et ajoutez 1.

Vous obtenez alors un nouveau nombre, sur lequel vous répétez la procédure. Et ainsi de suite... La **conjecture de SYRACUSE** (l'un de ses autres noms) s'énonce ainsi : quel que soit l'entier choisi au départ, on finira par obtenir 1. **Aucun mathématicien n'est en mesure de prouver sa véracité**, à l'heure actuelle (même si on la pense exacte)! Voici une fonction, impliquant une boucle « TANT QUE » (ainsi que des tests), permettant de débiter son exploration.

```

1 def verifie_collatz(n):
2     while n != 1:      # Tant que n est différent de 1...
3         print(n)       # On affiche n (!\ rien à voir avec la valeur renvoyée par la fonction)
4         if n % 2 == 0:  # n%2 est le reste de la division entière de n par 2. S'il est nul alors
5             n = n // 2  # n est pair et on peut diviser n par 2.
6         else:          # Sinon n est impair et le nouveau nombre à considérer, toujours désigné
7             n = 3*n + 1 # par la variable n, est le triple du nombre n précédent augmenté de 1.
8     return True        # Si on arrive ici, la fonction renvoie le booléen True (« vrai »).

```

On utilise cette fonction de la façon suivante :

```

n = 31
if verifie_collatz(n):
    print("""C'est bon pour "" + str(n))

```

Remarque : vous noterez que, d'un point de vue logique, nous ne sommes pas du tout assurés que l'exécution de cette fonction puisse systématiquement s'achever (car si la conjecture de SYRACUSE était fausse...).