Initiation au développement

9 — Fonctions anonymes, `map`, `filter`

Anthony Labarre

Université Gustave Eiffel

Plan d'aujourd'hui

1. Les fonctions anonymes (**`lambda`**)
2. La fonction `map`
3. La fonction `filter`

Python pas à pas

Les fonctions anonymes (`lambda`)

Fonctions en argument

- Rappel : on peut passer des fonctions en argument d'autres fonctions (cf. cours précédents) ;
- Cela permet de réutiliser du code très facilement ;
- Parfois, on n'a besoin de la définition d'une fonction très simple qu'à un seul endroit ;
- Dans ces situations, il est commode d'utiliser des fonctions dites **anonymes**, définies au moment où elles sont nécessaires ;

Fonctions anonymes : le mot-clé `lambda`

Syntaxe des `lambda`

On déclare à la volée une **fonction anonyme** à l'aide de

```
lambda x: expression(x)
```

qui équivaut à

```
def f(x):
    return expression(x)
```

Fonctions anonymes : le mot-clé `lambda`

Syntaxe des `lambda`

On déclare à la volée une **fonction anonyme** à l'aide de

```
lambda x: expression(x)
```

qui équivaut à

```
def f(x):
    return expression(x)
```

Attention :

1. on ne peut pas appeler les fonctions anonymes (...) ;

Fonctions anonymes : le mot-clé `lambda`

Syntaxe des `lambda`

On déclare à la volée une **fonction anonyme** à l'aide de

```
lambda x: expression(x)
```

qui équivaut à

```
def f(x):
    return expression(x)
```

Attention :

1. on ne peut pas appeler les fonctions anonymes (...) ;
2. on ne peut donc les déclarer qu'à des endroits bien précis ;

Fonctions anonymes : le mot-clé `lambda`

Syntaxe des `lambda`

On déclare à la volée une **fonction anonyme** à l'aide de

```
lambda x: expression(x)
```

qui équivaut à

```
def f(x):
    return expression(x)
```

Attention :

1. on ne peut pas appeler les fonctions anonymes (...) ;
2. on ne peut donc les déclarer qu'à des endroits bien précis ;
3. on n'écrit pas de `return`.

`lambda` : quelques exemples simples

Les fonctions anonymes suivantes pourront nous servir :

- `lambda x: x % 2` (déterminer si `x` est impair) ;

`lambda` : quelques exemples simples

Les fonctions anonymes suivantes pourront nous servir :

- `lambda x: x % 2` (déterminer si `x` est impair) ;
- `lambda couple: couple[0] != couple[1]`
  (déterminer si `couple` est formé d'éléments différents) ;

`lambda` : quelques exemples simples

Les fonctions anonymes suivantes pourront nous servir :

- `lambda x: x % 2` (déterminer si `x` est impair) ;
- `lambda couple: couple[0] != couple[1]`
  (déterminer si `couple` est formé d'éléments différents) ;
- `lambda vecteur: sum(vecteur)`
  (utiliser la somme des composantes de `vecteur`) ;

`lambda` : quelques exemples simples

Les fonctions anonymes suivantes pourront nous servir :

- `lambda x: x % 2` (déterminer si `x` est impair) ;
- `lambda couple: couple[0] != couple[1]`
  (déterminer si `couple` est formé d'éléments différents) ;
- `lambda vecteur: sum(vecteur)`
  (utiliser la somme des composantes de `vecteur`) ;

Le nombre de paramètres sera imposé par l'itérable auquel on applique la fonction.

Application à `sorted`

- Rappel : `sorted` possède un paramètre `key` qui permet de définir la manière dont sont faites les comparaisons ;

Application à `sorted`

- Rappel : `sorted` possède un paramètre `key` qui permet de définir la manière dont sont faites les comparaisons ;
- Ce paramètre a l'impact suivant ; pour deux éléments quelconques `a` et `b` de `ma_liste` :

Application à `sorted`

- Rappel : `sorted` possède un paramètre `key` qui permet de définir la manière dont sont faites les comparaisons ;
- Ce paramètre a l'impact suivant ; pour deux éléments quelconques `a` et `b` de `ma_liste` :
  1. `sorted(ma_liste)` :
     place `a` avant `b` si `a < b` ;

Application à `sorted`

- Rappel : `sorted` possède un paramètre `key` qui permet de définir la manière dont sont faites les comparaisons ;
- Ce paramètre a l'impact suivant ; pour deux éléments quelconques `a` et `b` de `ma_liste` :
  1. `sorted(ma_liste)` :
     place `a` avant `b` si `a < b` ;
  2. `sorted(ma_liste, key=fonction)` :
     place `a` avant `b` si `fonction(a) < fonction(b)` ;

Application à `sorted`

- Rappel : `sorted` possède un paramètre `key` qui permet de définir la manière dont sont faites les comparaisons ;
- Ce paramètre a l'impact suivant ; pour deux éléments quelconques `a` et `b` de `ma_liste` :
  1. `sorted(ma_liste)` :
     place `a` avant `b` si `a < b` ;
  2. `sorted(ma_liste, key=fonction)` :
     place `a` avant `b` si `fonction(a) < fonction(b)` ;

- Attention :

Application à `sorted`

- Rappel : `sorted` possède un paramètre `key` qui permet de définir la manière dont sont faites les comparaisons ;
- Ce paramètre a l'impact suivant ; pour deux éléments quelconques `a` et `b` de `ma_liste` :
  1. `sorted(ma_liste)` :
     place `a` avant `b` si `a < b` ;
  2. `sorted(ma_liste, key=fonction)` :
     place `a` avant `b` si `fonction(a) < fonction(b)` ;

- Attention :
  1. c'est bien `key=fonction` sans parenthèses ;

Application à `sorted`

- Rappel : `sorted` possède un paramètre `key` qui permet de définir la manière dont sont faites les comparaisons ;
- Ce paramètre a l'impact suivant ; pour deux éléments quelconques `a` et `b` de `ma_liste` :
  1. `sorted(ma_liste)` :
     place `a` avant `b` si `a < b` ;
  2. `sorted(ma_liste, key=fonction)` :
     place `a` avant `b` si `fonction(a) < fonction(b)` ;

- Attention :
  1. c'est bien `key=fonction` sans parenthèses ;
  2. la signature de `fonction` doit concorder avec l'usage qu'on en fait (ici, elle ne doit comporter qu'un seul paramètre) ;

Application à `min` (ou `max`)

Le même genre d'approche fonctionne pour `min` (ou `max`) :

- `min(L)` renvoie un élément `x` tel que `x <= y` pour tout élément `y` de `L` ;

Application à `min` (ou `max`)

Le même genre d'approche fonctionne pour `min` (ou `max`) :

- `min(L)` renvoie un élément `x` tel que `x <= y` pour tout élément `y` de `L` ;
- `min(L, key=f)` renvoie un élément `x` tel que `f(x) <= f(y)` pour tout élément `y` de `L` ;

Application à `min` (ou `max`)

Le même genre d'approche fonctionne pour `min` (ou `max`) :

- `min(L)` renvoie un élément `x` tel que `x <= y` pour tout élément `y` de `L` ;
- `min(L, key=f)` renvoie un élément `x` tel que `f(x) <= f(y)` pour tout élément `y` de `L` ;

Exemple (`lambdamin.py`)

```
if __name__ == "__main__":
    s = 'voici une phrase dont on veut obtenir le minimum'
    lst = s.split()
    print(min(lst))
    print(min(lst, key=len))
    print(min(lst, key=lambda mot: mot[0]))
    print(min(lst, key=lambda mot: mot[-1]))
```

Utilisation des `lambda`

- Ce concept convient bien quand :

Utilisation des `lambda`

- Ce concept convient bien quand :
  - la définition de la fonction est relativement courte ;

Utilisation des `lambda`

- Ce concept convient bien quand :
  - la définition de la fonction est relativement courte ;
  - on n'a pas vraiment besoin de cette fonction ailleurs ;

Utilisation des `lambda`

- Ce concept convient bien quand :
  - la définition de la fonction est relativement courte ;
  - on n'a pas vraiment besoin de cette fonction ailleurs ;
- Par exemple, quand on veut ajouter une valeur à tous les éléments d'une liste ;

Utilisation des `lambda`

- Ce concept convient bien quand :
  - la définition de la fonction est relativement courte ;
  - on n'a pas vraiment besoin de cette fonction ailleurs ;
- Par exemple, quand on veut ajouter une valeur à tous les éléments d'une liste ;
- On les retrouve souvent dans les appels aux fonctions `map` et `filter`, qu'on va découvrir maintenant ;

Python pas à pas

La fonction `map`

La fonction `map`

- Python fournit un mécanisme très pratique pour appliquer une transformation à tous les éléments d'un itérable ;

La fonction `map`

- Python fournit un mécanisme très pratique pour appliquer une transformation à tous les éléments d'un itérable ;
- On peut avoir recours à la fonction `map(f, I)`, qui applique la fonction `f` à tous les éléments de l'itérable `I` ;

La fonction `map`

- Python fournit un mécanisme très pratique pour appliquer une transformation à tous les éléments d'un itérable ;
- On peut avoir recours à la fonction `map(f, I)`, qui applique la fonction `f` à tous les éléments de l'itérable `I` ;
- **Attention :**

La fonction `map`

- Python fournit un mécanisme très pratique pour appliquer une transformation à tous les éléments d'un itérable ;
- On peut avoir recours à la fonction `map(f, I)`, qui applique la fonction `f` à tous les éléments de l'itérable `I` ;
- **Attention :**
  - `map` ne modifie pas l'itérable ;

La fonction `map`

- Python fournit un mécanisme très pratique pour appliquer une transformation à tous les éléments d'un itérable ;
- On peut avoir recours à la fonction `map(f, I)`, qui applique la fonction `f` à tous les éléments de l'itérable `I` ;
- **Attention :**
  - `map` ne modifie pas l'itérable ;
  - `map` renvoie un nouvel itérable de type `map` contenant les éléments modifiés ;

La fonction `map`

- Python fournit un mécanisme très pratique pour appliquer une transformation à tous les éléments d'un itérable ;
- On peut avoir recours à la fonction `map(f, I)`, qui applique la fonction `f` à tous les éléments de l'itérable `I` ;
- **Attention :**
  - `map` ne modifie pas l'itérable ;
  - `map` renvoie un nouvel itérable de type `map` contenant les éléments modifiés ;
- Si on veut une liste, un ensemble, . . ., il faut procéder au transtypage adéquat ;

Utilisation de `map`

Exemple (map.py)

```
from random import random

if __name__ == "__main__":
    L = [random() for i in range(10)]
    M = list(map(str, L))   # réels -> chaînes
    print(M)
    N = list(map(int, L))   # réels -> entiers
    print(N)
    # multiplier tous les éléments par 10
    P = list(map(lambda x: x * 10, L))
    print(P)
```

Quand utiliser `map` ?

- Remarquons qu'on peut imiter le comportement de `map` à l'aide de compréhensions :

Quand utiliser `map` ?

- Remarquons qu'on peut imiter le comportement de `map` à l'aide de compréhensions :

Exemple (`map` et compréhensions)

```
# ma_liste = ...
# ces deux lignes produisent le même résultat:
M = list(map(str, ma_liste))
N = [str(element) for element in ma_liste]
```

Quand utiliser `map` ?

- Remarquons qu'on peut imiter le comportement de `map` à l'aide de compréhensions :

Exemple (`map` et compréhensions)

```
# ma_liste = ...
# ces deux lignes produisent le même résultat:
M = list(map(str, ma_liste))
N = [str(element) for element in ma_liste]
```

- Quel est donc l'intérêt de `map` ?

Quand utiliser `map` ?

- Remarquons qu'on peut imiter le comportement de `map` à l'aide de compréhensions :

Exemple (`map` et compréhensions)

```
# ma_liste = ...
# ces deux lignes produisent le même résultat:
M = list(map(str, ma_liste))
N = [str(element) for element in ma_liste]
```

- Quel est donc l'intérêt de `map` ?
- Il y en a plusieurs :

Quand utiliser `map` ?

- Remarquons qu'on peut imiter le comportement de `map` à l'aide de compréhensions :

Exemple (`map` et compréhensions)

```
# ma_liste = ...
# ces deux lignes produisent le même résultat:
M = list(map(str, ma_liste))
N = [str(element) for element in ma_liste]
```

- Quel est donc l'intérêt de `map` ?
- Il y en a plusieurs :
  1. `map` donne parfois du code plus concis et compréhensible ;

Quand utiliser `map` ?

- Remarquons qu'on peut imiter le comportement de `map` à l'aide de compréhensions :

Exemple (`map` et compréhensions)

```
# ma_liste = ...
# ces deux lignes produisent le même résultat:
M = list(map(str, ma_liste))
N = [str(element) for element in ma_liste]
```

- Quel est donc l'intérêt de `map` ?
- Il y en a plusieurs :
  1. `map` donne parfois du code plus concis et compréhensible ;
  2. mais surtout, `map` ne produit pas de liste ! : chaque nouvel élément est créé au fur et à mesure ;

Quand utiliser `map` ?

- Remarquons qu'on peut imiter le comportement de `map` à l'aide de compréhensions :

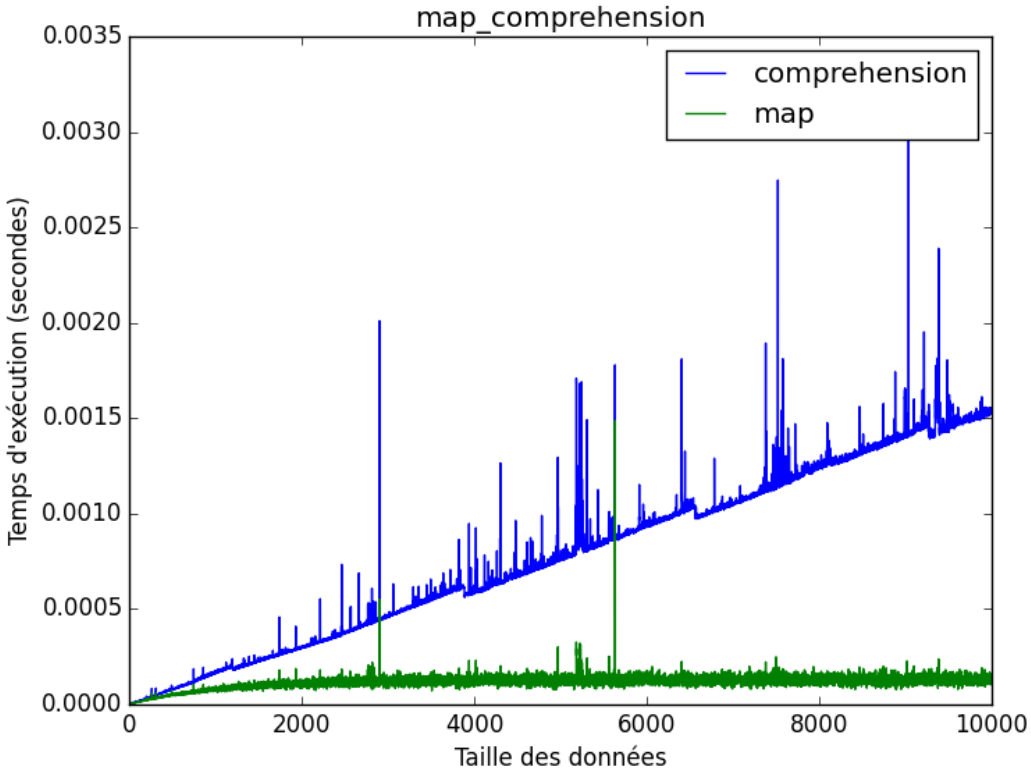
Exemple (`map` et compréhensions)

```
# ma_liste = ...
# ces deux lignes produisent le même résultat:
M = list(map(str, ma_liste))
N = [str(element) for element in ma_liste]
```

- Quel est donc l'intérêt de `map` ?
- Il y en a plusieurs :
  1. `map` donne parfois du code plus concis et compréhensible ;
  2. mais surtout, `map` ne produit pas de liste ! : chaque nouvel élément est créé au fur et à mesure ;
- On consomme donc moins d'espace mémoire, et on a du code plus rapide ;

Comparaison en pratique

- En utilisant ce qu'on a vu au cours d'algorithmique, on peut facilement comparer les deux approches ;
- Ici, le code construit une liste aléatoire d'entiers, puis recherche un élément aléatoire `x` en convertissant tout en chaînes (moyennes sur 50 exécutions) ;



Utilisation dans les doctests

- `map` permet d'écrire des doctests plus flexibles ;

Exemple

Testons si la fonction `couples(n)` renvoie bien tous les couples d'éléments différents de $\{1, 2, \ldots, n\}$:

```
>>> sorted(map(sorted, couples(3)))
[[1, 2], [1, 3], [2, 3]]
```

- Avantages :
  - on ne doit pas se préoccuper de l'ordre de génération des couples, ni des éléments dans les couples ;
  - si les types renvoyés par `couples` changent, on ne doit pas adapter le doctest ;

Conclusions sur `map`

- En conclusion :
  - si l'on n'a pas besoin de conserver l'objet qu'on construit, l'utilisation de `map` est préférable ;
  - sinon, optez pour la version la plus compréhensible ;

- De manière générale : si vous n'observez que des différences négligeables dans vos tests de performances, conservez la version la plus lisible ;

Python pas à pas

La fonction `filter`

La fonction `filter`

- Si l'on veut filtrer des éléments plutôt que les transformer, on peut utiliser `filter` ;

La fonction `filter`

- Si l'on veut filtrer des éléments plutôt que les transformer, on peut utiliser `filter` ;
- La fonction `filter(f, I)` récupère tous les éléments de l'itérable `I` pour lesquels la fonction `f` renvoie **`True`** ;

La fonction `filter`

- Si l'on veut filtrer des éléments plutôt que les transformer, on peut utiliser `filter` ;
- La fonction `filter(f, I)` récupère tous les éléments de l'itérable `I` pour lesquels la fonction `f` renvoie **`True`** ;
- Attention : `filter` renvoie un itérable de type `filter` ;

La fonction `filter`

- Si l'on veut filtrer des éléments plutôt que les transformer, on peut utiliser `filter` ;
- La fonction `filter(f, I)` récupère tous les éléments de l'itérable `I` pour lesquels la fonction `f` renvoie **`True`** ;
- Attention : `filter` renvoie un itérable de type `filter` ;
- Si on veut une liste, un set, . . ., il faut procéder au transtypage adéquat ;

La fonction `filter`

- Si l'on veut filtrer des éléments plutôt que les transformer, on peut utiliser `filter` ;
- La fonction `filter(f, I)` récupère tous les éléments de l'itérable `I` pour lesquels la fonction `f` renvoie **`True`** ;
- Attention : `filter` renvoie un itérable de type `filter` ;
- Si on veut une liste, un set, . . ., il faut procéder au transtypage adéquat ;

Exemple (filter.py)

```
if __name__ == "__main__":
    s = 'voici une phrase dont on veut filtrer les mots'
    lst = s.split()
    print(list(filter(lambda mot: len(mot) > 3, lst)))
    print(list(filter(lambda mot: mot[0] == 'v', lst)))
```

La fonction `filter`

- On peut également (et parfois, on doit) utiliser des méthodes de la classe de l'objet sur laquelle on travaille ;

La fonction `filter`

- On peut également (et parfois, on doit) utiliser des méthodes de la classe de l'objet sur laquelle on travaille ;
- Par exemple, si l'on veut afficher la liste des minuscules d'un texte :

```
>>> texte = "ABCbla"
>>> print(list(filter(str.islower, texte)))
['b', 'l', 'a']
```

La fonction `filter`

- On peut également (et parfois, on doit) utiliser des méthodes de la classe de l'objet sur laquelle on travaille ;
- Par exemple, si l'on veut afficher la liste des minuscules d'un texte :

```
>>> texte = "ABCbla"
>>> print(list(filter(str.islower, texte)))
['b', 'l', 'a']
```

- . . .alors que l'utilisation de `texte` comme paramètre échoue :

```
>>> texte = "ABCbla"
>>> print(list(filter(texte.islower, texte)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: islower() takes no arguments (1 given)
```

Quand utiliser `filter` ?

- Tout comme pour `map`, on peut imiter le comportement de `filter` à l'aide de compréhensions :

Quand utiliser `filter` ?

- Tout comme pour `map`, on peut imiter le comportement de `filter` à l'aide de compréhensions :

Exemple (`filter` et compréhensions)

```
# ma_liste = ...
# ces deux lignes produisent le même résultat:
M = list(filter(lambda x: x % 2, ma_liste))
N = [element for element in ma_liste if element % 2]
```

Quand utiliser `filter` ?

- Tout comme pour `map`, on peut imiter le comportement de `filter` à l'aide de compréhensions :

Exemple (`filter` et compréhensions)

```
# ma_liste = ...
# ces deux lignes produisent le même résultat:
M = list(filter(lambda x: x % 2, ma_liste))
N = [element for element in ma_liste if element % 2]
```

- Le principal intérêt de `filter` est de donner lieu à du code plus concis et compréhensible ;

Quand utiliser `filter` ?

- Tout comme pour `map`, on peut imiter le comportement de `filter` à l'aide de compréhensions :

Exemple (`filter` et compréhensions)

```
# ma_liste = ...
# ces deux lignes produisent le même résultat:
M = list(filter(lambda x: x % 2, ma_liste))
N = [element for element in ma_liste if element % 2]
```

- Le principal intérêt de `filter` est de donner lieu à du code plus concis et compréhensible ;
- Comme les objets filtrés sont copiés à la volée, on économise aussi de la mémoire. . .par contre, le code n'est pas nécessairement plus rapide ;

[Visualisation sur PythonTutor]

Précautions sur `map` et `filter`

Comme pour les `file`, les itérables renvoyés par `map` et `filter` ne peuvent être parcourus qu'une fois.

Exemple

```
>>> x = map(lambda elem: elem + 1, range(10))
>>> 1 in x
True
>>> 1 in x
False
```

Dans ce genre de cas : il faut répéter l'appel à `map` ou `filter`, ou transtyper.