

Introduction à Python

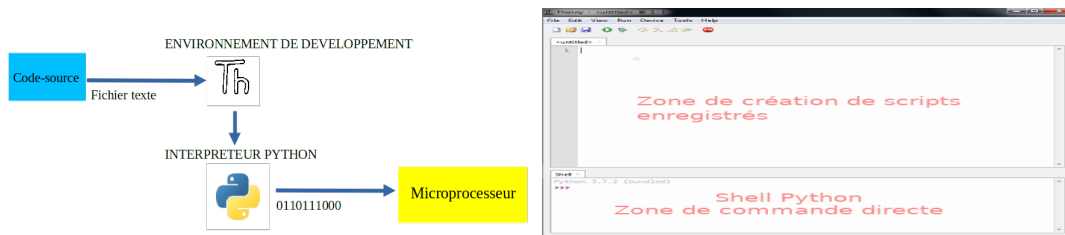


FIGURE 1.1 – Principe de Thonny

1.1

Programmer en Python ?

1. directement avec l'**interpréteur Python**. Il s'agit du programme qui pourra lire et interpréter vos **codes-sources Python**. Il les traduira en code machine composé de 0 et de 1
2. avec un *Environnement de Développement*, comme **Thonny** ou **IDLE**. Voir FIGURE 1.1 page 1
3. en utilisant un site Web adapté, comme **Basthon**.

1.2

Thonny

1.2.1 Fenêtre de prog ou console

On distingue (voir FIGURE 1.1 page 1) :

- La **zone programmation** en haut. Permet de créer des programmes : du code-source qu'on exécute avec la flèche verte.
- La **zone interactive** en bas. Les chevrons > > > indiquent que l'interpréteur est prêt à exécuter *immédiatement mais une seule fois* l'instruction fournie.

1.2.2 Les types d'erreurs

Lorsqu'on fournit un texte à l'interpréteur, il analyse la demande. Plusieurs cas possibles :

- **Pas d'erreur** : la demande est bien formulée et la réponse de l'interpréteur correspond à ce que nous voulions qu'il fasse.

```
>>> 3.5 + 1.0
4.5
```

- **Erreur de syntaxe** : la demande ne correspond pas aux règles de syntaxe du langage. L'interpréteur ne comprend pas la demande!

```
>>> Ca va bien ?
SyntaxError: invalid syntax
```

- **Erreur d'exécution** : demande est bien formulée mais qui provoque une erreur lors de l'exécution.

```
>>> 12 / (3 - 3)
ZeroDivisionError: division by zero
```

- **Erreur sémantique** : votre interprétation de la demande n'est pas la même que celle de l'interpréteur. Sa réponse est donc "bonne" mais ne correspond pas à ce que vous vouliez faire.

```
>>> 5,2 + 1,2
(5, 3, 2)
```

1.2.3 Fonctionnement de la console

L'algorithme est basé sur une boucle infinie :

TANT QUE VRAI

1. **ATTENTE** : de la touche ENTREE
2. **INTERPRETATION** : interprète la demande
3. **AFFICHAGE** : résultat ou erreur

La console a été programmée pour afficher le résultat final obtenu.

Pourquoi programmer ?

L'informatique est une science qui consiste à résoudre les problèmes. Elle est composée de plusieurs piliers. Voir FIGURE 1.2 page 2)

- **les données** : pour travailler sur un problème, il faut parvenir à encoder les informations liées à ce problème.
- **l'algorithmique** : la science de la résolution de problèmes ((rapidement, correctement et machinalement))
- **la machine** : comment fonctionne-t-elle ?
- **le langage** : comment communiquer sans ambiguïté ?

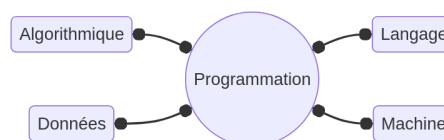


FIGURE 1.2 – 4 piliers

Chacun de ces piliers fait partie de l'informatique. C'est la **programmation** qui les relie entre eux.

Rappels de programmation de seconde ?

1.4.1 Programmation dans Thonny

Il faut écrire le programme dans la fenêtre du haut, sauvegarder et lancer le programme à l'aide de la **flèche verte**.

```

1 a = 10
2 b = 15
3 c = (a + b) / 2
4 print(c)
  
```

- L1 : place une note dans a
- L2 : place une autre note dans b
- L3 : calcule la moyenne et la place dans c
- L4 : affiche le contenu de c via print()

1.4.2 VARIABLE - Affectation

Pour affecter un contenu à un nom de variable, on réalise ce qu'on nomme une **affectation**.

Exemple : **a = 10**. Ceci permet de créer une variable nommée **a** qui référence la valeur 10. Notez bien que c'est 10 qu'on associe à **a** : on va **de la droite vers la gauche**. C'est pour cela qu'on écrit **a ← 10** dans un algorithme.

1.4.3 VARIABLE - Incrémentation

Incrémenter veut dire modifier la valeur en lui rajoutant une autre valeur.

Incrémenter de 1 veut dire qu'on augmente la valeur de 1 : 45 passe à 46. Ce +1 est la **valeur** par défaut en l'absence d'information.

Incrémenter de 2 veut dire qu'on augmente la valeur de 2 : 45 passe à 47.

Incrémenter de -1 veut dire qu'on augmente la valeur de -1 : 45 passe à 44. On dit plutôt **décrémenter**.

Pour incrémenter a de 1, on écrit **a = a + 1**

- ... = a ... : évaluer la valeur actuelle de a,
- ... = a + 1 : évaluer cette valeur + 1 et
- a = ... : affecter cette évaluation à a.

1.4.4 Instruction conditionnelle

Une **instruction conditionnelle** permet d'exécuter un bloc d'instructions uniquement si la condition évaluée est vraie (**True** en Python).

Ci-dessous, un exemple où on change l'appréciation si la note est bonne (if en anglais)

```

1 fatigue = "En forme !"
2 appreciation = "Moyen"
3 note = 20
4 if note > 15 :
5     appreciation = "Très bien"
6 fatigue = "Grosse fatigue"
  
```

TABULATION : le bloc d'instructions à effectuer est décalé de 4 espaces (ligne 5). Ce décalage permet à l'interpréteur Python de signaler le début et la fin du bloc. La ligne 6 n'appartient donc pas au bloc.

Traduction :

- L4 : si la **note** est strictement supérieure à 15
- L5 tabulée : alors **appreciation** passe à "Très bien".
- L6 : **fatigue** passe toujours à "Grosse fatigue"

1.4.5 Fonction : notion

ENTREE(S) ⇒ Fonction ⇒ SORTIE

1.4.6 Fonction : native

C'est une fonction présente de base dans Python. On en fait l'appel en notant son nom et en fournissant entre parenthèses les entrées séparées par des virgules.

```
>>> max(10, 0, 50, 40)
50
```

1.4.7 Fonction : personnelle

On distingue deux choses :

- A - La déclaration (mise en mémoire)
- B - L'appel (utilisation)

A - Déclarer une fonction revient à stocker des instructions.

On écrit dans l'ordre

- le mot-clé **def**.
- le nom de la fonction
- entre parenthèses : les variables d'entrée séparées par des virgules
- le caractère :
- les instructions du bloc tabulé

```
1 def double(x) :
2     return 2 * x
```

Le mot-clé **return** indique ce que la fonction va renvoyer.

B - Lancer un appel de fonction revient à fournir les entrées et attendre la réponse de la fonction.

Sur l'exemple, l'appel se fait LIGNE 4.

```
1 def double(x) :           # Déclaration
2     return 2 * x         # Déclaration
3
4 n = double(12)           # Appel
```

Traduction et déroulé

L1(déclaration) L4(appel)-L1-L2-L4(réception)

- L1 : déclaration d'une fonction **double()** qui va recevoir une donnée qu'on placera dans une variable **x**.
- L2 : la fonction devra renvoyer le double de **x**.
- L4 (appel à la fonction **double()** en lui envoyant 12)
- L1 L'interpréteur place 12 dans **x**.
- L2 : la fonction envoie 24
- L4 (réception) : on stocke la réponse 24 dans une variable **n**.

1.4.8 Fonction : plus complexe

Voici une fonction qui récupère la valeur d'une note, rajoute 5, bloque la note à 20 si elle dépassait 20 et renvoie le résultat final.

```
1 def nouvelle_note(note) :
2     note = note + 5
3     if note > 20 :
4         note = 20
5     return note
6
7 n = nouvelle_note(12)
```

Traduction et déroulé

L1(déclaration) L7(appel)-L1-L2-L3-L5-L7(réception)

- L1 : déclaration de **nouvelle_note()** qui va recevoir une donnée qu'on placera dans **note**.
- L7 (appel à la fonction **nouvelle_note()** en lui envoyant 12)
- L1-5 : on exécute les lignes avec **note** contenant 12 initialement. Après la ligne 2, **note** contient donc 17. Le test de la condition étant alors fausse (17 n'est pas supérieur à 20), on passe en ligne 5 : la fonction envoie 17.
- L7 (réception) : on stocke 17 dans **n**.

1.4.9 EXACTEMENT la même action

Pour réaliser **plusieurs fois exactement la même série d'actions**, il suffit d'utiliser une boucle bornée POUR, qu'on déclare à l'aide du mot-clé **for**.

```
1 print("Avant boucle")
2
3 for _ in range(5) :
4     print("A")
5     print("-- B")
6
7 print("Après boucle")
```

Déroulé

- L1
- L3 - L4 - L5
- L3 - L4 - L5
- L3 - L4 - L5
- L3 - L4 - L5
- L3 - L4 - L5
- L3 - L4 - L5
- L7

Résultat du programme

```
Avant boucle
A
-- B
A
-- B
A
-- B
A
-- B
A
-- B
Après boucle
```

1.4.10 PRESQUE la même action

A - Variable de boucle

On va réaliser **des actions un peu différentes** en fonction d'une **variable de boucle** dont le contenu va évoluer à chaque tour de boucle. On place le nom de cette variable derrière le mot clé **for**.

Ci-dessous, **k** est donc la variable de boucle.

B - Valeurs successives de la variable

```
for k in range(5):
```

Avec cette déclaration de boucle, la variable de boucle **k** commence à 0 et est incrémentée de 1 à chaque tour de boucle : 0 puis 1 puis 2 puis 3 puis 4. Elle n'ira pas jusqu'à 5 qui est **une borne exclue**.

Attention : le 5 correspond bien au nombre de tours de boucle effectués : 0-1-2-3-4.

C - Déroulé avec variable de boucle

```
1 print("Avant boucle")
2
3 for k in range(3) :
4     print("A")
5     print(k)
6
7 print("Après boucle")
```

Déroulé

- L1
- L3(k=0) - L4 - L5
- L3(k=1) - L4 - L5
- L3(k=2) - L4 - L5
- L7

Résultat du programme

```
Avant boucle
A
0
A
1
A
2
Après la boucle
```

1.4.11 boucle TANT QUE

A - Principe Il s'agit de réaliser une boucle sans prévoir à l'avance combien de tours on va réaliser exactement.

A chaque début de tour, on teste une condition (dite de poursuite) :

- si elle est vraie, on fait un tour.
- si elle est fausse, on quitte la boucle.

Le mot-clé **while** déclare cette boucle TANT QUE. C'est l'indentation qui renseigne sur le bloc à réaliser :

```
1 gain = 10
2 nb_mois = 0
3
4 while gain < 5000 :
5     gain = gain * 2
6     nb_mois = nb_mois + 1
7
8 print(gain)
9 print(nb_mois)
```

Principe de la boucle

- L4 : évaluation de la condition de poursuite.
- SI son évaluation donne True, on réalise L5-6 puis on revient à L4 et on évalue à nouveau.
- SINON, on quitte et part en ligne 8.

B - Un classique : valeur de seuil Le programme précédent permet de répondre à ce problème : on considère un gain initial de 10 euros. On double le gain tous les mois. Tant qu'il y a moins de 5000 euros, on laisse la somme sur le compte. Au bout de combien de mois va-t-on atteindre 5000 euros ou plus ?

Si on lance le programme :

```
>>> %Run boucleTQ.py
5120
9
```

Traduction et déroulé

- L1-2 : initialisation à 10 et 0 des variables.
- L4 : évaluation à True car gain vaut 10.
- L5-6 : gain passe à 20 et nb_mois à 1.
- L4 : évaluation à True car gain vaut 20.
- L5-6 : gain passe à 40 et nb_mois à 2.
- ...
- L4 : évaluation à True car gain vaut 2560.
- L5-6 : gain passe à 5120 et nb_mois à 9.
- L4 : évaluation à False car gain vaut 5120.
- L8-9 : affichage des variables