Fiche-Triche pour Python

Julien Grange
<julien.grange@lacl.fr>
UPEC

Généralités

Python voit le jour en 1991. En 2008 paraît **Python 3**, incompatible avec Python 2.

Caractéristiques

- Langage interprété
- Typage dynamique
- Orienté objet
- Mémoire gérée par un ramasse-miettes
- Définition des blocs via l'indentation (4 espaces)

Atouts et faiblesses

Python facilite l'écriture rapide de code, et possède un grand nombre de bibliothèques. L'interprétation du code est en revanche assez lente. Notez que beaucoup de bibliothèques sont écrites en C, et sont très performantes.

Interprétation

Sous Unix, l'interpréteur de Python est `python` (lien symbolique vers `python3`). Il permet d'intepréter le code contenu dans un fichier source (`$ python source.py`) ou d'évaluer du code de manière interactive (`$ python`).
Possibilité d'intégrer du code Python dans un notebook Jupyter (`jupyter.org`).

Syntaxe élémentaire

Affectation :	`a=0`
Importation :	`import bibli` ou `from bibli import *`
Ne rien faire :	`pass`
Commentaire :	`# ceci est un commentaire`

Une valeur spéciale pour représenter l'absence de valeur : `None`. Attention, tous les `None` sont égaux !

Chaînes de caractères

Pas de notion de caractère : un caractère est une chaîne de taille 1.

Littéral :	`'foo'` ou `"foo"`
Formatage :	`"1 %d 3 %s" % (2,"4")`
Répétition :	`s=3*"bla"`
Concaténation :	`"Hello " + "world!"`
Conversion en chaîne :	`str(a)`
Conversion en entier :	`int("42")`
Affichage :	`print(a,b,c)`

Pour récupérer la longueur d'une chaîne, une sous-chaîne, un caractère, etc., se référer au paragraphe sur les **Listes** : la syntaxe est la même.

Expressions booléennes

Deux booléens :	`True` et `False`
(In)égalité :	`a==b` `a!=b` `a<b` `a<=b`
Opérateurs :	`not e` `e1 and e2` `e1 or e2`

L'évaluation est paresseuse : si `e1` est évaluée à `False` dans l'expression `e1 and e2`, alors `e2` ne sera pas évaluée.

Fonctions

```
def mutliplie(a,b=2):
    return a*b
multiplie(3)          évalué à 6
multiplie(3,b=4)      évalué à 12
```

Pour modifier une variable gloable depuis une fonction, la redéclarer au début du corps avec le mot-clef `global`.

Structures de données

Tuples

Affectation :	`t=(42,"foo",True)`	
Tuple vide :	`()`	
Singleton :	`(42,)`	différent de `(42)` !
Accès en lecture :	`t[1]`	évalué à `"foo"`
Destructuration :	`(a,_,b)=t`	`a==42` et `b==True`

Les tuples sont **non mutables** : il est interdit d'écrire `t[i]=x` !

Listes

Déclaration :	`l=[1,2,"3",4]`	
Longueur :	`len(l)`	
Élément :	`l[2]`	évalué à `"3"`
Recherche d'indice :	`l.index(4)`	évalué à `3`
Test d'appartenance :	`2 in l`	évalué à `True`
Sous-liste (tranche) :	`l[1:3]`	évalué à `[2,"3"]`
	`l[1:]`	évalué à `[2,"3",4]`
	`l[:2]`	évalué à `[1,2]`
	`l[-2:]`	évalué à `["3",4]`
Remplacement :	`l[0]="un"`	`l==["un",2,"3",4]`
	`l[:3]=["un",23]`	`l==["un",23,4]`
Suppression :	`del l[2:4]`	supprime la tranche
Copie superficielle :	`copie=l[:]`	
Ajout en queue :	`l.append(5)`	
Retrait en queue :	`l.pop()`	retire et renvoie `5`
Chaîne vers liste :	`list("py")`	évalué à `["p","y"]`

Dictionnaires

Tableaux associatifs (ensemble de paires *clef:valeur*), avec clefs non mutables.

Déclaration :	`d={1:"foo","1":"bar","baz":1}`	
Accès en lecture/écriture :	`d["1"]`	`d[42]=10`

Structures de contrôle

Conditionnelle

```
if cond1:
    bloc_if
elif cond2:
    bloc_elif
else:
    bloc_else
```

Boucle `while`

```
while cond:
    bloc_while
```

Boucle `for`

Plusieurs possibilités (voir les **Itérateurs** pour plus de détails) :

```
for i in range(10):
    bloc_for                        i=0,1,···,9
for i in range(4,16,3):
    bloc_for                        i=4,7,10,13
for i in "abc":
    bloc_for                        i="a","b","c"
for i in [4,2,1]:
    bloc_for                        i=4,2,1
for i,j in enumerate([4,2,1]):
    bloc_for                        (i,j)=(0,4),(1,2),(2,1)
for k in {"a":42,"b":True}:
    bloc_for                        k="a","b"
```

Classes et objets

```
class Etu:
    lstetu=[]                                   attribut de classe
    def __init__(self,nom):                     constructeur
        self.nom=nom                            attribut d'instance
        self.lstetu.add(self)
    def __repr__(self):                         toString
        return "nom :  " + self.nom
    def numetu(self):
        return self.lstetu.index(self)

alice=Etu("Alice")                              instanciation
bob=Etu("Bob")
print(alice)                                    nom :  Alice
bob.numetu()                                    évalué à 1
```

On notera que `bob.numetu()` est traduit en `Etu.numetu(bob)`.
Conventions à respecter :

- les attributs/méthodes commençant par '_' sont privés
- on appelle `self` le premier argument des méthodes

Pour définir l'égalité (`==`) entre objets, implémenter `__eq__` :

```
def __eq__(self,obj):
    if isinstance(obj,MaClasse):
        return self.x==obj.x and self.y==obj.y
    return False
```

Héritage

```
class EtuInfo(Etu):
    def __init__(self,nom,os):
        Etu.__init__(self,nom)
        self.os=os
    def __repr__(self):
        return Etu.__repr__(self)+", os :  "+self.os
```

L'héritage mutliple est autorisé, via "`class B(A1,A2):`".

Exceptions

Pour lever une exception : `raise MonException(···)`, où la classe `MonException` hérite d'`Exception`.

Pour attraper une exception :

```
try:
    bloc_try
except Exception1 as e:     l'exception est capturée dans e
    bloc_exception1
except Exception2:
    bloc_exception2
```

Utilisation avancée des itérateurs

Définitions par compréhension

Liste des carrés des nombres non-divisibles par 3 entre 0 et 100 :

```
[n*n for n in range(100) if n%3!=0]
```

Version dictionnaire, avec les-dits nombres comme clef :

```
{n:n*n for n in range(100) if n%3!=0}
```

Itérateurs

`for ··· in x` fonctionne pour tout **itérable** `x`.
Un itérable est un objet possédant une méthode `__iter__(self)` renvoyant un **itérateur**. Un itérateur est un objet possédant une méthode `__next__(self)` qui

- renvoie l'élément suivant de la séquence à chaque appel
- lève une exception `StopIteration` en fin de séquence

Générateurs

Objet sur lequel on peut itérer, qui se définit comme une fonction, et utilise le mot-clef `yield`, qui fournit la séquence des éléments.
Quand (et si) le générateur termine, une exception `StopIteration` est levée.
Pour générer tous les carrés jusqu'à un seuil :

```
def gen_carres(n):
    i=0
    while i<n:
        yield i*i
        i=i+1
for k in gen_carres(10):
    print(k)                affiche 0,1,4,9,···,81
```

Arguments et fichiers

Arguments en ligne de commande

```
import sys
```

`sys.argv` est la liste des arguments passés en ligne de commande, à la manière de C, en ignorant l'intepréteur. Par exemple, si l'on exécute `$python source.py foo bar`, alors `sys.argv` aura comme valeur `["source.py","foo","bar"]`

Système de fichiers

Les fichers se manipulent via des *file objects*.

open()

`f=open(filename,mode)`
`filename` : chemin vers le fichier
`mode` : mode d'ouverture
`f` : Un *file object*.
Le mode d'ouverture est précisé via une des chaînes suivantes :
`"r"` : lecture seule, curseur en début de fichier (défaut)
`"w"` : écriture seule, tronque le fichier
`"a"` : écriture seule, curseur en fin de fichier
`"r+"` : lecture/écriture, curseur en début de fichier
`"w+"` : lecture/écriture, tronque le fichier
`"a+"` : lecture/écriture, curseur en fin de fichier
`"w"`, `"a"`, `"w+"`, et `"a+"` créent le fichier s'il n'existe pas.

read()

`s=f.read(n)` ou `s=f.read()`
`n` : nombre de caractères à lire ; tout le fichier si `read()`
`s` : chaîne des caractères lus
Pour lire une ligne à la fois, `f.readline()`.
Pour la liste des lignes, `f.readlines()`.
On peut directement itérer sur les lignes, via "`for ligne in f:`".

write()

`n=f.write(s)`
`s` : chaîne de caractères à écrire
`n` : nombre de caractères écrits

seek()

`f.seek(offset,position)`
Pour se déplacer de `offset` caractères dans le fichier, à partir :

- du début du fichier, si `position==0` (valeur par défaut)
- de la position courante si `position==1`
- de la fin du fichier si `position==2`

Retourne la nouvelle position.

close()

`f.close()` : à ne pas oublier une fois qu'on a terminé !

os.remove()

Pour supprimer un fichier :

```
import os
os.remove(filename)
```