Chapitre 1: Variables, types et boucles

Brice Mayag
brice.mayag@dauphine.fr

M1 SIREN: maj. SIEE/GSI

Dauphine | PSL
UNIVERSITÉ PARIS

Sommaire

Sommaire

Les instructions d'un algorithme ont généralement pour but de traiter des données d'entrée (numériques, textes, images, ...) qui peuvent être saisies par l'utilisateur, résulter d'un calcul de l'ordinateur, ...

Par variable on désigne le fait d'attribuer un nom ou un identifiant à une information.

Les espaces mémoire où ces données sont conservées sont appelés variables.

Les variables sont essentielles pour pouvoir écrire un programme informatique.

Définition

Une variable est définie par trois composantes :

- un identifiant (le nom de la variable)
- un type de données
- une valeur courante, dans le domaine des valeurs du type

En langage algorithmique, on utilise le symbole $\longleftarrow$ pour indiquer qu'une valeur est affectée à une variable : $a \longleftarrow 1$.

En Python, on utilise le symbole "=", qui s'appelle dans ce contexte opérateur d'affectation.

Les mots clé réservés du langage ne peuvent pas être utilisés comme noms de variables. Par exemple and, as, break, class, def, if, else, while, for, return, None, True, False

In [18]:
```
age=26 # 'age' est l'identifiant de la variable
print(age)
type(age)
```

```
26
```

Out[18]: int

In [23]:
```
taille=1.80 # 'taille' est l'identifiant de la variable
print(taille)
type(taille)
```

```
1.8
```

Out[23]: float

In [20]:
```
Nom='Turing' # 'Nom' est l'identifiant de la variable
print(Nom)
type(Nom)
```

```
Turing
```

Out[20]: str

Le type indique le contenu de la variable, et permet à l'interpréteur de Python de savoir comment la manipuler. En Python, le typage est dynamique, il est implicitement lié à l'information manipulée (voir les trois exemples ci-dessus).

Python possède un certain nombre de types de variables fondamentaux ou déjà définis, à partir desquels il sera possible de définir aussi ses propres types

Les variables permettent de manipuler les informations beaucoup plus facilement.

L'autre avantage des variables est de pouvoir écrire des programmes pour des valeurs qui varient.

Les variables jouent un rôle similaire à celui des arguments d'une fonction mathématique.

La valeur d'une variable peut être au choix : une constante, une autre variable ou une expression

In [22]:
```
# Exemple 1 : somme des n premiers entiers
n = 12
somme = 0 # initialisation : la somme est nulle
for i in range(1,n): # pour tous les indices de 1 à n
    somme=somme+i # on ajoute le i-ème entier à la somme
print(somme)
```

```
66
```

Le symbôle ♯ marque le début d'un commentaire. Les commentaires aident à la compréhension d'un programme, mais n'en font pas partie.

Python impose une instruction par ligne.

Déclaration et initialisation d'une variable

Dans la plupart des langages de programmation, pour pouvoir utiliser une variable, il faut procéder en deux temps :

1. Déclarer la variable avec un certain type
2. lui affecter une valeur

En Python, il n'est pas nécessaire de déclarer explicitement les variables en précisant leur type. Le typage est automatique, en fonction de la donnée affectée à la variable.

Sommaire

Les structures de données

Structure de données

Une structure de données est définie par 3 choses :

1. Un **type** qui est simplement un nom permettant de classifier les données relevant de ce nom.
2. Un **ensemble** qui définit avec précision les éléments appartenant au type.
3. des **opérations** utilisées par les programmes pour calculer sur ces données

On parle de structures algébriques.

Les booléens

Ce type de données est prédéfini dans Python par :

- son nom bool
- ses valeurs {True,False}
- ses opérations
  - not : $bool \rightarrow bool$,
  - and : $bool \times bool \rightarrow bool$, et
  - or : $bool \times bool \rightarrow bool$

Nom vient de George Boole [1815-1864] mathématicien anglais intéressé par les propriétés algébriques des valeurs de vérité.

Les entiers relatifs

Ce type de données est prédéfini dans Python par :

- son nom int
- son ensemble de valeurs est théoriquement $\mathbb{Z}$ mais est en fait borné selon la machine utilisée.
- ses opérations
  - +, -, *, /, %, ** : $int \times int \rightarrow int$,
  - ==, >, <, >=, <=, != : $int \times int \rightarrow bool$

Les opérations de calcul (+,-,** ...) sont prioritaires sur les opérateurs de comparaison (==,<=, ...).

Les nombres réels

En fait, les nombres réels ne sont pas représentables en machine. On les approxime par des nombres rationnels (i.e. avec des virgules).
Ce type de données est prédéfini dans Python par :

- son nom float
- son ensemble de valeurs est théoriquement $\mathbb{R}$ mais est en fait un sous-ensemble de $\mathbb{Q}$. De plus, il est borné selon la machine utilisée.
- ses opérations
  - +, -, *, /, ** : $float \times float \rightarrow float$,
  - int : $float \rightarrow int$, et float : $int \rightarrow float$
  - ==, >, <, >=, <=, != : $float \times float \rightarrow bool$

Les flottants s'écrivent en utilisant le point pour séparer la partie décimale de la partie entière.

In [3]:
```
x = 3
y = 3.0
print("x =", x, "est de type ", type(x))
print("y =", y, "est de type ", type(y))
```

```
x = 3 est de type  <class 'int'>
y = 3.0 est de type  <class 'float'>
```

Les chaînes de caractères

Type immuable (on ne peut pas changer les valeurs)

Description formelle

Une chaîne de caractères est une application $s : S \to char$ où S est une séquence finie d'entiers positifs $\{0, \ldots, n\}$ et char est l'ensemble des 256 caractères ascii.

Ce type de données est prédéfini dans Python par :

- son nom str
- son ensemble de valeurs est tout mot fini (i.e. une suite de caractères ascii).
- ses opérations
  - $+ : str \times str \to str$ (concaténation de deux chaînes de caractères),
  - $* : str \times int \to str$ l'entier doit être positif,
  - len : $str \to int$ (longueur d'une chaîne de caractères),
  - $==, >, <, >=, <=, != : str \times str \to bool$

Les chaînes de caractères - suite

En python, une chaîne se décrit en extension en écrivant le mot avec des guillemets : $s =$ "$a_1 \dots a_n$".

La notation $s[i]$ pour $i = 0, \dots, (n-1)$ dénote la lettre à la position $i+1$ dans la chaîne s.

On peut accéder à des sous chaînes de caractères d'une chaîne donnée. Soit s une chaîne de longueur n. s[i :j] avec $i < j$ et $i, j \leq n$, dénote la sous-chaîne $s[i]s[i+1] \dots s[j-1]$

In [24]:
```
t="Ceci est un texte"
print(t, type(t))
```

```
Ceci est un texte <class 'str'>
```

In [25]:
```
t='Ceci est un texte introduit avec des apostrophes'
print(t, type(t))
```

```
Ceci est un texte introduit avec des apostrophes <class 'str'>
```

Plusieurs fonctions et opérateurs permettent de manipuler une ou plusieurs chaînes de caractères :

- la fonction `str` permet de convertir un nombre, un vecteur, ou un autre objet en chaîne de caractères
- l'opérateur `+` permet la concaténation de chaînes de caractères
- l'opérateur `*`permet la répétition d'une chaîne de caractères

- les fonctions `in` et `not in` permettent de tester si une chaîne de caractères en contient une autre
- `[n]` permet d'extraire le (n+1)-ème caractère de la chaîne (le premier a pour indice 0)
- `[i:j]` permet d'extraire les caractères des positions `i` à `j-1` de la chaîne de caractères.
- ...

In [6]:
```
x = 6.251
s=str(x)
print(x, type(x))
print(s, type(s))
# print(len(x))
print(len(s))
print(s[2])
print(s[1:4])
print(s[0:4])
```

```
6.251 <class 'float'>
6.251 <class 'str'>
5
2
.25
6.25
```

Les listes

Description formelle

Une liste est une application $s : S \to TYPE$ où S est une séquences d'entiers positifs $\{0, \ldots, n\}$ et TYPE est l'union de tous les types de données.

Ce type de données est prédéfini dans Python par :

- son nom lists
- son ensemble de valeurs est tout mot fini sur l'ensemble TYPE pris comme alphabet.
- ses opérations
  - $+$: $list \times list \to list$ (concaténation de deux listes),
  - $*$: $list \times int \to list$ l'entier doit être positif,
  - len : $list \to int$ (longueur d'une liste),
  - $==, >, <, >=, <=, != : list \times list \to bool$

En python, une liste se décrit en extension en écrivant le mot entre crochets : $l = [a_1, \ldots, a_n]$. La notation [] dénote la liste vide.

In [26]:
```
# Exemples
x = [4,5]                 # création d'une liste composée de deux entiers
print(x)
y = x [0]                 # accède au premier élément
print(y)
y = x [-1]                # accède au dernier élément
print(y)
x = ["un",1,"deux",2]     # création d'une liste composée de
                          # deux chaînes de caractères
                          # et de deux entiers, l'ordre d'écriture est important
print(x)
x = [ ]                   # crée une liste vide
print(x)
x = list ()               # crée une liste vide
print(x)
```

```
[4, 5]
4
5
['un', 1, 'deux', 2]
[]
[]
```

Plusieurs opérations sont possibles sur les listes.

Il est possible de modifier, insérer ou supprimer des éléments, ainsi que de trier les listes.

Les opérations sont similaires à celles qui s'appliquent aux chaînes de caractères.

- `x in ls` : vrai si `x` est un élément de `l`
- `x not in ls` : vrai si `x` n'est pas un élément de `l`
- `l+t` : concaténation de deux listes

- `l*n` : concaténation de `n` copies de `l`
- `min(l)`, `max(l)`, `sum(l)`
- `len(l)` : renvoie le nombre d'éléments de `l`
- `list(x)` : convertit l'objet `x` en liste lorsque cela est possible
- `l.count(x)` : renvoi le nombre d'occurences de l'élément `x` dans la liste `l` (on reviendra sur cette syntaxe particulière lorsqu'on parlera des **classes**). `x` en tant qu'élément, la liste `l` ne contiendra qu'un élément de plus.
- ...

In [51]:
```
# Exemple de tri
x = [9, 0, 3, 5, 4, 7, 8]     # définition d'une liste
print(x)                      # affiche cette liste
x.sort()                      # trie la liste par ordre croissant
print(x)                      # affiche la liste triée
```

```
[9, 0, 3, 5, 4, 7, 8]
[0, 3, 4, 5, 7, 8, 9]
```

Sommaire

Les expressions conditionnelles

Condition en python

Une expression conditionnelle python se déclare de la façon suivante

```
if expression_conditionnelle :
    Instructions
else :
    Instructions
```

Les expressions conditionnelles - suite

Dans le cas où l'on ne souhaite pas rien faire lorsque la condition est fausse, on peut omettre la partie "else".

Enfin, il arrive que l'on ait des "cascades" d'expressions conditionnelles. On peut alors utiliser le mot clé elif.

```
if expression_conditionnelle :
   Instructions
elif expression_conditionnelle :
   Instructions
else :
   Instructions
```

La clause `else` est facultative. Lorsque `condition` est fausse et il n'y a aucune instruction à exécuter dans ce cas, la clause `else` est inutile.

Exemple : Le code suivant permet de tester si un nombre est positif ou négatif :

```
x=1
if (x>=0) :
    print("le nombre est positif")
else :
    print("le nombre est négatif")
```

S'il est nécessaire d'enchaîner plusieurs tests, il est possible de condenser en utilisant le mot-clé `elif` :

```
if condition1 :
    instruction1
    instruction2
    ...
elif condition2 :
    instruction3
    instruction4
    ...
elif condition3 :
    instruction5
    ...
else :
    instruction6
    instruction7
    ...
```

Remarques

- Le décalage des instructions par rapport aux lignes contenant les mots-clé `if`, `elif`, `else` est très important. Il fait partie de la syntaxe du langage Python et s'appelle **indentation**.
- Afin de comprendre mieux le fonctionnement des tests, on peut toujours compléter l'écriture de l'algorithme pseudo-code ou du programme par un schéma!

Les boucles While et for

Boucle while

Une boucle while python se déclare de la façon suivante

```
while expression_conditionnelle :
Instructions
```

Boucle for

Une boucle for python se déclare de la façon suivante

```
for variable in collection :
Instructions
```

Les collections dans les boucles "for" peuvent être de plusieurs formes :

- "range(ind_début,ind_fin,pas)". La variable va prendre pour valeur à chaque étape de la boucle "for" ind_début,ind_début+pas,...,ind_fin.
- une chaine de caractères ou une liste. Dans ce cas là, la variable va parcourir toute les valeurs se trouvant aux positions allant de 0 à $len(s) - 1$ où s est la chaîne de caractères ou la liste.

- La boucle `for` est **déterministe** (le nombre d'itérations à exécuter est connu est fixé au départ).
- La boucle `while` est **indéterministe** (chaque itération est conditionnée par une expression booléenne, dont la valeur change au cours de l'itération).

Syntaxe d'une boucle `while`

```
while condition :
    instruction1
    instruction2
    ...
```

Tant que `condition` est vérifiée, la suite d'instruction est exécutée.

Comme pour les expressions conditionnelles, **l'indentation** est essentielle! Le décalage des lignes par rapport à l'instruction `while` permet de les inclure dans la boucle.

In [2]:
```
# Exemple sur le rôle de l'indentation
n = 0
while (n < 3) :
    print("à l'intérieur ", n)
    n += 1    # l'opérateur += est utilisé ici pour incrémenter les valeurs de
n
print("à l'extérieur ", n)
```

```
à l'intérieur  0
à l'intérieur  1
à l'intérieur  2
à l'extérieur  3
```

Remarque La condition qui régit une boucle `while` doit nécessairement être modifiée à l'intérieur de la boucle. Dans le cas contraire, on est dans le cas d'une **boucle infinie**, et il est impossible d'en sortir.

Exemple

```
n = 0
while (n<3) :
    print(n)
    n+1
```

Syntaxe d'une boucle `for`

```
for x in ensemble :
    instruction1
    instruction2
    ...
```

La suite d'instructions est exécutée pour chaque élément `x` de `ensemble`.

In [3]:
```
# Exemple 1
ensemble = (1,2,3,4)
for x in ensemble :
    print(x)
```

```
1
2
3
4
```

In [1]:
```
# Exemple 2
ls =  [4,5,3,-2,-8,-2]
somme = 0
for i in range(0, len(ls)) :
    somme += ls[i]
    print(somme)
```

```
4
9
12
10
2
0
```

In [2]:
```
# Exemple 3
ls =  [4,5,3,-2,-8,-2]
somme = 0
for i in range(0, len(ls)) :
    somme += ls[i]
print(somme)

# Remarquez ici le rôle de l'indentation!
```

```
0
```