I11: Chapitre 3
Structures de données
Instruction `for`

Nicolas Méloni

Licence 1: 1er semestre
(2018/2019)

Données simples/composites

- Les types `int`, `float` et `bool` sont appelés des types simples
- Leurs valeurs, appelées constantes littérales, sont simples
- Il existe également des données composées de plusieurs valeurs : ce sont les données composites.
- Nous verrons principalement 3 structures :
  - les chaînes de caractères
  - les listes
  - les tuples

Les chaînes de caractères

Le type `str`

- Une chaîne de caractères en Python est une séquence de caractères unicode.
- C'est une suite d'éléments ordonnées, indicées par des entiers.
- L'opérateur `[i]` avec i un entier (positif ou négatif) permet d'accéder aux différents caractères de la chaîne.

```
s="Bonjour a tous!"
print(s[0],s[3],s[7],s[8])

>>>
B j   a
```

```
s = "Bonjour a tous!"
print(s[-1],s[-2],s[-4],s[-5])

>>>
! s o t
```

Les chaînes de caractères

- Afficher les caractères d'une chaîne à l'endroit puis à l'envers.

```
s="Rayon X"
i=0
while i<len(s):
    print(s[i])
    i=i+1

>>>
R
a
y
o
n

X
```

```
s = "Rayon X"
i=-1
while -len(ch)<=i:
    print(s[i])
    i=i-1

>>>
X

n
o
y
a
R
```

Extraction de sous-chaîne

- **Syntaxe** : `[i:j]`
- représente les caractères entre les indices i et j (exclus)

```
>>> s="Python 3"
>>> s[1:5]
 "ytho"
>>> s[:3]
 "Pyt"
>>> s[5:]
 "n 3"
```

```
>>> s[-4:-1]
 "on "
>>> s[:-3]
 "Pytho"
>>> s[-4:]
 "on 3"
```

Ensemble d'indices

- **Syntaxe** : `[i:j:k]`
- représente les caractères entre les indices i et j (exclus) avec un pas de k : $i, i+k, i+2k, i+3k, \ldots$

```
>>> s="Programmer"
>>> s[1:6:2]
 "rga"
>>> s[:7:3]
 "Pgm"
>>> s[::4]
 "Pre"
```

```
>>> s[-8:-2:1]
 "ogramm"
>>> s[7:2:-1]
 "mmarg"
>>> s[10:2:-2]
 "rmag"
```

Opérations sur les chaînes de caractères

Les opérateurs

- `+` : concaténation
- `*` : répétition
- `len` : fonction retournant la longueur d'une chaîne

Exemples

```
>>> s1,s2 = "Bonjour", "a tous"
>>> s1[1:4]+s2[::2]
 'onjatu'
>>> s1[-4:-1]*2
 'joujou'
>>> len(s1[2:]+s2[:2])
 6
```

Manipulations classiques

1. Écrire un script qui affiche le nombre d'occurences de la lettre `e` (majuscule ou minuscule) dans une chaîne de caractères saisie au clavier.
2. Écrire un script qui crée une copie d'une chaîne de caractère `ch` en remplacant tous les `i` par des 1 et les `a` par des 4.
3. Écrire un script qui concatène une série de chaînes de caractères en les séparant par des espaces ; la saisie s'arrête quand la chaîne saisie est vide.

Manipulation des chaînes de caractères

- Écrire un script qui affiche le nombre d'occurences de la lettre e (majuscule ou minuscule) dans une chaîne de caractères saisie au clavier.

```
nb_e = 0
i = 0
ch = input()
while i<len(ch):
    if ch[i]=='e' or ch[i]=='E':
        nb_e = nb_e+1
    i = i+1
print(nb_e)
```

Manipulation des chaînes de caractères

- Écrire un script qui crée une copie d'une chaîne de caractère `ch` en remplacant tous les `i` par des 1 et les `a` par des 4.

```
ch = input()
copie = ""
i=0
while i<len(ch):
    if ch[i]=='i':
        copie=copie+"1"
    elif ch[i]=='a':
        copie=copie+"4"
    else:
        copie=copie+ch[i]
```

Manipulation des chaînes de caractères

- Écrire un script qui concatène une série de chaînes de caractères en les séparant par des espaces ; la saisie s'arrête quand la chaîne saisie est vide.

```
mots = ""
continuer = True
while continuer:
    ch = input()
    mots = mots+" "+ch
    continuer = (ch == "")
```

Le type `tuple`

- Un tuple en Python (type `tuple`), est une séquence d'éléments hétérogènes **non modifiable**.
- **Syntaxe** : `(element_1,element_2,...,element_n)`
  - les éléments ne sont pas forcément du même type (hétérogènes)
  - on peut définir un tuple vide par `()`
  - on peut définir un tuple à un seul élément par `(element,)`
  - les éléments sont indicés de 0 à $n-1$
  - les éléménts d'un tuple ne peuvent pas être modifié

Les tuples

Exemples

```
>>> t=(1,'deux',3.0,4,'cinq')
>>> t[0]
 1
>>> t[2]
 3
>>> t[1:4]
 ('deux',3.0,4)
>>> t[0] = 0
 Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: 'tuple' object does not support item assignment
```

Initialisation des tuples

La fonction `range`

- La fonction `range` retourne une séquence d'entiers pouvant servir à l'initialisation de listes ou tuple, ou au parcours d'objet composites indexés par des entiers.
- `range(f)` correspond à la suite $0, 1, \ldots, f-1$
- `range(d, f)` correspond à suite $d, d+1, \ldots, f-1$
- `range(d, f, p)` correspond à la suite $d, d+p, d+2p, d+3p, \ldots$ jusqu'à $f-1$ exclu.

Exemples

```
>>> t = tuple(range(4))
 (0, 1, 2, 3)
>>> t = tuple(range(2,5))
 (2, 3, 4)
>>> t = tuple(range(1,11,2))
 (1, 3, 5, 7, 9)
```

Opérations sur les tuples

Les opérateurs

- `+` : concaténation
- `*` : répétition
- `len` : fonction retournant la longueur d'une chaîne

Exemples

```
>>> t1,t2 = (1,2,3,4), (-1,-2,-3,-4)
>>> t1[:3]+t2[2:]
 (1, 2, 3, -3, -4)
>>> t1[-4:-1]*2
 (1, 2, 3, 1, 2, 3)
>>> len(t1 + t2[::2])
 6
```

Le type `list`

- Une liste en Python (type `list`), est une séquence d'éléments hétérogènes.
- **Syntaxe** : `[element_1,element_2,...,element_n]`
  - les éléments ne sont pas forcément du même type (hétérogènes)
  - on peut définir une liste vide par `[]`
  - les éléments sont indicés de 0 à $n-1$
  - chaque élément d'une liste peut être modifié

```
>>> l=[1,'deux',3.0,4,'cinq']
>>> l[0]
 1
>>> l[2]
 3
>>> l[1:4]
 ['deux',3.0,4]
```

```
>>> l1 = ['a','b']
>>> l2 = ['1','2']
>>> l = [l1,l2]
 [['a','b'],['1','2']]
>>> l[0][0]
 'a'
>>> l[0][1] = 0
>>> l
 [['a',0],['1','2']]
```

Initialisation des listes

La fonction `range`

- La fonction `range` retourne une séquence d'entiers pouvant servir à l'initialisation de listes ou tuple, ou au parcours d'objet composites indexés par des entiers.
- `range(f)` correspond à la suite $0, 1, \ldots, f-1$
- `range(d, f)` correspond à suite $d, d+1, \ldots, f-1$
- `range(d, f, p)` correspond à la suite $d, d+p, d+2p, d+3p, \ldots$ jusqu'à $f-1$ exclu.

Exemples

```
>>> l = list(range(4))
 [0, 1, 2, 3]
>>> l = list(range(-4,-1))
 (-4, -3, -2)
>>> l = list(range(-4,12,3))
 [-4, -1, 2, 5, 8, 11]
```

Opérations sur les listes

Les opérateurs

- `+` : concaténation
- `*` : répétition
- `len` : fonction retournant la longueur d'une chaîne

Exemples

```
>>> l1,l2 = [1,2,3,4], [-1,-2,-3,-4]
>>> l1[:3]+l2[2:]
 [1, 2, 3, -3, -4]
>>> l1[-4:-1]*2
 [1, 2, 3, 1, 2, 3]
>>> len(l1 + l2[::2])
 6
```

Exemples

```
nb = "12345"
L=[nb, 3.14, ["je","tu","il","nous","vous","ils"], (-1,2)]
```

- `L[0] =`
- `L[1]+int(L[0][1]) =`
- `(2,-1)+L[3] =`
- `L[2][1:4] =`
- `L[-2][1::2] =`
- `L[0][:2]*L[-1][-1] =`

Exemples

```
nb = "12345"
L=[nb, 3.14, ["je","tu","il","nous","vous","ils"], (-1,2)]
```

- `L[0] = "12345"`
- `L[1]+int(L[0][1]) =`
- `(2,-1)+L[3] =`
- `L[2][1:4] =`
- `L[-2][1::2] =`
- `L[0][:2]*L[-1][-1] =`

Exemples

```
nb = "12345"
L=[nb, 3.14, ["je","tu","il","nous","vous","ils"], (-1,2)]
```

- `L[0] = "12345"`
- `L[1]+int(L[0][1]) = 5.14`
- `(2,-1)+L[3] =`
- `L[2][1:4] =`
- `L[-2][1::2] =`
- `L[0][:2]*L[-1][-1] =`

Exemples

```
nb = "12345"
L=[nb, 3.14, ["je","tu","il","nous","vous","ils"], (-1,2)]
```

- `L[0] = "12345"`
- `L[1]+int(L[0][1]) = 5.14`
- `(2,-1)+L[3] = (2,-1,-1,2)`
- `L[2][1:4] =`
- `L[-2][1::2] =`
- `L[0][:2]*L[-1][-1] =`

Exemples

```
nb = "12345"
L=[nb, 3.14, ["je","tu","il","nous","vous","ils"], (-1,2)]
```

- `L[0] = "12345"`
- `L[1]+int(L[0][1]) = 5.14`
- `(2,-1)+L[3] = (2,-1,-1,2)`
- `L[2][1:4] = ["tu","il","nous"]`
- `L[-2][1::2] =`
- `L[0][:2]*L[-1][-1] =`

Exemples

```
nb = "12345"
L=[nb, 3.14, ["je","tu","il","nous","vous","ils"], (-1,2)]
```

- L[0] = "12345"
- L[1]+int(L[0][1]) = 5.14
- (2,-1)+L[3] = (2,-1,-1,2)
- L[2][1:4] = ["tu","il","nous"]
- L[-2][1::2] = ["tu,"nous","ils"]
- L[0][:2]*L[-1][-1] =

Exemples

```
nb = "12345"
L=[nb, 3.14, ["je","tu","il","nous","vous","ils"], (-1,2)]
```

- `L[0] = "12345"`
- `L[1]+int(L[0][1]) = 5.14`
- `(2,-1)+L[3] = (2,-1,-1,2)`
- `L[2][1:4] = ["tu","il","nous"]`
- `L[-2][1::2] = ["tu,"nous","ils"]`
- `L[0][:2]*L[-1][-1] = "1212"`

Modification et affectation des listes

- Les listes font parties des types modifiables.
- Une liste peut être modifiée par élément ou par tranche d'éléments.

```
>>> l = [1,2,3,4]
>>> l[0] = 0
>>> l
 [0,2,3,4]
>>> l[1] = [1,2]
>>> l
 [0, [1, 2], 3, 4]
```

```
>>> l[1:3]=[10]
>>> l
 [0,10,4]
>>> l[0:2]=[1,2,3]
>>> l
 [1, 2, 3, 4]
```

Modification et affectation des listes

- L'opérateur spécial += permet d'ajouter un/des élément(s) en fin de liste.
- **Syntaxe** : `liste += [element_1,...,element_N]`

```
>>> l = [1,2,3,4]
>>> l += [5]
>>> l
 [1,2,3,4,5]
```

```
>>> l += [6,7]
>>> l
 [1,2,3,4,5,6,7]
```

Modification et affectation des listes

- Contrairement aux types précédents, les listes ne sont pas des éléments mais représentent des zones mémoires.
- S'il existe plusieurs références à une même zone mémoire, alors modifier une partie de la zone mémoire à partir d'une référence modifie également les autres références !
- Ça n'est pas le cas pour tous les autres types.

```
>>> a = 2
>>> b = a
>>> a = 3
>>> print(a,b)
 3 2
```

```
>>> l1 = [1,2]
>>> l2 = l1
>>> l1[0] = 0
>>> print(l1,l2)
 [0,2] [0,2]
```

```
>>> l1 = [1,2]
>>> l2 = [1,2]
>>> l1[0] = 0
>>> print(l1,l2)
 [0,2] [1,2]
```

Modification et affectation des listes

- Attention notamment aux listes de listes !

```
>>> l1 = [1,2]
>>> l2 = [1,2]
>>> l3 = [l1,l2]
>>> print(l3)
 [[1,2],[1,2]]
>>> l3[0][0] = 0
>>> print(l3)
 [[0,2],[1,2]]
```

```
>>> l1 = [1,2]
>>> l2 = l1
>>> l3 = [l1,l2]
>>> print(l3)
 [[1,2],[1,2]]
>>> l3[0][0] = 0
>>> print(l3)
 [[0,2],[0,2]]
```

Modification et affectation des listes

Liste de tuples et tuple de listes

```
>>> l1 = [1,2]
>>> l2 = [3,4]
>>> t = (l1,l2)
>>> t[0][0] = 1
```

```
>>> t1 = (1,2)
>>> t2 = (3,4)
>>> l = [t1,t2]
>>> l[0][0] = 1
```

Modification et affectation des listes

Liste de tuples et tuple de listes

```
>>> l1 = [1,2]
>>> l2 = [3,4]
>>> t = (l1,l2)
>>> t[0][0] = 1
```

- `t = ([0,2],[3,4])`

```
>>> t1 = (1,2)
>>> t2 = (3,4)
>>> l = [t1,t2]
>>> l[0][0] = 1
```

Modification et affectation des listes

Liste de tuples et tuple de listes

```
>>> l1 = [1,2]
>>> l2 = [3,4]
>>> t = (l1,l2)
>>> t[0][0] = 1
```

- `t = ([0,2],[3,4])`

```
>>> t1 = (1,2)
>>> t2 = (3,4)
>>> l = [t1,t2]
>>> l[0][0] = 1
```

- `TypeError: 'tuple' object does not support item assignment`

Copie de liste

- Pour copier une liste il faut donc la copier élément par élément :

```
# version 1

l = [1,2,3,4,5,6]
l_copie = []
i = 0
while i < len(l):
    l_copie += [l[i]]
    i=i+1
```

```
# version 2

l = [1,2,3,4,5,6]
l_copie = [0]*len(l)
i = 0
while i < len(l):
    l_copie[i] = l[i]
    i=i+1
```

Copie de liste

- Attention à nouveau aux listes de listes !

```
# Copie de surface d'une liste
l = [[1,2],[3,4]]
l_copie = []
i = 0
while i < len(l):
    l_copie += [l[i]]
    i=i+1

l[0][0]=0
print(l, l_copie)

>>>
 [[0,2],[3,4]] [[0,2],[3,4]]
```

Copie de liste

- Attention à nouveau aux listes de listes !

```
# Copie profonde d'une liste
l = [[1,2],[3,4]]
l_copie = []
i = 0
while i < len(l):
    l_temp = []
    j=0
    while j < len(l[i]):
        l_temp += [l[i][j]]
        j=j+1
    l_copie += [l_temp]
    i=i+1

l[0][0]=0
print(l, l_copie)

>>>
 [[0,2],[3,4]] [[1,2],[3,4]]
```

Manipulations classiques

1. Saisir les éléments d'une liste au clavier, la saisie se termine quand la chaîne vide est saisie (elle ne doit pas être prise en compte).
2. Calculer la moyenne des éléments d'une liste `l_float` de flottants prédéfinie.
3. Stocker dans une liste les indices où apparait le caractère *saut de ligne* d'une chaine de caractère `ch` prédéfinie.

Manipulations des listes

- Saisir les éléments d'une liste au clavier, la saisie se termine quand la chaîne vide est saisie.

```
l = []
saisie = input()
while saisir != "":
    l += [saisie]
    saisie = input()
```

Manipulations des listes

- Calculer la moyenne des éléments d'une liste `l_float` de flottants prédéfinie.

```
moyenne = 0
i = 0
while i < len(l_float):
    moyenne = moyenne + l_float[i]
    i = i+1
moyenne = moyenne/len(l_float)
```

Manipulations des listes

- Stocker dans une liste les indices où apparait le caractère *saut de ligne* d'une chaine de caractère `ch` prédéfinie.

```
l_indice = []
i = 0
while i < len(ch):
    if ch[i] == '\n':
        l_indice += [i]
```

Manipulations de deux listes

1. Stocker dans une liste les indices des cases contenant des éléments identiques de deux listes `l1` et `l2` de même longueur.
2. Afficher toutes les combinaisons possibles de paires d'élements appartenant à deux listes `l1` et `l2`.

Manipulations de plusieurs listes

- Stocker dans une liste les indices des cases contenant des éléments identiques de deux listes `l1` et `l2` prédéfinies de même longueur.

```
l_indice = []
i = 0
while i < len(l1):
    if l1[i] == l2[i]:
        l_indice += [i]
    i = i+1
```

Manipulations de plusieurs listes

- Afficher toutes les combinaisons possibles de paires d'élements appartenant à deux listes l1 et l2 prédéfinies.

```
i = 0
while i < len(l1):
    j = 0
    while j < len(l2):
        print(l1[i],l2[i])
        j = j+1
    i = i+1
```

L'instruction `for`

- **Syntaxe :** `for <element> in <sequence>`
  - `<element>` est un nom de variable
  - `<sequence>` est un **itérable**
  - Lors de l'exécution de la boucle, la variable `<element>` prendra tour à tour toutes les valeurs des éléments composant la structure `<sequence>`.
- Dans ce cours, on considérera qu'un itérable est simplement une donnée de type `list`, `str`, `tuple` ou le résultat de la fonction `range`.

L'instruction for

Parcours de chaîne de caractères

```
chaine = "Bonjour"
for car in chaine:
    print(car)

>>>
B
o
n
j
o
u
r
```

L'instruction `for`

Parcours de chaîne de caractères

```
t = ("toto", 1, 2.4)
for el in t:
    print(el)

>>>
toto
1
2.4
```

L'instruction `for`

Parcours de chaîne de caractères

```
l = [(1,2), 'rien', 3.14]
for case in l:
    print(case)

>>>
(1, 2)
rien
3.14
```

L'instruction `for`

Parcours d'un intervalle d'entiers

```
for i in range(5):
    print(i)

>>>
0
1
2
3
4
```

```
for nombre in range(-7,8,3):
    print(nombre)

>>>
-7
-4
-1
2
5
```

Comparaison `for` / `while`

- L'instruction `for` ne permet que de parcourir des ensembles finis dont la taille est connue avant l'exécution de la boucle.
- L'instruction `while` permet de créer une répétition d'instructions selon une condition quelconque.

for

```
somme = 0
n = int(input("entrer un nombre: "))
for i in range(n):
    somme = somme+i
print(somme)

>>>
entrer un nombre: 100
4950
```

Comparaison for / while

- L'instruction `for` ne permet que de parcourir des ensembles finis dont la taille est connue avant l'exécution de la boucle.
- L'instruction `while` permet de créer une répétition d'instructions selon une condition quelconque.

while

```
somme = 0
while somme < 100:
    n = int(input("entrer un nombre: "))
    somme = somme+n
print(somme)
>>>
entrer un nombre: 50
entrer un nombre: 25
entrer un nombre: 65
140
```

Parcours de liste avancé

- L'instruction `for` peut être utilisée avec plusieurs variables lorsque la structure à parcourir s'y prête.

```
# parcours classique
l=[(0.1,0), (1,2),
   (3.14,6.12)]
for t in l:
   x = t[0]
   y = t[1]
   print(x,y)
>>>
0.1 0
1 2
3.14 6.12
```

```
# parcours avance


l=[(0.1,0), (1,2),
   (3.14,6.12)]
for x,y in l:
   print(x,y)
>>>
0.1 0
1 2
3.14 6.12
```

Parcours de liste avancé

- On considère une liste prédéfinies `l_points` contenant des tuples de 2 flottants représentant des points du plan. Écrire un script qui demande à l'utilisateur de saisir trois flottant `x_p,y_p` et `r` et affiche tous les points de la liste se trouvant à l'intérieur du cercle de centre `(x_p,y_p)` et de rayon `r`.

Parcours de liste avancé

```
# la liste l_points est predefinies

x_p = float(input())
y_p = float(input())
r = float(input())

for x,y in l_points:
    X = x_p-x
    Y = y_p-y
    dist = (X**2+Y**2)**(0.5)
    if dist <= r:
        print(x,y)
```