Master ReLAI
-
UE Programmation I

Jérôme Kirman

Année 2012-2013

La programmation

Programmer, c'est donner une série d'*instructions* dans un certain *langage* à une machine pour obtenir un résultat.

Un programme est un fichier texte contenant une série d'instructions.

Pour être *exécuté*, il doit être soit lu directement (par un *interpréteur*), soit *compilé* pour obtenir un fichier exécutable (instructions machine).

Langages

Un langage de programmation est défini très strictement par une *norme* donnant sa *syntaxe* et sa *sémantique*.

La syntaxe définit quels programmes sont valables, la sémantique ce qui résultera de leur exécution.

On peut classer les langages par leur proximité avec le langage machine : les langages "bas niveau" manipulent directement la mémoire et le processeur, alors que les langages "haut niveau" permettent de parler de concept plus abstraits (la traduction est automatique).

Types de langages

Il existe de nombreux langages et styles de programmation, appartenant à plusieurs familles (ou paradigmes) :

- Programmation impérative : un programme est simplement une liste d'instructions portant sur un ensemble de données, et est exécuté pas à pas.
- Programmation orientée-objets : les données sont des objets de type défini, chaque objet possède ses propres méthodes (services) servant d'interface avec son contenu interne (caché).
- Programmation fonctionnelle : tout est fonction (au sens mathématique du terme), ce qui permet d'implémenter facilement des idées bien formalisées (= mathématisées).
- Etc...

Python 3

On utilisera la version 3 du langage Python, en programmant dans un style impératif. Nos programmes seront interprétés, sans passer par une phase de compilation.

L'interpréteur

L'interpréteur Python ne connaît qu'un dialecte, et ne peut évidemment pas deviner les intentions du programmeur. Il exécutera donc aveuglément les instructions qu'on lui donne, quel qu'en soit le résultat.

C'est au programmeur de s'assurer que son programme produit bien le résultat attendu, calculé rapidement et sans erreurs.

Fonctionnement

L'interpréteur lit les instructions ligne par ligne, leur donne un sens, et effectue les opérations demandées.

On manipule des données en les stockant dans des variables, puis en effectuant des opérations sur ces variables.

Les données sur lesquelles on travaille peuvent être numériques, textuelles, logiques, etc...

Les opérations de base sont limitées, mais peuvent être combinées pour former des fonctions plus complexes, qui deviennent à leur tour des instructions simples, etc...

Notion de variable

Une *variable* est un nom "simple" donné à un espace dans la mémoire de l'ordinateur.

Plutôt que de manier des données informatiques brutes, on donne un nom à une information qu'on veut enregistrer (le contenu d'une phrase à analyser, le nombre de mots dans un dictionnaire, etc...) et on se sert ensuite de ce nom pour désigner la valeur associée.

On peut voir les variables comme des étiquettes attachées à des données.

Notion de fonction

Une *fonction* est un nom donné à une série d'instructions portant sur un ensemble de variables, et calculant un résultat (ou produisant un effet) qui dépend de la valeur de celles-ci.

L'interpréteur connaît de nombreuses fonctions "par défaut", et on peut y ajouter de nouvelles fonctions selon les besoins du programme (par exemple, une fonction qui détermine la grammaticalité d'une phrase, ou qui calcule le nombre de mots dans un dictionnaire).

Assignations

L'opérateur d'*assignation* '=' est essentiel en programmation impérative : il permet d'affecter une valeur à une variable.

Affectation de trois variables

```
x = 4
mot = "test"
y = x
```

Les deux lignes ci-dessus créent implicitement trois variables (nommées respectivement `x`, `mot` et `y`) et contenant respectivement le nombre 4, la suite de caractères 'test', et `y` la valeur représentée par `x` (c'est à dire 4).

Opérations simples

On peut effectuer toutes les opérations mathématiques de base sur des variables contenant des nombres avec les *opérateurs* '+', '−', '*', '/'.

En outre, l'opérateur '+' permet de *concaténer* (ajouter bout à bout) deux chaînes de caractères.

Opérations de base

```
x = 2*20 + 2
y = 5/7
esp = " "
sn = "un" + esp + "syntagme" + esp + "nominal"
```

Dans l'exemple ci dessus, `x` prend la valeur 42, `y` devient égal$^{(*)}$ à $5 \div 7$, `esp` est un espace, et donc `sn` devient la chaîne "un syntagme nominal".

Indices et slices

Les mots et les phrases sont stockés en mémoire sous forme d'une *chaîne de caractères*. On peut décomposer cette suite d'éléments en accédant à un caractère à la fois (par son *indice*, CàD sa position dans la série) ou à n'importe quelle sous-partie de la chaîne (grâce aux *slices*).

On peut aussi stocker une série d'informations de même nature dans une *liste*, et accéder à un élément de cette liste (ou à n'importe quelle sous-liste) de la même façon.

Comptage des positions

Pour de bonnes raisons, en informatique, le premier élément d'une série est indicé par 0. (Et non par 1.)

Python accepte des positions négatives ; dans ce cas, il compte en partant de la fin.

Indices et slices - suite

Indices

```
phrase = "Cette phrase est fausse."
lst = [4, 8, 15, 16, 23, 42]
x = lst[0]
r = lst[-1]
ponct = phrase[lst[4]]
```

Slices

```
phrase = "Cette phrase est fausse."
lst = [4, 8, 15, 16, 23, 42]
lst2 = lst[2:4]
verbe = phrase[13:16]
adjectif = phrase[-7:-1]
```

Comparaisons

Il existe un autre type d'opérateurs : les opérateurs de comparaison. Ils permettent de comparer deux variables entre elles, et ont pour résultat une *valeur de vérité*, c'est à dire vrai (True) ou faux (False).

L'opérateur '==' teste si deux valeurs sont égales, '! =' si elles sont différentes, et '<', '>', '<=', '>=' comparent les grandeurs.

Comparaisons

```
phrase = "Comparaison n'est pas raison."
vv1 = phrase[-1] == "."
vv2 = phrase[-1] != "?"
vv3 = 2+2 < 5
vv4 = vv1 == True
vv5 = 0 >= 1
```

Dans les exemples ci dessus, les variables `vv1` à `vv4` ont pour valeur "True" (les 4 premier tests à droite sont tous vrais), et `vv5` a pour valeur "False" (car 0 est plus petit que 1).

Opérateurs logiques

Un dernier type d'opérateur permet d'agir sur des valeurs de vérité : '**not**' pour la négation (la valeur n'est pas vraie), '**or**' pour la disjonction (au moins une valeur est vraie), '**and**' pour la conjonction (les deux valeurs sont vraies).

Résultat des opérateurs

vv1	vv2	**not** vv1	vv1 **or** vv2	vv1 **and** vv2
False	False	True	False	False
False	True	"	True	False
True	False	False	True	False
True	True	"	True	True

Il existe encore d'autres opérateurs (notamment '**in**', qui teste si un élément est dans un ensemble), mais ils sont d'un usage moins courant. La liste complète est trouvable aisément dans la documentation Python.

Contrôle de l'exécution

Un programme impératif exécute toutes ses instructions dans l'ordre.

Pour effectuer certaines tâches, on a besoin de choisir entre plusieurs comportements distincts, ou de répéter un ensemble de procédures plusieurs fois.

A cet effet, on utilise des *structures de contrôles* :

Conditions Exécuter ou non des instructions en fonction d'une valeur de vérité.

Boucles Répéter plusieurs fois un ensemble d'instructions à l'identique.

Fonctions Regrouper de manière logique plusieurs instructions en une seule.

Indentation

En Python, les instructions soumises à une structure de contrôle doivent être *indentées* correctement(eg. précédées de 4 espaces supplémentaires). Sinon, le programme est incorrect.

Instructions conditionnelles

Pour choisir d'effectuer ou non une série d'instructions, on utilise les instructions '**if** [test] :' et '**else** :'.

Si le test après **if** est vrai, l'interpréteur exécute les instructions indentées qui suivent ; sinon, il examine le **else** (s'il y en a) puis passe à la suite.

Type d'une phrase

```
phrase = [...]
type = "A examiner"
if (phrase[-1] == "."):
    type = "Affirmation"
else if (phrase[-1] == "?"):
    type = "Interrogation"
else if (phrase[-1] == "!"):
    type = "Exclamation"
else:
    type = "Inconnu"
```

- Note : Les parenthèses autour d'un test ne sont pas indispensables.

Boucles

Une instruction de *boucle* répète un certain nombre de fois le bloc d'instruction indenté qui la suit :

- '**while** [test] :' répète le bloc aussi longtemps que le test qui la suit est vrai. (Le bloc doit rendre le test faux un jour !)
- '**for** [elem] **in** [suite] :' répète le bloc autant de fois qu'il y a d'éléments dans `suite`, et crée une variable `elem` qui prend successivement toutes les valeurs contenues dans `suite`.

Les *n* premiers entiers et nombres pairs

```
n = [...]
lst = [0]*n ; lst2 = [0]*n

i = 0
while i < n:
____ lst[i] = i
____ i = i + 1

for x in lst:
____ lst2[x] = 2*x
```

Notion de fonction II

Une déclaration de fonction permet de regrouper sous un même nom une séquence d'instructions spécifique. En outre, une fonction reçoit des *paramètres* (ou *arguments*) et produit une *valeur de retour*.

On peut voir une fonction comme une boîte noire, dont le fonctionnement interne est toujours le même, et qui reçoit des entrées (les paramètres) et produit une sortie (la valeur de retour) qui dépend de la valeur de ses entrées.

Une fois une fonction déclarée, on peut l'*invoquer* (ou l'*appeler*) par son nom, en précisant les valeurs qu'on donne à ses différents paramètres. Cet appel est automatiquement remplacé par la valeur de retour calculée par la fonction.

Définir des fonctions

Déclaration et appel d'une fonction à deux arguments

```
def n_elems (lst, n):
____ rlst = [""]*n
____ for i in range(n):
________ rlst[i] = lst[i]
____ return rlst

lst = ["bois", "branche", "bûcheron", "buisson",
       "chêne", "clairière", "feuille", "forêt",
       "nid", "pin", "scie"]

lst_b = n_elems (lst, 4)
```

Portée des variables

- Note : Les variables correspondant aux arguments de la fonction n'existent qu'à l'intérieur de cette fonction, et sont prioritaires.

Quelques fonctions utiles

print Affiche son argument à l'écran. (Voir documentation.)

input Affiche une requête à l'écran, et retourne la réponse.

len Retourne la longueur d'une liste, séquence, etc...

range Enumère les entiers jusqu'à son argument. (Pour les indices.)

list Transforme une séquence (mot) en une liste d'éléments.

Remarques

- 'Python' : pas le serpent, les acteurs !
- Toutes les données et instructions se résument à des "0"/"1" et des opérations binaires, mais le langage occulte presque totalement cela. (C'est à ça qu'il sert !)
- Certains langages de programmation laissent au programmeur le soin de gérer la mémoire associée au variables. En Python tout est automatique... mais la mémoire est consommée quand même !
- Les nombres à virgules sont affichés (et calculés !) sur la base de valeurs approchées. La différence peut être significative !
- Les opérateurs logiques n'évaluent que le strict nécessaire. Si le premier test suffit à donner une réponse, les autres sont ignorés.

Références

Liens :

- http ://docs.python.org/release/3.0.1/ (Python en général)
- http ://docs.python.org/release/3.0.1/reference/ (Norme)
- http ://docs.python.org/release/3.0.1/tutorial/ (Tutorial)