

Introduction à la Programmation en Python

Adeline Pierrot

June 6, 2019

Sommaire

- Introduction générale
- Principes d'un programme Python
- Variables, Types
- Expressions (arithmétiques, booléennes, priorité)
- Instructions (affectation, affichage, lecture)
- Conditionnelles
- Boucles
- Listes
- Fonctions

Constat : les ordinateurs sont partout

- Ordinateurs de bureau (fixes ou portables)
- Station de travail
- Clusters d'ordinateurs

Mais aussi:

- Tablettes
- Calculatrices (programmables)
- Téléphones portables, appareils photos, GPS, lecteurs MP3, ...
- Puces (cartes bancaires...)
- Box, routeurs wifi, ...

A quoi sert un ordinateur ?

Entre autres :

- Stocker de l'information
Exemple : textes, musiques, images...
- Automatiser certaines tâches, en particulier des calculs
Exemple : Calcul de la moyenne aux concours des milliers d'élèves les ayant passés, et classement.

Avantages de l'ordinateur:

- grande mémoire et rapidité de calcul

Inconvénients:

- Un ordinateur n'est pas intelligent ! Il se contente de faire exactement ce qu'on lui a demandé, même si c'est stupide
↪ **c'est au programmeur (vous !) d'être intelligent à la place de l'ordinateur !**

Traiter automatiquement des informations

- **Entrée**: informations venant du clavier, de la souris, de capteurs, de la mémoire, d'autres ordinateurs, ...
- Traitement des informations en exécutant un **programme**
- **Sortie**: information envoyées vers l'écran, la mémoire, d'autres ordinateurs, ...

Qu'est-ce qu'un programme ?

Informellement, un programme est une **séquence d'instructions** qui spécifie **étape par étape** les opérations à effectuer pour obtenir à partir des **entrées** un **résultat** (la **sortie**).

Langages de programmation

- Moyen de communiquer des instructions à l'ordinateur.
- Il en existe plein : environ un nouveau langage par semaine depuis 50 ans!
- Chacun a ses avantages et inconvénients.
- Heureusement les concepts sont presque toujours les mêmes d'un langage à l'autre.

But du cours d'informatique:

- Apprendre les **concepts** généraux, pour pouvoir **s'adapter** à n'importe quel langage en cas de besoin.
- être capable concrètement d'**écrire des programmes** dans un langage de programmation (Python).

Utilisation réelle de l'informatique:

Programmes avec des millions de lignes

Ce qu'on enseigne:

Apprentissage des concepts sur des problèmes simples et déjà résolus

Difficulté:

Apprendre des outils conçus pour les programmes de 100000 lignes en travaillant sur des programmes de 10 lignes...

↪ Faire preuve de **rigueur** et chercher à **comprendre en profondeur**

↪ Importance des TP pour bien **comprendre par soi-même**

Principe général d'un programme (Python):

- Chaque ligne correspond à une **instruction** (= ordre donné à l'ordinateur).
- Lorsqu'un lance un programme, l'ordinateur l'effectue étape par étape, **ligne par ligne**.
- La seule chose conservée d'une étape à l'autre est l'état de la mémoire
- Pour stocker le résultat d'une étape intermédiaire, on utilise les **variables**.

Exemple de programme Python

```
x = 4  
print("x vaut",x)
```

Le même programme en C

```
int main() {  
    int x = 4;  
    printf("x vaut %d",x);  
    return 0;  
}
```

Exemple de conditionnelle en Python

```
if x >= 0:
    a = x
    print("x est positif, de valeur absolue", a)
else:
    a = -x
    print("x est négatif, de valeur absolue", a)
print("Fin du programme")
```

Le même bout de programme en C

```
if (x >= 0) {
    a = x;
    printf("x est positif, de valeur absolue %d",a);
} else {
    a = -x;
    printf("x est négatif, de valeur absolue %d",a);
}
printf("Fin du programme");
```

Spécificités de Python

- Langage interprété (et pas compilé comme C)
- Pas de main
- La fin d'une instruction est indiquée par un retour à la ligne (et non un point-virgule contrairement à C ou Java)
- Les blocs d'instructions sont définis par l'indentation (et non par des accolades contrairement à C ou Java)
- Pas besoin de parenthèses autour de la condition d'un `if` ou d'un `while`, mais besoin d'un `:` après.
- On ne déclare pas le type des variables

Variable

Sert à **stocker de l'information** durant l'exécution d'un programme.

Une variable possède trois propriétés:

- un **nom** (ou **identificateur**)
 - sert à désigner la variable
 - est choisi par le programmeur
 - doit commencer par une lettre, mais peut contenir des chiffres.
- une **valeur**
 - correspond à l'information qu'on veut stocker
 - peut changer en cours d'exécution du programme (d'où le nom de variable)
- un **type**
 - attribué automatiquement par Python lors de l'initialisation
 - sert à savoir comment doit se comporter la variable (quelles sont les valeurs possibles, comment on la stocke en mémoire, quel effet ont les opérateurs, par ex + ou * ...).

Notion de type

Les variables peuvent contenir toutes sortes de données différentes.

Le **type** d'une variable correspond à la sorte de donnée qu'elle contient.

Voici différents types de base:

- Les entiers (**int**) Ex: 1, 42, -32765
- les réels (**float**) Ex: 10.43, -1.0324432
- les chaînes de caractères (**str**) Ex: "Bonjour", "toi et lui", 'a'
- les booléens (**bool**) : True, False

Types (suite)

L'effet des opérateurs dépend du type des variables.

Ex : $a+b$ correspond à

- la somme si a et b sont des entiers (ou des réels)
- la concaténation si a et b sont des chaînes de caractères:

```
a = "2"
```

```
b = "3"
```

```
print(a+b)
```

```
> 23
```

Ne pas confondre le nom et la valeur des chaînes de caractères!

```
Paul = "Pierre"
```

```
x = Paul
```

```
print(x)
```

```
> Pierre
```

Les expressions

Expression: Combinaison de *valeurs* par des *opérations* donnant une nouvelle *valeur*

Exemple: L'expression $3 * (1 + 3) + (1 + 4) * (2 + 4)$ vaut 42

Plus précisément, une expression peut être:

- une valeur constante
Exemples: 2 , 56.7 , 'u' ou True
- une variable
- toute combinaison d'opérations valides mettant en œuvre des constantes et/ou des variables

Exemple: $2 * y + 4 \leq a$

Expressions arithmétiques

Opérations sur les entiers:

opération	exemple	résultat
opposé	$-(-5)$	5
addition	$17 + 5$	22
soustraction	$17 - 5$	12
multiplication	$17 * 5$	85
division entière (quotient)	$17 // 5$	3
reste de la division entière	$17 \% 5$	2
exponentiation (puissance)	$5 ** 2$	25

Attention, la multiplication n'est pas implicite, le symbole $*$ doit toujours être indiqué explicitement entre les deux opérandes

Expressions booléennes

Une variable booléenne ne peut prendre que deux valeurs: True et False.

Opérations sur les booléens:

opération	exemple	résultat
négation (non)	not True	False
conjonction (et)	True and False	False
disjonction (ou)	True or False	True

négation : valeur contraire

conjonction : vrai si et seulement si les deux sont vrais

disjonction : vrai si et seulement si au moins l'un des deux est vrai

Expressions booléennes: comparaisons

La condition dans une expression booléenne résulte dans la majorité des cas d'une ou plusieurs comparaisons:

symbole Python	symbole mathématique
<	<
<=	≤
==	=
!=	≠
>=	≥
>	>

Attention, ne pas confondre = (affectation) et == (comparaison)

Évaluation paresseuse des expressions booléennes

Exemple: Quelle est la valeur des expressions suivantes:

- `False and (3*x + 1 >= 2 or 1/(1+x) < 42)`
- `True or (3*x + 1 >= 2 or 1/(1+x) < 42)`

Deux possibilités:

- **l'évaluation complète:** évaluer tous les opérandes des expressions booléennes
- **l'évaluation paresseuse:** stopper l'évaluation dès que possible:
 - Pour une conjonction `a and b` on peut s'arrêter si `a` est faux
 - Pour une disjonction `a or b` on peut s'arrêter si `a` est vrai

Python utilise l'évaluation paresseuse.

Ordre de priorité

Valeur des expressions suivantes:

- `6 / 3 * 2` vaut 4
- `6 + 3 * 2` vaut 12
- `3 + 4 <= 2 * 8` vaut True
- `not 1 < 2 and 1 == 2` vaut False

Les expressions sont évaluées de gauche à droite suivant l'ordre de priorité décroissante suivant:

* / %
+ -
<, <=, ==, !=, >=, >
not
and
or

Le parenthésage

Les parenthèses servent à modifier l'ordre de priorité

Exemple

opération	valeur
$5 + 4 * 2$	13
$(5 + 4) * 2$	18

Instruction et Expression

Expression: correspond à une valeur

Ex : $2+3*x$

Instruction: correspond à une action (ordre pour l'ordinateur)

Ex : `print(x)`

Une instruction se trouve seule sur une ligne, tandis qu'une expression ne doit pas se trouver seule sur une ligne.

Parmi les instructions de base : l'affichage et l'affectation de variable.

Affectation de variable

Format:

Nom de variable = Expression

Ex : $x = 2 * y + 3$

~~$f(x) = x + 3$~~ **incorrect**: la partie gauche doit être un nom de variable

~~$x + 1 = 3$~~ **incorrect**: la partie gauche doit être un nom de variable

Effet:

L'ordinateur calcule la valeur de l'expression, puis donne cette valeur à la variable dont le nom est indiqué à gauche du signe =

↪ **En informatique, $x=y$ et $y=x$ ne signifient pas la même chose !!**

```
x = 2
```

```
y = 4
```

```
x = y
```

```
print(x)
```

```
> 4
```

```
x = 2
```

```
y = 4
```

```
y = x
```

```
print(x)
```

```
> 2
```

Raccourci

La syntaxe `i++` n'existe pas en Python.

Par contre on peut utiliser le raccourci `i+=1`
pour remplacer l'instruction `i=i+1`,
et plus généralement `x+=expression`
pour remplacer l'instruction `x=x+expression`.

De même avec `--`

(Raccourci présenté uniquement pour information, pas nécessaire de le savoir et inutile d'en parler aux élèves de lycée)

Affichage

- Se fait à l'aide de la fonction `print`.

Ex : `print(x)`

- Par défaut, Python va à la ligne après l'instruction `print`.
On peut lui spécifier de faire autrement.

Ex : `print(x, end=' ')` pour mettre un espace à la fin
(et pas de retour à la ligne).

- On peut mettre plusieurs éléments dans une même instruction `print`. Ils seront alors séparés par des espaces (et par défaut Python ira à la ligne une fois à la fin de l'instruction `print`).

Ex : `print("x vaut", x, "et y vaut", y)`

Lecture

- La fonction `input` permet de récupérer une valeur entrée au clavier.

Il faut ensuite faire quelque chose de cette valeur, par exemple la stocker dans une variable.

Ex : `x = input()`

- Par défaut, la valeur entrée est considérée comme une chaîne de caractères.

Si elle correspond à autre chose, par exemple un entier, il faut la convertir.

Ex : `x = int(input())`

- Il est fortement recommandé de toujours indiquer à l'utilisateur ce qu'il doit entrer quand on utilise `input`

Ex : `x = int(input("Entrez un entier "))`

Conditionnelles

```
if age < 0:
    print("Erreur")
elif age < 12:
    print("Enfant")
elif age < 18:
    print("Adolescent")
else:
    print("Adulte")
```

```
if age < 0:
    print("Erreur")
if 0 <= age < 12:
    print("Enfant")
if 12 <= age < 18:
    print("Adolescent")
else:
    print("Adulte")
```

```
if age < 0:
    print("Erreur")
if 0 <= age < 12:
    print("Enfant")
if 12 <= age < 18:
    print("Adolescent")
if age >= 18:
    print("Adulte")
```

Les deux programmes du haut ont exactement le même comportement.

Celui du bas n'est pas correct.

Boucles

Boucle	while	for
Python	while condition:	for i in range(a,b,c):
C	while (condition){	for (int i=a;i<b;i=i+c){

Choisir entre une boucle for et une boucle while

- Si on connaît à l'avance le nombre de répétitions à effectuer, ou plus généralement, si on veut parcourir une valeur itérable, on choisit une boucle for.
- A l'inverse, si la décision d'arrêter la boucle ne peut s'exprimer que par un test, c'est la boucle while qu'il faut choisir.

Itérables

En Python, une boucle for parcourt un itérable:

```
for element in iterable:
```

Par exemple, si L est une liste, on peut écrire:

```
for x in L:  
    print(x)
```

L'expression `range(n)` est un itérable, défini ainsi:

- Son premier élément est 0
- Son dernier élément est $n-1$
- Après l'élément i vient l'élément $i+1$.

De même, `range(m, n)` est un itérable de premier élément m et dernier élément $n-1$ (il est donc vide si $m \geq n$)

et `range(m, n, p)` est un itérable où après i vient $i+p$: de m inclus à n exclu par pas de p (si $p < 0$, alors le dernier élément est $n+1$).

Boucles imbriquées

Qu'affiche ce programme ?

```
for i in range(3):  
    for j in range(2):  
        print(i, j)
```

Réponse :

```
0 0  
0 1  
1 0  
1 1  
2 0  
2 1
```

Listes Python

- Une liste Python se définit par la donnée explicite de ses éléments entre crochets `[]`.

- Ses éléments peuvent être de n'importe quel type :

```
L = [3, 2.5, True, [1,2,3], "bonjour"]
```

- La fonction `len()` retourne le nombre d'éléments d'une liste :

```
len(L)
```

```
> 5
```

- On accède aux éléments par leur indice, entre crochets.

Le premier élément a pour indice 0.

Le dernier élément a pour indice `len(L) - 1`

```
L[0]
```

```
> 3
```

```
L[len(L)-1]
```

```
> "bonjour"
```

Exemple Listes Python

```
L = [1,10,-2]
somme = 0
for i in range(len(L)) :
    somme = somme + L[i]
print(somme)

> 9
```

Le fait que `range(n)` va de 0 à $n-1$ est pratique pour traiter les listes.

Fonctions

Dans les vrais programmes informatiques, on est souvent amené à utiliser plusieurs fois le même enchaînement d'instructions. Pour pouvoir les réutiliser facilement sans avoir à les ré-écrire, on en fait une fonction.

Bien comprendre la différence entre la **définition** d'une fonction et son **appel**. Une fonction qui est définie mais jamais appelée ne sera pas exécutée !

Certaines fonctions sont prédéfinies en Python (print, input...).

Une fonction peut prendre des **paramètres** en entrée, ou aucun, mais dans tous les cas il faut mettre les parenthèses de la fonction lors de son appel.

Si lors de l'exécution d'une fonction on arrive à une instruction **return**, ce return **stoppe la fonction** et renvoie la valeur correspondante.

Appels de fonctions

Il y a plusieurs sortes de fonctions:

- Une fonction qui contient un `return` correspond à une valeur.
Son appel est une expression.
Il ne doit pas être seul sur une ligne.
- Une fonction sans `return` n'est pas associée à une valeur.
Son appel est une instruction.
Il est utilisé seul sur une ligne.

Exemple de fonction qui...

	affiche	renvoie
demande à l'utili- sateur	<pre>def fonc(): x=int(input("x?")) print 3 * x</pre>	<pre>def fonc(): x=int(input("x?")) return 3 * x</pre>
appel	<pre>fonc()</pre>	<pre>print(fonc()) y = 2 + fonc()</pre>
prend en para- mètre	<pre>def fonc(x): print 3 * x</pre>	<pre>def fonc(x): return 3 * x</pre>
appel	<pre>fonc(5)</pre>	<pre>print(fonc(5)) y = 2 + fonc(5)</pre>

paramètre formel \neq paramètre effectif

Paramètre formel : utilisé dans la **définition** de la fonction.

Un paramètre formel n'a pas d'existence "réelle". C'est juste une **notation** qui nous sert à définir la fonction.

En particulier un paramètre formel n'a pas de valeur !

Paramètre effectif : utilisé lors d'un **appel** de la fonction.

C'est un paramètre sur lequel on va effectivement lancer les calculs de la fonction. Le paramètre effectif peut être une variable, une constante, une expression comme $3*x+2...$

Un paramètre effectif doit avoir une **valeur** !

Exemple

```
def triple(x):  
    print(3*x)  
  
triple(4)  
y = 2  
triple(y)  
triple(3*4-5)  
print(3*x) incorrect: x n'existe pas ici !
```

Ici la fonction “triple” a un paramètre formel nommé x , il y a donc un paramètre effectif pour chaque appel de la fonction. Le premier paramètre effectif est la constante 4, le deuxième est la variable y (qui a pour valeur 2) et le troisième est l'expression $3*4-5$.