Introduction au langage python

Gauthier Picard

SMA/G2I/ENS Mines Saint-Etienne

gauthier.picard@emse.fr

Sommaire

Sommaire

Historique

- Développé en 1990 par Guido van Rossum
- En 1995, sortie de Python 1.6.1, compatible GPL
- Actuellement, deux versions en parallèle : 2.7 et 3.2.3

Licence

- Sous propriété de la Python Software Foundation (PSF)
- Licence libre proche de la licence BSD

Caractéristiques

- très clair, syntaxe lisible
- de fortes capacités d'introspection
- expression naturelle de code procédural
- modularité totale, avec paquets hiérarchiques
- gestion des exceptions
- nombreux types de base : listes, chaînes, tables de hachage
- gestion transparente de la mémoire par référence comptage
- manipulation des références, pas des pointeurs
- lambda-calcul, fonctions anonymes …
- orienté objet intuitive : classes, héritage, exceptions
- typage dynamique : « duck typing »
- vastes bibliothèques standard et des modules de tierces parties
- extensions et modules en C, C ++ (ou Java pour Jython, ou langages .NET pour IronPython)
- intégrable dans les applications comme scripts d'interfaçage

Modes d'exécution

Un langage interprété

- Le programme `python` en mode interactif
- Le programme `ipython` : un shell Python évolué
- La fonction `eval` pour évaluer du code dans python

Un langage compilé

- Compilation en bytecode de machine virtuelle (comme Java)
- Les fichiers `.pyc` sont générés dynamiquement

Syntaxe simple

Éléments de syntaxe

- Nombres : 2
- Chaînes : `'Hello, world'`
- Expressions : `2 * len('Hello, world')`
- Affectations : `a = 2 * len('Hello, world')`
- Commentaires : `a = 2 * b # Double of b`

Règles de syntaxe de Python

- Une instruction par ligne, pas de séparateur
- Affectations multiples autorisées : `a = b = 0`
- Les affections n'ont pas de valeur

Blocs

- Pas de délimiteurs (pas de { et de })
- L'indentation délimite les blocs
- Utilisation : conditions, boucles, fonctions

Exemple (Bloc dans condition)

```
if nb < 0 :
    print "warning, nb < 0"
    nb = 0
max = min + nb
```

Conditions

Syntaxe de `if`

- Syntaxe de base : `if condition: instruction`
- Clause `else` : `else: instruction`
- Contraction du `else-if` : `elif condition: instruction`

Les opérateurs booléens

- Les opérateurs s'écrivent en toutes lettres : and, `or`, `not`
- Le `or` et le and sont *court-circuits*
- La valeur est la dernière expression évaluée : `0 or 1 and 2 or 3` vaut 2

Boucles

La boucle `while`

- Syntaxe de base : `while condition: instruction`
- Fonctionne comme en Java

La boucle `for`

- Syntaxe de base : `for variable in iterable: instruction`
- Un iterable est un objet supportant le protocole *itération* (e.g. les listes)

Exemple (Boucle for avec range)

```
for i in range(5): print 2*i
```

Types atomiques

Tout est objet, en Python ! (méthodes et attributs)

Les types simples

- Nombres (entiers, flottants, complexes) : `42`, `2.5`, `(3+1j)`
- Les booléens : `True` et `False`
- Le type spécial `None`

Les types complexes

- Le type fichier : `file(chemin)`
- Des types divers, comme `code` ou `function`

Listes et tuples

Séquences d'éléments ordonnés

Listes

- *mutable*
- Syntaxe : `[ 2, 3, 5, 7 ]`

Tuples

- *immutable*
- Syntaxe : `( 2, 3, 5, 7 )`

Opérateurs et fonctions

- Accès à un élement : `l[2]`, `l[-2]`
- Accès à une série d'éléments (*slice*) : `l[2:5]`, `l[:-2]`
- Copie d'une liste : `l[:]`
- Longueur : `len`
- Appartenance : `in`
- Concaténation : `+`
- Duplication : `*`

Chaînes

- délimitées par ' et | ' ' |
- chaînes multilignes : ' ' ' et ' ' "' ' '
- *immutables*

Méthodes et opérateurs

- Méthodes : `lower`, `split`, `join`, ...
- Se comportent comme des *tuples* de caractères
- L'opérateur % : formatage `printf` simple et avancé comme en C

Dictionnaires

- Tables de hachage
- clés de type *immutable* et objets de type *mutable* ou pas

Utilisation

- Création de dictionnaire vide : `h1={}`
- Création de dictionnaire non vide :

```
h2={ "answer": 42, "question": None }
```

- Accès à un élément : `h2["answer"]`
- Méthodes : `keys()`, `values()`, `items()`

Exemple (Boucle sur un dictionnaire)

```
for k, v in knights.iteritems():
    print k, v
```

Sommaire

Fonctions

- Syntaxe : `def function(parameters): bloc`
- Valeur de retour : `return`

Le passage de paramètres

- Valeur par défaut
- Appel classique : `mul(4)` ou `mul(4,3)`
- Appel nommé : `mul(a = 4)` ou `mul(b = 3, a = 4)`
- Appel mixte : `mul(4, b = 3)`

Exemple (Définition de fonction avec valeur par défaut)

```
def sum(a, b = 2):
    return a + b
```

Les objets de type `function`

- Les fonctions sont des objets comme les autres
- On peut donc les passer en paramètres (programmation fonctionnelle)
- Utilité : par exemple, la méthode `sort`

Les lambdas

- Fonctions anonymes
- Ne peuvent contenir qu'une expression

Exemple (Appel à la fonction sort() avec lambda)

```
l.sort(lambda s1,s2 : cmp(len(s1),len(s2)))
```

Documentation (ou *doc string*)

- Une *doc string* est un commentaire visible par Python (comme `/** ... */` en Java)
- Exploitable par pydoc pour générer une documenation HTML (comme `javadoc`)
- Elle se met en première ligne d'une fonction, classe ou module
- Elle peut être récupérée par l'attribut __doc__ et la méthode interactive `help()`

Exemple (Documentation par doc string)

```
def add(a, b):
    """This method does an addition"""
    return a + b
print add.__doc__
```

Fonctions à arguments variables

- On passe les paramètres depuis ou vers des conteneurs
- l'opérateur * convertit des arguments non nommés en tuple
- l'opérateur ** convertit des arguments nommés en dictionnaire

Exemple (Utilisation de l'opérateur *)

```
def test_var_args(farg, *args):
    print "formal arg:", farg
    for arg in args :
        print "another arg:", arg

test_var_args(1, "two", 3)
```

Exemple (Utilisation de l'opérateur **)

```
def test_var_kwargs(farg, **kwargs):
    print "formal arg:", farg
    for key in kwargs :
        print "another keyword arg: %s : %s" % (key, kwargs[key])

test_var_kwargs(farg=1, myarg2="two", myarg3=3)
```

Déclaration des classes

Syntaxe

`class name(inheritance): code`

- L'héritage peut se faire sur n'importe quelle classe, ou sur `object`
- Créer un objet se fait en appelant la classe

Sémantique

- Toutes les méthodes sont *virtuelles*
- L'héritage multiple est possible (résolution des conflits C3)
- Fonction `isinstance` et attribut `__class__`
- Méthodes privées avec prefixe `__`

Déclaration des classes (suite.)

Exemple (Définition d'une classe)

```
class Bag :
    def __init__(self):
        self.data = []
    def add(self, x):
        self.data.append(x)
```

Méthodes liées et libres

Le paramètre `self`

- Toute méthode prend en premier paramètre l'objet lui-même
- Par convention, on l'appelle `self`

Les méthodes libres et liées

- Lorsqu'une méthode est récupérée sur un objet, elle est dite *bound*
- Lorsqu'une méthode est récupérée sur une classe, elle est dite *unbound*
- L'objet `self` est implicite dans les méthodes liées, mais doit être spécifié dans les méthodes libres

Méthodes liées et libres (suite.)

Exemple (Méthode liée et non liée)

```
class Foo :
   def foo(self, msg):
       print msg

o = Foo()
fl = o.foo        # methode liee
fl('Hello')

fnl = Foo.foo     # methode non liee
fnl(o,'Hello')
```

Méthodes statiques

Pour créer une méthode statique, il faut écrire une méthode sans le `self` et utiliser le décorateur `@staticmethod`

Exemple (Déclaration d'une méthode statique)

```
class Singleton :
    _instance = None

    def __init__(self):
        assert self._instance == None
        self._instance = self

    @staticmethod
    def getInstance():
        if Singleton._instance == None :
            Singleton._instance = MyClass()
        return Singleton._instance
```

Principes des exceptions

- Très similaires aux exceptions en Java
- Elle se propage, remontant la pile d'appel, jusqu'à être interceptées
- Une exception est une classe héritant de la classe `Exception`

Exemple d'exceptions

- Génériques : `RuntimeError`
- Liées au code : `SyntaxError`, `NameError`, `AttributeError`
- Liées au système : `SystemExit`, `IOError`, `KeyboardInterrupt`
- ...

Traitement des exceptions

Le bloc `try-except`

- Du code pouvant génére une exception *peut* (et non pas *doit*) être entouré d'un bloc `try:` et d'un bloc `except:`
- Le code présent dans le `try` sera exécuté jusqu'à ce qu'une exception survienne
- Le code se trouvant dans le `except` ne sera exécuté que si une exception survient
- Le `except` stoppe la propagation de l'exception

Exemple (Traitement d'une exception)

```
while True :
    try :
        x = int(raw_input("Please enter a number: "))
        break
    except :
        print "Oops!  That was no valid number.  Try again..."
```

Traitement des exceptions (suite.)

Récupération plus évoluée

- Un type peut être précisé dans la clause `except`, pour ne récupérer qu'un type d'exceptions
- Toutes les exceptions héritant de ce type seront aussi interceptées
- Il est possible d'avoir plusieurs bloc `except` de suite
- Tous les autres cas peuvent être interceptés par la clause `else:`
- On peut également récupérer l'objet de type exception avec la syntaxe `except ErrorType as error`

Traitement des exceptions (suite.)

Exemple (Traitement évolué d'une exception)

```
import sys
try :
    f = open('myfile.txt')
    s = f.readline()
    i = int(s.strip())
except IOError as e :
    print "I/O error({0}): {1}".format(e.errno, e.strerror)
except ValueError :
    print "Could not convert data to an integer."
else :
    print "Unexpected error:", sys.exc_info()[0]
```

Nettoyage : traitement après interception

La clause `finally`

- Lorsqu'une erreur survient, on peut avoir besoin de nettoyer
- Le code de la clause `finally` sera exécuté systématiquement, après le reste
- La clause `finally` ne stoppe pas la propagation

Exemple (Clause finally)

```
try :
    12/0
except ZeroDivisionError :
    print "division by zero!"
finally :
    print 'Goodbye, world!'
```

Déclenchement

La clause `raise`

- Déclencher une exception se fait avec le mot-clé `raise`
- Dans une clause `except`, un `raise` sans paramètre propage l'exception courante

Exemple (Déclenchement d'une exception)

```
try :
     raise NameError('HiThere')
except NameError :
     print 'An exception flew by!'
     raise
```

Sommaire

Utilisation de modules et paquetages

La clause `import`

- Permet d'importer un module externe
- On peut préciser un nom avec as

Exemple (Import d'un module)

```
import os
import os.path as ospath
print os.getenv("PATH")
print ospath.join("home", "picard", ".emacs")
```

Utilisation de modules et paquetages (suite.)

La clause `from`

- Permet d'importer des symboles directement dans l'espace courant
- Permet de n'importer que ce dont on a besoin
- Il est possible, mais déconseillé, d'utiliser *

Exemple (Clause from)

```
from os import *
from sound.effects import echo
```

Définition de modules et paquetages

Création de modules

- Un module est un fichier définissant des symboles
- Le code est exécuté à l'importation (sauf cas `__name__ == "__main__":`)
- Un module peut en importer d'autres, mais pas de manière circulaire

Création de paquetages

- Un paquetage est répertoire contenant des modules (et éventuellement des paquetages)
- Il doit contenir un fichier `__init__.py`

Sommaire

Références utiles

Ce support est très largement inspiré du support de par Gaël LE MIGNOT & Jérôme PETAZZONI :

http ://cours.pilotsystems.net/cours-insia/cours-de-python-a-linsia-pour-ing1-et-ing2srt/slideshow.pdf

Livres et liens divers

- la documentation officielle de python : http ://docs.python.org/index.html
- un cours qui reprend le tutoriel en grande partie, en français : http ://web.univ-pau.fr/~puiseux/enseignement/python/python.pdf
- un très bon livre, gratuit et complet : http ://inforef.be/swi/download/apprendrepython.pdf
- traduction en français d'un livre référence : http ://diveintopython.adrahon.org/
- association francophone python : http ://www.afpy.org/
- une fiche synthétique bien pratique : http ://rgruet.free.fr/PQR27/PQR2.7.html