Introduction à l'informatique

Cours 4

kn@lri.fr
http://www.lri.fr/~kn

université
PARIS-SACLAY

Résumé des épisodes précédents

On a survolé le langage Python. On sait maintenant :

- Définir des variables : **`toto = 42`**
- Écrire des expressions simples : **`(toto / 4) * 27`**
- Faire des entrées/sorties simples avec **`input('…')`** et **`print('…')`**
- Écrire de code dont l'exécution est conditionné par un test :

```
…
if temperature >= 30:
   print('Il fait chaud')
else:
   print('Il fait bon')
```

Plan

Pourquoi programme-t-on ?

Programmer un ordinateur permet :

- D'effectuer des calculs sans erreur
- D'effectuer des **calculs répétitifs**
- De traiter de façon similaire un grand nombre d'objets

De nombreuses tâches implique une forme de répétition

- « pour chaque étudiant dont la note est >= 10, le marquer ADMIS. »
- « pour chaque usager de la médiathèque, vérifier la date de retour. Si elle est inférieure à la date courante, envoyer un rappel »
- « Soit la liste des entiers entre `2` et `n`. Pour chaque entier `i` de la liste, supprimer tous les multiples stricts de `i` »

L'instruction `while`

En Python, l'instruction **`while`** permet de répeter un **bloc d'instructions** tant qu'une condition est vraie :

```
while e:
    i1
    …
    in
isuite
```

Rappel : en Python, un bloc est un ensemble d'instructions décalés de 4 espaces par rapport au début de la ligne.

Le bloc d'instructions $\mathtt{i_1}$, ..., $\mathtt{i_n}$ est réptété tant que **`e`** s'évalue en **`True`**.

Le bloc d'instructions $\mathtt{i_1}$, ..., $\mathtt{i_n}$ est appelé le **corps** de la boucle.

Exemple

Affichons la table de multiplication par 6 :

```
#fichier table_6.py
i = 0
print ('----------')
while i <= 10:
    print (str(i) + ' * 6 = ' + str(i * 6))
    i = i + 1
print ('----------')
```

```
$ python3 table_6.py
----------
0 * 6 = 0
1 * 6 = 6
2 * 6 = 12
3 * 6 = 18
4 * 6 = 24
5 * 6 = 30
6 * 6 = 36
7 * 6 = 42
8 * 6 = 48
9 * 6 = 54
10 * 6 = 60
----------
```

Attention à la condition

Lorsque l'on écrit « **`while e:`** » :

- La condition **`e`** doit pouvoir varier, sinon la boucle ne termine pas ou n'est jamais exécutée
- Si **`e`** contient **une variable**, cette dernière doit être modifiée correctement dans le corps de la boucle

Exemple :

```
i = 0
print ('----------')
while i <= 10:
    print (str(i) + ' * 6 = ' + str(i * 6))
    i = i  - 1
```

La boucle ci-dessus ne s'arrête jamais (l'utilisateur doit interrompre le programme avec **`CTRL-C`** dans le terminal.

Plan

Boucle `for`

Un motif très courant est la réptition pour un intervalle fixé, par pas constant. On peut utiliser pour cela l'instruction :

```
for var in range(debut, fin, pas):
    i1
    …
    in
isuite
```

La variable **`var`** prend tour à tour les valeurs :

- **`debut`**
- **`debut + pas`**
- **`debut + 2*pas`**
- **`debut + 3*pas`**
- ...

jusqu'à la valeur **`fin`** *exclue*.

Exemple

```
for i in range(7, 22, 3):
    print(str(i))
```

affiche :

```
7
10
13
16
19
```

ici **`19 + 3 = 22`**, la borne supérieure étant exclue, on s'arrête à 19.

On peut écrire **`range(a, b)`** à la place de **`range(a, b, 1)`** et **`range(b)`** pour **`range(0, b, 1)`**.

Le **`pas`** peut être négatif, dans ce cas, **`fin`** doit êtreinférieur à **`debut`**.

Si l'intervalle est vide on ne rentre pas dans la boucle.

Plan

Structure de données

Une **structure de données** est une façon **d'organiser** des valeurs, les **relations** entre ces dernières et les **opérations** permettant de les manipuler.

Une structure de données permet en particulier l'accès efficaces aux données pour certaines utilisations.

Exemples de noms de structures de données :

- tableaux
- listes chaînées
- arbres binaires de recherche
- tas
- piles
- files
- filtres de Bloom
- arbres de Patricia
- ...

Tableau

Un tableau permet de stocker une **collection ordonnée et finie** de valeurs et d'accéder **efficacement à un élément arbitraire** de la collection.

- `[e1, …, en]` : définition d'un tableau
- `t[i]` : accède au $i^{ème}$ élément du tableau `t`. **Attention, les indices commencent à 0**.
- `t[i] = e` : mise à jour du $i^{ème}$ élément du tableau `t`.
- `len(t)` : longueur du tableau `t`

```
>>> tab = [1, 3, 5, 4, 19, 2]
>>> tab
[1, 3, 5, 4, 19, 2]
>>> tab[4]
19
>>> tab[4] = 42
>>> tab
[1, 3, 5, 4, 42, 2]
```

Opérations (un peu) avancées

- `t1 + t2` : concaténation de deux tableaux (renvoie un nouveau tableau avec les éléments de `t1` et `t2` bout à bout).
- `t * n` : concatène `n` fois le tableau `t` avec lui même.

```
>>> t1 = [1, 2, 3]
>>> t2 = [4, 5, 6]
>>> t1 + t2
[1, 2, 3, 4, 5, 6]
>>> t3 = t1 + t2
>>> t3[0] = 10
>>> t3
[10, 2, 3, 4, 5, 6]
>>> t1
[1, 2, 3]
>>> [0] * 10
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
```

Parcours d'un tableau

Il est courant de vouloir parcourir un tableau par indices croissants. On utilise simplement une boucle **`for`**:

```
for i in range(len(tab)):
  # on peut utiliser tab[i] ici
```

On peut aussi parcourir par indices décroissants, en faisant attention :

```
for i in range(len(tab)-1, -1, -1):
  # on peut utiliser tab[i] ici
```

- La dernière case d'un tableau est à l'indice **`len(tab) - 1`**
- L'instruction range exclut la borne de fin, il faut donc aller jusqu'à **`-1`** pour bien avoir l'indice **`0`**

Indices « invalides »

Accéder à un indice supérieur ou égal à la taille est une erreur :

```
>>> tab = [1, 3, 5, 4, 19, 2]
>>> tab[7]
Traceback (most recent call last):
File "", line 1, in <module>
IndexError: list index out of range
>>> tab[-1]
2
>>> tab[-2]
19
>>> tab[-6]
1
>>> tab[-7]
Traceback (most recent call last):
File "", line 1, in <module>
IndexError: list index out of range
```

Les indices négatifs partent de la *fin* du tableau. C'est ~~dégueulasse~~dangereux, non-intuitif, et différent des autres langages. On n'utilisera pas ça, c'est moche.

Exemple : Pyramide des ages

Soit le tableau:

```
pda = [691165, 710534, ..., 2160]  #106 valeurs en tout.
```

Dans la case **i** se trouve le nombre de personnes en France dont l'age est entre **i** et **i+1**. On souhaite écrire un programme qui donne le nombre de personne dont l'age est compris entre deux bornes, demandées à l'utilisateur.

```
entree = input('Age minimal ')
age_min = max(int(entree), 0)    #on ne veut pas d'age négatif
entree = input('Age maximal ')
age_max = min(int(entree), len(pda)-1) #on ne veut pas d'age plus grand
                                       #que ce qui est dans le tableau
total = 0
for i in range(age_min, age_max+1): #+1 car range exclut la borne sup.
    total = total + pda[i]
print(str(total) + ' personnes ont entre ' + str(age_min) + ' et ' +
      str(age_max) + ' ans')
```

Plan

Exceptions

En Python les erreurs **autre que les erreurs de syntaxe** s'appellent des **exceptions**.

Elle servent à signaler une situation « exceptionnelle ». Lorsqu'une telle erreur se produit, on dit que le programme a « levé une exception ».

```
>>> tab = [1, 3, 5, 4, 19, 2]
>>> tab[7]
Traceback (most recent call last):
File "", line 1, in <module>
IndexError: list index out of range
>>> 1/0
Traceback (most recent call last):
File "", line 1, in <module>
ZeroDivisionError: division by zero
>>> int('ABC')
Traceback (most recent call last):
File "", line 1, in <module>
ValueError: invalid literal for int() with base 10: 'ABC'
```

Rattrapage d'exceptions

On veut parfois vouloir gérer une erreur au moment où elle se produit. On peut pour cela utiliser la construction **`try/except`**.

```
try:
    i1
    …
    in
except E:
    j1
    …
    jm
```

Le bloc $\mathbf{i_1}$, ..., $\mathbf{i_n}$ est exécuté. Si une instruction lève l'exception *E*, alors il s'interrompt et le bloc $\mathbf{j_1}$, ..., $\mathbf{j_m}$ est exécuté.

On utilisera cette construction à des endroits bien choisis, sans en abuser (généralement indiqués par l'énoncé de l'exercice). En général pour rattraper une **`ValueError`** levée par la fonction **`int()`**.

Exemple

```
try:
    age = int(input('Entrer votre age : '))
except ValueError:
    age = 20   #Valeur par défaut en cas d'erreur de int()
               #c'est à dire, si l'utilisateur n'a pas saisi
               #un age valide.
```

Plan

But

On souhaite écrire le programme « trouve le nombre » qui :

- Choisi un entier aléatoire `n` entre 0 et 100
- Demande à l'utilisateur de saisir un entier `i`.
- Si `i` et `n` sont égaux, la partie est terminée
- Sinon le programme indique si `i` est trop grand ou trop petit et l'utilisateur peut rejouer.

De plus, on souhaite que le programme re-demande la saisie en cas d'erreur (i.e. si l'utilisateur ne saisi pas un entier correct).

Entier aléatoire


```python
from random import randint

n = randint(0, 100)
# n contient un entier aléatoire entre 0 et 100 inclus
```

Pour le reste, démo (et voir le fichier final devine.py)

Conclusion *

- L'instruction **`while`** permet de faire des boucles sur des conditions complexes
- L'instruction **`for`** permet de faire des boucles sur des ensembles de valeurs finis.
- Les tableaux permettent de stocker des collections ordonnées de valeurs.
- La construction **`try/except`** permet de « bloquer » certaines erreurs et agir en conséquence

* : bien des choses ont été passées sous silence. Elles seront expliquées au fur et à mesure.