Python : Introduction au langage

Laurent Signac

Ensip – Université de Poitiers

Laurent.Signac@univ-potiers.fr

Variables

Caractéristique des variables :

identificateur (nom) Il doit être bien choisi, surtout dans un programme long. Chaque langage dispose de règles (différentes) très précises décrivant la syntaxe des identificateurs

type Dans certains langages, le type d'une variable peut changer en cours d'exécution du programme (Python, Ruby). Dans d'autres, le type est fixé une fois pour toutes (Java, C).

portée On ne peut utiliser le nom d'une variable que dans le bloc qui la contient.

Bien nommer les variables

Always use descriptive names, the longer the scope, the longer the name.

Idiomatic Python, E. Franchi

Types de données

Types informatiques

Le type caractérise la nature de l'objet manipulé : nombre entier, à virgule, couleur, liste...

Type algorithmique $\neq$ type langage

Les langages sont restrictifs. En algorithmique, on peut se permettre d'utiliser des types en ignorant leur existence concrète.

- entiers (taille limitée en C, (presque) quelconque en Python)
- réels (flottants de taille et précision limitées en C et en Python)

Types scalaires Python

- `int` : nombres entiers (taille limitée par la RAM)
- `float` : nombres à virgule flottante IEEE 754 double précision
- `bool` : booléens (les 2 valeurs sont `True` et `False`)

Un ordinateur calcule faux

L'essentiel des calculs sur ordinateur se fait avec des flottants. Les flottants n'étant pas les réels, les calculs sur ordinateur sont intrinsèquement faux.

Affectation en Python 1

Python a un modèle d'affectation qui n'est pas habituel et pose plus de problèmes aux programmeurs expérimentés qu'aux débutants.

- Une variable est un *nom* qu'on donne à une *donnée*
- Un objet (une donnée) peut avoir plusieurs noms.
- Un nom ne peut pas désigner plusieurs objets en même temps, mais il peut désigner des objets différents à des moments différents.

L'affectation a un sens particulier en Python

Les données ont une existence en mémoire et les variables sont des références vers ces données en mémoire. Retenir : En Python, les variables sont des noms qu'on donne aux objets.

Affectation en Python 2

Opération d'affectation

`x = DATA`

- finalise la création en mémoire de l'objet `DATA`,
- puis `x` devient une référence vers cet objet (un nom pour cet objet)

En Python, une affectation ne duplique jamais un objet. Elle donne un nouveau nom à un objet existant.

Métophore qui marche bien

Les noms de variable sont des post-it qu'on déplace sur les objets

Opérations

- Opérations arithmétiques : `+ - * / % ** //`
- Opérateurs booléens : `and or not`
- Opérateurs de comparaisons : `> < == != >= <=`
- ...

Exercice ▸ invites.py invites.py

M. X a l'habitude d'organiser des réceptions. Le nombre de convives doit nécessairement être pair. De plus il n'accepte un nombre d'invités multiple de 3 que si c'est aussi un multiple de 5. Écrivez l'expression booléenne qui indique si un nombre n d'invités satisfait aux critères de M. X.

Attention aux priorités

```
True or True and False = True
(True or True)and False = False
True or (True and False)= True
```

Il y a aussi des priorités sur les opérateurs arithmétiques et de comparaison.

– Alors ? C'est un garçon ou une fille ?
– Oui

Précédence des opérations (simplifiée)

Source :
`https://docs.python.org/3/reference/expressions.html#operator-precedence`

- Boolean operators (`or`...)
- Comparisons (`>=`, `in`...)
- Bitwise operators(`^`...)
- Arithmetic (`+`, `*`...)

Les 3 expressions suivantes ont un sens, et pas la même valeur :

```
10 < 12 and 5 < 7
(10 < 12) and (5 < 7)
10 < (12 and 5) < 7
```

Instructions et expressions

- Une expression a une valeur, et (normalement) pas d'effet de bord
- Une instruction n'a pas de valeur. Elle a généralement un effet de bord

- La somme des 10 premiers entiers est une expression
- Vider le lave-vaisselle est une instruction

- `3 * i + 5` : expression
- `pgdc(105, i)+ 3` : expression
- `p = pgdc(23, 13)` : instruction

Expression / Instruction

Une affectation est une instruction ($\approx$ un ordre, une phrase à l'impératif) et non une expression ($\approx$ un groupe nominal)

Conversion d'unité - Fonctions

```python
def celcius_fahrenheit(dc) :
    df = 32 + 1.8 * dc
    return df

# Programme principal
v = float(input("Température en deg C ?"))
v = celcius_fahrenheit(v)
print("C'est équivalent à", v, "deg F")
```

▸ conversion.py

Types, déclarations...

En Python, on ne déclare pas les variables, on ne fixe pas leur type, il n'y a pas de contrainte sur les types des paramètres (le paradigme (critiquable) utilisé est le duck-typing)

Conversion d'unité - Tests

```
def celcius_fahrenheit(dc) :
    df = 32 + 1.8 * dc
    return df

# Programme principal
v = float(input("Température en deg C ?"))
if v < -273.15 :
    print("Il fait vraiment froid...")
elif v < 0 :
    print("Petite laine conseillée...")
elif v > 50 :
    print("Sous le soleil...")
else :
    print("Température supportable")
v = celcius_fahrenheit(v)
print("C'est équivalent à",v,"deg F")
```

▸ conversiontest.py

Opérateurs

Opérateurs de comparaison ordinaires (==, !=), et logique booléenne **and**, **or**, **not**)

Conversion d'unité - Boucles

```python
def celcius_fahrenheit(dc) :
    df = 32 + 1.8 * dc
    return df

# Programme principal
for c in range(-100, 100, 10) :
    print(c,"deg C =",celcius_fahrenheit(c),"deg F")
```

```python
# Autre version
c = -100
while c < 100 :
    print(c,"deg C =",celcius_fahrenheit(c),"deg F")
    c = c + 10
```

▸ conversionboucles.py

Le piège de `range`

`range(n)` est équivalent à `range(0,n)`. `range(a,b)` renvoie un objet *itérable* qui ne contient pas `b`. `range(a,b,p)` fait de même avec un pas égal à `p`.
Pour voir *ce qu'il y a dans un range* : `list(range(11,16,3))`

Indentation

Une première difficulté : l'indentation comme élément syntaxique.
Message à passer : une erreur d'indentation en Python = une parenthèse manquante en C = un begin/end mal placé en Ada ⇒ C'est une vraie **faute**

Règles

- après `:` on indente (`def`, `if`, `elif`, `else`, `while`, `for`...)
- 2 lignes appartiennent au même bloc si elles sont indentées *exactement* pareil, sans aucune ligne moins indentée entre elles.

Difficulté

Tabulation est un caractère, qui visuellement a la même largeur que 4 espaces (mais ça dépend...). Une ligne indentée avec une tabulation suivie d'une ligne indentée avec 4 espaces ne **sont pas** indentées pareil.
Réglage de l'IDE : transformer automatiquement les tabulations en 4 espaces.

Les listes

Python n'a pas de tableau, mais des listes

```
>>> l = [1,1,2,3,5,8,13]
>>> l[4]
5
>>> l.append(l[-1]+l[-2])
>>> l
[1,1,2,3,5,8,13,21]
>>> del l[0]
[1,2,3,5,8,13,21]
>>> len(l)
7
```

```
def suitelucas(suite, p, q, nbtermes) :
    while len(suite) < nbtermes :
        suite.append(p * suite[-1] - q * suite[-2])
    return suite
```

```
>>> suitelucas([0,1], 1, -1, 10))
[0, 1, 1, 2, 3, 5, 8, 13, 21, 34]
```

suitelucas.py

Chaînes de caractères

Python 3 : `str` est le type chaine Unicode

- `str` compte de nombreuses méthodes : `split`, `lower`, `replace`, `join`...
- `ord`, `chr` permettent d'accéder aux codes Unicode des caractères.
- le module `re` gère les expressions régulières

Slicing

Le slicing s'utilise sur les séquences (listes, tuples, chaînes...) : `l[debut:fin:pas]`

```
l = "abcdefghijklmnopqrst"

l[-1]      # t
l[10]      # k
l[3:10]    # defghij
l[3:]      # defghijklmnopqrst
l[3:10:2]  # dfhj
l[5:3:-1]  # fe
l[::-1]    # tsrqponmlkjihgfedcba
```

Module turtle

```
def sierp(t, l, limit=5):
    if l > limit:
        for i in range(3):
            sierp(t, l / 2, limit)
            t.lt(30)
            t.fd(l)
            t.rt(150)


def go():
    import turtle
    turtle.setup(500,500)
    t = turtle.Turtle()
    t.speed(10)
    turtle.tracer(5, 0)
    sierp(t, 200,2)
    turtle.update()
    turtle.mainloop()
```

▸ sierpinski.py