Cours d'algorithmique et programmation 2

Cours 2

Anne Parrain

UFR des Sciences
Licence Sciences et Technologie
mentions Mathématiques et Informatique
Semestre 2

Année universitaire 2017–2018

Section 1

Types mutables et types non mutables

Les listes

Une liste est une structure de données qui permet de mémoriser une collection d'objets, quelque soit leur type.

```
>>> langages = ["python","scheme","maple","pascal","c",
>>> langages[0]
'python'
>>> langages[5]
'java'
>>> len(langages)
7
>>> langages[0] = "Python"  ## On peut modifier une lis
>>> langages
['Python', 'scheme', 'maple', 'pascal', 'c', 'java', 'p
>>> type(langages)
<type 'list'>
```

Pour ajouter un élément

On peut utiliser la **méthode** append qui permet d'ajouter un élément à une liste.

```
>>> langages.append("assembleur")  # ajoute l'element ass
>>> langages
['Python', 'scheme', 'maple', 'pascal', 'c', 'java', 'php
>>> len(langages)
8
```

À vous !

Que se passe-t-il dans chacun de ces cas ? Est-ce que le contenu de la variable a changé ?

1.
```
1 a = 3
2 a = a + 2
```

2.
```
1 l = [1, 2, 3]
2 l.append(4)
```

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

4.
```
1 s = "Bonjour"
2 s = s + " tout le monde !"
```

Réponses (1)

La fonction `id` retourne la référence contenue dans une variable.

1.
```
>>> a = 3
>>> id(a)
10455104
>>> a = a + 2
>>> id(a)
10455168
```

2.
```
>>> l = [1, 2, 3]
>>> id(l)
140589985385736
>>> l.append(4)
>>> id(l)
140589985385736
```

Réponses (2)

1.
```
>>> l = [1, 2, 3]
>>> id(l)
140589984578952
>>> l = l + [4, 5]
>>> id(l)
140589984578632
```

2.
```
>>> s = "Bonjour"
>>> id(s)
140589984568968
>>> s = s+" tout le monde !"
>>> id(s)
140589984550536
```

Conclusions (1)

On dira d'un type qu'il est mutable si ces éléments sont modifiables.

Dans les types de données qu'on connaît :

- **types non mutables** : les valeurs numériques (`int`, `float`), les booléens (`boolean`), les chaînes de caractères (`str`), les tuples (`tuple`)
- **types mutables** : les listes (`list`), les dictionnaires (`dict`), les ensembles (`set`)

Remarque :

- On ne peut mettre une valeur par défaut pour un paramètre que pour un type non mutable.

À vous ! (1)

```
1  def multiplie_et_soustrait(m,n) :
2      '''int, int -> int
3      retourne le multiple de n et soustrait n a m.
4      '''
5      res = m * n
6      m = m - n
7      return res
8
9  if __name__ == '__main__' :
10     mon_entier = 10
11     mon_resultat = multiplie_et_soustrait(mon_entier,8)
12     print(mon_resultat)
13     print(mon_entier)
```

Conclusion ? Correction ?

Mais encore . . . - À vous ! (2)

```
1  def retire_multiples(l, n) :
2      '''list(int), int -> list(int)
3      retourne la liste de tous les multiples de n
4      qui sont presents dans l.
5      '''
6      if n == 0 :
7          return []
8      ind = 0
9      res = []
10     while ind < len(l) :
11         if l[ind] % n == 0 :
12             res.append(l[ind])
13             del(l[ind])
14         else :
15             ind += 1
16     return res
17 if __name__ == '__main__' :
18     l = [10, 7, 6, 8, 4, 12, 5, 13]
19     l_pairs = retire_multiples(l,2)
20     print(l_pairs,' -',l)
```

Conclusion

- Une fonction ne peut pas modifier ses paramètres d'un type non mutable
- Une fonction peut modifier ses paramètres d'un type mutable

Règles à observer :

- Si une fonction modifie un de ses paramètres mutables, alors elle l'indiquera par la notation `inout` dans sa spécification
- Pour lever toute ambigüité, une fonction qui ne modifiera pas ses paramètres ne les utilisera pas à gauche d'une affectation.

Corrections

```
1  def multiplie_et_soustrait(m,n) :
2      '''int, int -> (int,int)
3      retourne le multiple de n
4      et le resultat de la soustraction de n a m.
5      '''
6      res1 = m * n
7      res2 = m - n
8      return res1,res2
9
10 def retire_multiples(l, n) :
11     '''list(int) (inout), int -> list(int)
12     retourne la liste de tous les multiples de n
13     qui sont presents dans l.
14     '''
```

D’autres corrections - À vous !

```
1  def est_vivant(guerrier):
2      '''TGuerrier -> boolean
3      teste si un guerrier est encore vivant
4      '''
5
6  def donne_un_coup(guerrier):
7      '''TGuerrier -> int
8      un joueur 1 donne un coup
9      le coup depend de la force et de l'adresse
10     '''
11
12 def prend_un_coup(guerrier, nbPts):
13     '''TGuerrier, int -> TGuerrier
14     perte de points de vie
15     '''
16
17 def attaque(guerrier1,guerrier2):
18     '''TGuerrier, TGuerrier -> (TGuerrier, TGuerrier)
19     effectue l'attaque de guerrier1 sur guerrier2
20     et la riposte de guerrier2 sur guerrier1.
       '''
```

Section 2

Accéder à une sous-liste

L'opérateur : (1)

Il s'utilise par `liste[debut:fin]` où `debut` et `fin` sont les indices de la tranche souhaitée, avec `liste[debut]` inclus et `liste[fin]` exclu.
Cet opérateur est aussi utilisé pour les chaînes de caractères (qui sont des listes un peu particulières).

```
>>> liste = [1,5,567,'aze','gfd',456]
>>> liste[2:4]
[567, 'aze']
>>> liste[2:5]
[567, 'aze', 'gfd']
>>> mot = "badaboumbambam"
>>> mot[4:8]
'boum'
```

L'opérateur : (2)

Si `debut` est omis, alors par défaut la tranche commencera à l'indice `0` (indice du premier élément de la liste), si `fin` est omis alors la tranche s'arrêt au dernier élément de la liste.

```
>>> def estPrefixe(mot1,mot2):
        return len(mot1) <= len(mot2) and mot1 == mot2[:l

>>> estPrefixe("broc","broche")
True
>>> estPrefixe("broc","brioche")
False
>>> def estSuffixe(mot1,mot2):
        return len(mot1) <= len(mot2) and mot1 == mot2[le

>>> estSuffixe("roche","broche")
True
>>> estSuffixe("poche","broche")
False
>>> estSuffixe("abroche","broche")
False
```

L'opérateur : (3)

C'est également pratique pour copier une tranche d'une liste dans une autre liste.

```
>>> mot = "badaboumbambam"
>>> boum = mot[4:8]
>>> boum
'boum'
>>> boum = boum+"...aaaahh"
>>> boum
'boum...aaaahh'
>>> mot
'badaboumbambam'
>>> liste
[1, 5, 567, 'aze', 'gfd', 456]
>>> listeMots = liste[3:5]
>>> listeMots
['aze', 'gfd']
>>> listeMots[1] = 'rty'
>>> listeMots
['aze', 'rty']
>>> liste
```

L'opérateur : (4)

En particulier, `liste[:]` désigne donc la liste complète. Remarquez la différence entre les deux affectations suivantes :

```
>>> liste
[1, 5, 567, 'aze', 'gfd', 456]
>>> l1 = liste
>>> l2 = liste[:]
>>> l1
[1, 5, 567, 'aze', 'gfd', 456]
>>> l2
[1, 5, 567, 'aze', 'gfd', 456]
>>> l1[1] = -128
>>> l1
[1, -128, 567, 'aze', 'gfd', 456]
>>> liste
[1, -128, 567, 'aze', 'gfd', 456]
>>> l2
[1, 5, 567, 'aze', 'gfd', 456]
>>> l2[0] = mot
```

Section 3

Tests unitaires

Tester une fonction

- Lorsqu'on écrit une fonction, on s'attend à un certain fonctionnement
- Le premier travail, la fonction écrite, est de vérifier ce fonctionnement => vous devez tester les fonctions que vous avez écrites
  - sur quelques cas standards d'appel
  - sur des cas particuliers
- tant que la fonction ne répond pas correctement, il faut la corriger.

Ce travail est fastidieux => les `doctests` sont là pour vous aider !

Tester une fonction

- Les tests unitaires montrent le fonctionnement attendu de la fonction.
- On peut les inclure directement dans la spécification de la fonction !
- Ces tests ont alors l'avantage d'être exécutables, et pourront nous permettre de "vérifier" automatiquement que la fonction écrite correspond bien à la fonction attendue.

Écrire les tests

```
1  def f(x):
2      '''int -> int
3         une fonction qui retourne 2*x+1
4
5         on simule une conversation avec l'interprete
6         python. Ne pas oublier l'espace entre les >>>
7         et l'appel de la fonction.
8         Ne pas oublier la ligne vide avant les tests.
9
10        >>> f(0)
11        1
12        >>> f(1)
13        3
14        >>> f(-0.5)
15        0.0
16        >>> f(100)
17        201
18     '''
19     return 2*x+1
```

Lancer les tests

Pour vérifier si la fonction créée correspond aux tests, vous avez deux solutions :

1. il suffit de lancer dans une console la commande suivante, si la fonction se trouve dans un fichier `intro.py` :
   `python3 -m doctest intro.py -v`
2. en mode interactif, il suffit de taper les trois commandes suivantes pour obtenir le même résultat :

```
>>> import doctest
>>> import intro
>>> doctest.testmod(intro)
```

3. Vous pouvez aussi ajouter les lignes ci-dessus comme action principale de votre module (s'il n'en a pas) :

```
1 if __name__ == '__main__' :
2     import doctest
3     doctest.testmod()
```

Résultat des tests

Vous obtenez alors l'affichage suivant :

```
Trying:
    f(0)
Expecting:
    1
ok
Trying:
    f(1)
Expecting:
    3
ok
Trying:
    f(-0.5)
Expecting:
    0.0
ok
```

Résultat des tests

```
Trying:
    f(100)
Expecting:
    201
ok
1 items had no tests:
    intro
1 items passed all tests:
   4 tests in intro.f
4 tests in 2 items.
4 passed and 0 failed.
Test passed.
```