

Formation au langage de programmation Python

Partie III construction en compréhension – λ -function

Formateur : IBRAHIM M. S.



Global Knowledge®

du 30/05 au 02/06 2017

Chapitre : construction en compréhension – λ -function

- 1 Conteneurs : approfondissements
 - Itérateurs
 - Fonctions anonymes
 - Construction en compréhension
- 2 Structures classiques de contrôle
- 3 Résumé — questions

Itérateurs

structure que Python traite naturellement élément par élément :

Itérateurs

structure que Python traite naturellement élément par élément :

- liste, tuple, set

Itérateurs

structure que Python traite naturellement élément par élément :

- liste, tuple, set
- string, fichier

Itérateurs

structure que Python traite naturellement élément par élément :

- liste, tuple, set
- string, fichier
- retour de certaines fonctions :

Itérateurs

structure que Python traite naturellement élément par élément :

- liste, tuple, set
- string, fichier
- retour de certaines fonctions :
 - ▶ `range`

Itérateurs

structure que Python traite naturellement élément par élément :

- liste, tuple, set
- string, fichier
- retour de certaines fonctions :
 - ▶ range
 - ▶ iterfind

Itérateurs

structure que Python traite naturellement élément par élément :

- liste, tuple, set
- string, fichier
- retour de certaines fonctions :
 - ▶ range
 - ▶ iterfind
 - ▶ ...

Itérateurs

structure que Python traite naturellement élément par élément :

- liste, tuple, set
- string, fichier
- retour de certaines fonctions :
 - ▶ range
 - ▶ iterfind
 - ▶ ...

Utilité

- traitement au niveau des composantes et non du bloc

Itérateurs

structure que Python traite naturellement élément par élément :

- liste, tuple, set
- string, fichier
- retour de certaines fonctions :
 - ▶ range
 - ▶ iterfind
 - ▶ ...

Utilité

- traitement au niveau des composantes et non du bloc
- plus simple à écrire/relire (pas de gestion des indices)

Itérateurs

structure que Python traite naturellement élément par élément :

- liste, tuple, set
- string, fichier
- retour de certaines fonctions :
 - ▶ range
 - ▶ iterfind
 - ▶ ...

Utilité

- traitement au niveau des composantes et non du bloc
- plus simple à écrire/relire (pas de gestion des indices)
- on ne charge pas tous les éléments afin de lancer le traitement

Itérateurs

structure que Python traite naturellement élément par élément :

- liste, tuple, set
- string, fichier
- retour de certaines fonctions :
 - ▶ range
 - ▶ iterfind
 - ▶ ...

Utilité

- traitement au niveau des composantes et non du bloc
- plus simple à écrire/relire (pas de gestion des indices)
- on ne charge pas tous les éléments afin de lancer le traitement
- **traitement plus rapide et plus économe en ressources/mémoire**

Fonction anonyme

- description d'un mécanisme d'association

Fonctions à la volée : les fonctions anonymes

Fonction anonyme

- description d'un mécanisme d'association
- ce mécanisme doit être décrit par une unique instruction

Fonctions à la volée : les fonctions anonymes

Fonction anonyme

- description d'un mécanisme d'association
- ce mécanisme doit être décrit par une unique instruction

Syntaxe

- `lambda sequence_of_arguments : expression`

Fonctions à la volée : les fonctions anonymes

Fonction anonyme

- description d'un mécanisme d'association
- ce mécanisme doit être décrit par une unique instruction

Syntaxe

- `lambda` `sequence_of_arguments` : `expression`

Utilisation

- `(lambda arguments_fictifs : expr) (argument_reels)`

Fonctions à la volée : les fonctions anonymes

Fonction anonyme

- description d'un mécanisme d'association
- ce mécanisme doit être décrit par une unique instruction

Syntaxe

- `lambda` sequence_of_arguments : expression

Utilisation

- `(lambda arguments_fictifs : expr) (argument_reels)`
- `f = lambda arguments_fictifs : expr; f (argument_reels)`

Fonctions à la volée : les fonctions anonymes

Fonction anonyme

- description d'un mécanisme d'association
- ce mécanisme doit être décrit par une unique instruction

Syntaxe

- `lambda` sequence_of_arguments : expression

Utilisation

- `(lambda arguments_fictifs : expr) (argument_reels)`
- `f = lambda arguments_fictifs : expr; f (argument_reels)`

Exemple

```
>>> lambda L :dict((x, [k for k in range(len(L)) if L[k]==x]) for x in set(L))  
<function <lambda> at 0x10f48fa60>
```

Les deux portions de codes suivantes sont équivalentes

```
>>> L = [1,2,1,3,1,4,5,6,0,2,3,2,9]

>>> d = {}
>>> for elem in set(L) :
...     if not elem in d.keys() :
...         d[elem] = []
...     for k in range(len(L)) :
...         if elem == L[k] :
...             d[elem].append(k)
...
>>> lambda L : dict((x, [k for k in range(len(L)) if L[k]==x]) for x in set(L))
<function <lambda> at 0x10f48ff28>

>>> d == (lambda L : dict((x, [k for k in range(len(L)) if L[k]==x]) for x in set(L))) (L)
True
```

Les deux portions de codes suivantes sont équivalentes

```
>>> L = [1,2,1,3,1,4,5,6,0,2,3,2,9]

>>> d = {}
>>> for elem in set(L) :
...     if not elem in d.keys() :
...         d[elem] = []
...     for k in range(len(L)) :
...         if elem == L[k] :
...             d[elem].append(k)
...
>>> lambda L : dict((x, [k for k in range(len(L)) if L[k]==x]) for x in set(L))
<function <lambda> at 0x10f4962f0>

>>> d == (lambda L : dict((x, [k for k in range(len(L)) if L[k]==x]) for x in set(L))) (L)
True
```

Intérêts

Les deux portions de codes suivantes sont équivalentes

```
>>> L = [1,2,1,3,1,4,5,6,0,2,3,2,9]

>>> d = {}
>>> for elem in set(L) :
...     if not elem in d.keys() :
...         d[elem] = []
...     for k in range(len(L)) :
...         if elem == L[k] :
...             d[elem].append(k)
...
>>> lambda L : dict((x, [k for k in range(len(L)) if L[k]==x]) for x in set(L))
<function <lambda> at 0x10f4966a8>

>>> d == (lambda L : dict((x, [k for k in range(len(L)) if L[k]==x]) for x in set(L))) (L)
True
```

Intérêts

- concision du code et meilleure lisibilité

Les deux portions de codes suivantes sont équivalentes

```
>>> L = [1,2,1,3,1,4,5,6,0,2,3,2,9]

>>> d = {}
>>> for elem in set(L) :
...     if not elem in d.keys() :
...         d[elem] = []
...     for k in range(len(L)) :
...         if elem == L[k] :
...             d[elem].append(k)
...
>>> lambda L : dict((x, [k for k in range(len(L)) if L[k]==x]) for x in set(L))
<function <lambda> at 0x10f481620>

>>> d == (lambda L : dict((x, [k for k in range(len(L)) if L[k]==x]) for x in set(L))) (L)
True
```

Intérêts

- concision du code et meilleure lisibilité
- ne pas définir une fonction que l'on ne réutilisera pas

Conteneurs en compréhension (ou en intention)

Syntaxe

```
new = cast_function(transfo(i) for i in old [if cond(i)])
```


Syntaxe

```
new = cast_function(transfo(i) for i in old [if cond(i)])
```

Cas d'utilisation : composition possible

- filtrage d'une liste d'éléments sur une condition

Syntaxe

```
new = cast_function(transfo(i) for i in old [if cond(i)])
```

Cas d'utilisation : composition possible

- filtrage d'une liste d'éléments sur une condition
- appliquer une même transformation aux éléments d'un conteneur

Conteneurs en compréhension (ou en intention)

Syntaxe

```
new = cast_function(transfo(i) for i in old [if cond(i)])
```

Cas d'utilisation : composition possible

- filtrage d'une liste d'éléments sur une condition
- appliquer une même transformation aux éléments d'un conteneur

```
>>> a,b,c = [1,4,7,6],[0,2,4],{"1","5","10","15","20","25"}
>>> [x for x in a if x > 5]
[7, 6]
>>> [2**x for x in b]
[1, 4, 16]
>>> [str(int(x)*10) for x in c]
['100', '50', '150', '10', '250', '200']
```

now exiting Console...

Tests booléens

Tests booléens

- il faut pour cela disposer de condition booléennes

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if cond :`
 \rightarrow `bloc`

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if cond :`
 → bloc
- `elif cond :`
 → bloc

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if` cond :
 → bloc
- `elif` cond :
 → bloc
- `else` :
 → bloc

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if` `cond` :
 → bloc
- `elif` `cond` :
 → bloc
- `else` :
 → bloc

Tant que :

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if` `cond` :
 → `bloc`
- `elif` `cond` :
 → `bloc`
- `else` :
 → `bloc`

Tant que :

- `while` `cond` :
 → `bloc`

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if` `cond` :
 → bloc
- `elif` `cond` :
 → bloc
- `else` :
 → bloc

Tant que :

- `while` `cond` :
 → bloc

Itération

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if` `cond` :
 → `bloc`
- `elif` `cond` :
 → `bloc`
- `else` :
 → `bloc`

Tant que :

- `while` `cond` :
 → `bloc`

Itération

- `for` `i in C` :
 → `bloc`

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if` `cond` :
 → bloc
- `elif` `cond` :
 → bloc
- `else` :
 → bloc

Tant que :

- `while` `cond` :
 → bloc

Itération

- `for` `i in C` :
 → bloc

Boucle

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if` `cond` :
 → bloc
- `elif` `cond` :
 → bloc
- `else` :
 → bloc

Tant que :

- `while` `cond` :
 → bloc

Itération

- `for` `i in C` :
 → bloc

Boucle

- pas de boucle classique

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if` `cond` :
 → bloc
- `elif` `cond` :
 → bloc
- `else` :
 → bloc

Tant que :

- `while` `cond` :
 → bloc

Itération

- `for` `i in C` :
 → bloc

Boucle

- pas de boucle classique
- `for from 1 to n`

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if` `cond` :
 → bloc
- `elif` `cond` :
 → bloc
- `else` :
 → bloc

Tant que :

- `while` `cond` :
 → bloc

Itération

- `for` `i in C` :
 → bloc

Boucle

- pas de boucle classique
- `for` from 1 to n
- on simule cela

Structures classiques de contrôle




Tests booléens

- il faut pour cela disposer de condition booléennes


Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`


Branchements :

- `if` `cond` :
 `bloc`
- `elif` `cond` :
 `bloc`
- `else` :
 `bloc`

Tant que :

- `while` `cond` :
 `bloc`

Itération

- `for` `i in C` :
 `bloc`

Boucle

- pas de boucle classique
- `for` from 1 to n
- on simule cela
- `for` `i in range(n)` :

Structures classiques de contrôle

Tests booléens

- il faut pour cela disposer de condition booléennes

Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`

Branchements :

- `if` `cond` :
 → bloc
- `elif` `cond` :
 → bloc
- `else` :
 → bloc

Tant que :

- `while` `cond` :
 → bloc

Itération

- `for` `i in C` :
 → bloc

Boucle

- pas de boucle classique
- `for` from 1 to n
- on simule cela
- `for` `i in range(n)` :
- avec un itérateur

Structures classiques de contrôle




Tests booléens

- il faut pour cela disposer de condition booléennes


Conditions logiques — quelques nouveautés :

- `A in B` — `X is Y` — `X is not Y` — `A|B` — `A & B` — `A < B`


Branchements :

- `if` `cond` :
 `bloc`
- `elif` `cond` :
 `bloc`
- `else` :
 `bloc`

Tant que :

- `while` `cond` :
 `bloc`

Itération

- `for` `i in C` :
 `bloc`

Boucle

- pas de boucle classique
- `for` from 1 to n
- on simule cela
- `for` `i in range(n)` :
- avec un itérateur
- ou une boucle `while`

Résumé de la séquence

Résumé de la séquence

- itérateurs

Résumé de la séquence

- itérateurs
- construction en compréhension

Résumé de la séquence

- itérateurs
- construction en compréhension
- λ -function (fonction anonyme)

Résumé de la séquence

- itérateurs
- construction en compréhension
- λ -function (fonction anonyme)
- opérateurs logiques

Résumé de la séquence

- itérateurs
- construction en compréhension
- λ -function (fonction anonyme)
- opérateurs logiques
- opérations ensemblistes

Résumé de la séquence

- itérateurs
- construction en compréhension
- λ -function (fonction anonyme)
- opérateurs logiques
- opérations ensemblistes
- filtrage d'un conteneur

Résumé de la séquence

- itérateurs
- construction en compréhension
- λ -function (fonction anonyme)
- opérateurs logiques
- opérations ensemblistes
- filtrage d'un conteneur

Questions ?