Introduction à Python

E. RAMAT

Université du Littoral - Côte d'Opale

13 septembre 2007

Plan

Introduction

Qu'est ce que Python ?

- interprété - machine virtuelle ;
- interactif ;
- portable ;
- orienté objets.

Introduction

Historique

- inventé en 1990 par Guido Van Rossum
- le nom de “Python” provient “Monty Python’s Flying Circus”, la série-TV
- prévu pour être un langage de scripts
- Python a été influencé par ABC et Modula-3
- première release en 1991

Introduction

- conçu pour être simple ;
- permet la programmation modulaire ;
- lisible ;
- développement rapide d'applications ;
- facile à intégrer ou à étendre avec d'autres langages.

- package Debian :
  - apt-get install python2.5 python2.5-doc
- peut être utilisé en mode interactif ou batch ;
- IDLE est un éditeur pour écrire et exécuter des programmes Python :
  - apt-get install idle-python2.5

- extensible (packages) ;
- “embarquable” dans des applications ;
- programmation orientée objets ;
- prototypage rapide ;
- compilation à la volée ;
- faible coût de maintenance ;
- libre ;
- l’espace est significatif (caractère d’intention) ;
- gestion des exceptions.

- “glue code” : langage permettant de faire facilement le lien entre différences applications ;
- script système ;
- applications graphiques : pygtk, par exemple ;
- applications de base de données et XML ;
- applications Web : Zope, par exemple.
- protocole réseau : librairie email, par exemple.

Plan

- typage dynamique :
  - les variables ne doivent pas être déclarées;
  - les variables doivent être initialisées ce qui affecte un type à la variable ;
- seules les valeurs sont typées ;
- presque tout peut être affecté à une variable (fonction, module, classe, ...).

Types

Types de base

- numérique (integer et float) :
  - decimal (631 ou 3.14) ;
  - octal (0631) ;
  - hexadecimal (0xABC) ;
  - long (122233445656455L) ;
- tous les opérateurs classiques (arithmétiques et binaires) ;
- attention au troncage sur les entiers : 1/2 = 0 ;
- mais aussi les complexes ;
- les booléens (True et False) et les opérateurs logiques.

Types
String

- concatenation : `"Hello" + "World" -> "HelloWorld"`
- repetition : `"UMBC" * 3 -> "UMBCUMBCUMBC"`
- index : `"UMBC"[0] -> "U"`
- sous-chaîne : `"UMBC"[1:3] -> "MB"`
- taille : `len("UMBC") -> 4`
- comparaison : `"UMBC" < "umbc" -> 0`
- recherche : `"M" in "UMBC" -> 1`
- il est possible d'utiliser les simples quotes pour délimiter les strings ;
- les strings sont non modifiables ("immutable").

Types
Liste

- valeur : [1,2,3,4] ;
- les éléments de la liste ne sont pas nécéssairement du même type ;
- manipulable comme un tableau indexé ;
- ajout en fin de liste ;
- mêmes opérateurs que pour les strings ;
- + les operations : append(), insert(), pop(), reverse() et sort().

Liste

```
>>> a = ['titi', 16, 'toto', 86]
>>> a[0]
'titi'
>>> a[3]
86

# indiçage par la fin : 0 dernier, -1 avant-dernier, ...
>>> a[-2]
'toto'

# de l'indice 1 à l'avant-dernier
>>> a[1:-1]
[16,'toto']

# construction d'une liste comme la sous-liste [0:2] concaténée à
# liste ['tutu', 2*2] où 2*2 est évalué
>>> a[:2] + ['tutu', 2*2]
['titi', 16, 'tutu', 4]
```

Liste

```
# construction d'une liste comme trois fois la sous-liste [0:3] +
>>> 3*a[:3] + ['Hello!']
['titi', 16, 'toto', 'titi', 16, 'toto', 'titi', 16, 'toto', 'Hell

# remplacement d'un élément (impossible avec les strings)
>>> a[1] = a[1] + 23
>>> a
['titi', 39, 'toto', 86]

# remplacement de quelques éléments
>>> a[0:2] = [1, 12]
>>> a
[1, 12, 'toto', 86]

# suppression d'éléments
>>> a[0:2] = []
>>> a
['toto', 86]
```

Liste

```
# insertion
>>> a[1:1] = ['a', 'b']
>>> a
['toto', 'a', 'b', 86]

# ajout d'une copie de la liste au début
>>> a[:0] = a
>>> a
['toto', 'a', 'b', 86, 'toto', 'a', 'b', 86]

# suppression de tous les éléments
>>> a[:] = []
```

Types

Tuple

- valeur : (1,2,3,4) ;
- non modifiable ;
- composé de valeurs hétérogènes.

Types

Dictionnaire

- liste de couples clé-valeur entourée d'accolades ;
- valeur : Map = "a":1,"b":2 ;
- accès (ajout d'une nouvelle entrée) : Map["a"] retourne 1 ;
- insertion : Map["c"] = 3 ;
- suppression : del Map["b"] ;
- iterations :
  - sur les clés : keys() ;
  - sur les valeurs : values() ;
  - sur les couples : items() ;
- existence : has_key("c") ;
- les valeurs peuvent être de n'importe quel type ;
- les clés doivent être non modifiables.

Dictionnaire

```
dico = {}          # dictionnaire vide
dico['I'] = 'je'
dico['she'] = 'elle'
dico['you'] = 'vous'
print dico         # {'I':'je', 'she':'elle', 'you':'vous'}
print dico['I']    # 'je'

del dico['I']
print dico         # {'she':'elle', 'you':'vous'}
```

Dictionnaire

```
dico.keys()           # ['we', 'she', 'you']
dico.values()         # ['nous', 'elle', 'vous']
dico.items()          # [('we', 'nous'), ('she', 'elle'),
('you', 'vous')]
dico.has_key('I')     # False
dico.has_key('you') # True
```

- a = b : pas de création de copie de b ;
- b = a, a et b référence le même objet ;

Dictionnaire

```
>>> a = [1,2,3]
>>> b = a
>>> a.append(4)
>>> print b
[1, 2, 3, 4]
```

Plan

Structures de contrôle

- if condition : instructions (elif condition : instructions)* [else : instructions]
- while condition : instructions
- for var in sequence : intructions
- break
- continue

Structures de contrôle

Condition

If

```
if x < 0:
    print "x est négatif"
elif x % 2:
    print "x est positif et impair"
else:
    print "x n'est pas négatif et est pair"
```

Tabulation

Les blocs sont délimités par le caractère tabulation.

For

```
for i in range(5):
    print i,          # 0 1 2 3 4

for i in range(1,5):
   print i, # 1 2 3 4
```

range

La fonction range génére une liste de valeurs arithmétiques (range(max) -> 0 .. max ; range(min,max) ; range(min,max,[step])).

Séquence et dictionnaire

```
>>> map = {'a': 1, 'b': 2}
>>> for k, v in map.items():
...     print k, v
...
a 1
b 2
```

Séquence et énumération

```
>>> for i, v in enumerate(['a', 'b', 'c']):
...     print i, v
...
0 a
1 b
2 c
```

Plan

Procédures et fonctions

- forme générale :

Procédures et fonctions

```
def f(arg1, arg2, ...):
    Intructions
    return    # pour les procédures
    return expression  # pour les fonctions
```

- “Return” n’est pas obligatoire pour les procédures.

Procédures et fonctions

Paramètres

- valeur par défaut :

Définition

```
def f(a,b,c):
...
def f(a=0,b=1,c=2):
...
```

- appel :

Appel

```
f(0,1)
f()
f(b=8,a=6)
```

- passage d'un nombre quelconque de paramètres sous forme d'un tuple :

Définition

```
def somme(*nombres):
    resultat = 0
    for nombre in nombres:
      resultat += nombre
    return resultat

print somme(23, 42)     # affiche : 65
```

- passage d'un dictionnaire :

Définition

```
def un_dict(**mots):
    return mots

print un_dict(a=23, b=42)
# affiche : {'a':23, 'b':42}
```

- filter(fonction, séquence) : retourne la séquence composée des éléments de la liste passée en paramètre dont la fonction retourne true ; la séquence peut être une liste, un tuple ou un string ;
- map(fonction, séquence) : appelle la fontion pour chaque élément de la séquence et retourne une liste des valeurs retournées par la fonction ;
- reduce(fonction, séquence) : retourne une valeur simple construite grâce à l'appel à la fonction (à deux paramètres) avec les deux premiers éléments de la séquence et le résultat de l'appel et l'élément suivant, etc ...

filter

```
>>> def f(x): return x % 2 != 0 and x % 3 != 0
...
>>> filter(f, range(2, 25))
[5, 7, 11, 13, 17, 19, 23]
```

map

```
>>> def cube(x): return x*x*x
...
>>> map(cube, range(1, 11))
[1, 8, 27, 64, 125, 216, 343, 512, 729, 1000]
```

reduce

```
>>> def sum(seq):
...     def add(x,y): return x+y
...     return reduce(add, seq, 0)
...
>>> sum(range(1, 11))
55
>>> sum([])
0
```

Plan

Module et package

Module

- un module est un fichier qui contient des définitions et des instructions Python ;
- le fichier doit avoir une extension .py ;
- dans un module, le nom du module est accessible via la variable globale __name__ ;
- l'instruction "import module-name" permet d'importer les fonctions du module importé ;
- certains modules sont prédéfinis comme sys.

Module et package

Package

- la notion de package est accessible via les “namespaces” ;
- la notation est intégrée dans le nom du module avec le point ;
- la structuration des packages reposent sur le système de fichiers ;
- par exemple :
  - A.B.C désigne le module C du sous-package B dans le package A ;
  - le module C est donc placé dans le sous-répertoire B du répertoire A;
- pour importer le module C :
  - “import A.B.C” ; il faut alors préfixer les éléments utilisés par A.B.C ;
  - “from A.B import C” ; il faut alors préfixer les éléments utilisés par C seulement ;
  - “from A.B.C import *” ; pas de préfixe.

Plan

Classe et objet

Classe

- une classe possède deux sortes d'attributs :
  - des données ;
  - des fonctions (appelées méthodes) ;
- Python définit des méthodes spéciales ;
- une classe peut hériter d'autres classes.

- tous les attributs sont publiques ;
- toutes les méthodes sont “virtuelles” (polymorphes) ;
- toutes les méthodes sont plubliques sauf si vous faites précéder le nom de la méthode par __ : le nom de la fonction est transformé en _nom-classe__fonction et on ne peut alors invoquer directement la fonction (ça “simule” la notion de private) ;
- il n’y a pas vraiment de constructeur :
  - on peut définir une fonction spéciale __init__ pour initialiser les attributs des instances ;
  - le concept de destructeur n’existe pas.

- définition :

Définition

```
class ClassName:
    statements

class ClassName(BaseClass1, BaseClass2...):
    statements
```

- documentation :

Documentation

```
class C:
    ''Une classe''
```

Classe et objet

Attribut

- attribut de classe :

Attribut de classe

```
class C:
    x = 1
```

- attribut d'instance :

Attribut d'instance

```
class C:
    ...
    self.x = 1

# création d'une instance
a = C()
# création d'un attribut
a.x = 1
```

Classe et objet

Attribut

- tout est dynamique : la preuve !
- suppression d'un attribut

Suppression

```
# création d'une instance de la classe C
a = C()
# suppression de l'attribut x
delete a.x
```

Classe et objet

Initialisation

- la fonction __init__ :
  - sans paramètre :

Attribut de classe

```
def __init__(self):
    self.data = []
```

Définition

le mot clé **self** désigne la future instance d'objet ou l'objet.

  - avec paramètres :

Attribut d'instance

```
class Complex:
    def __init__(self, realpart, imagpart):
        self.r = realpart
        self.i = imagpart
```

Classe et objet

Méthodes

- méthode de classe :

Attribut de classe

```
class C:
    def f():
        instructions
...
C.f()
```

- méthode d'instance :

Attribut d'instance

```
class C:
    def f(self):
        instructions
...
a = C()
a.f()
```

Classe et objet

Surcharge des opérateurs

- permet à un opérateur de posséder un sens un comportement spécifique par classe :

Opérateurs

```
x = 7+9              # addition entière
c = 'ab' + 'cd'      # concaténation
```

- Python possède des méthodes de surcharge :
  - tout type : __call__, __repr__, ...
  - nombre : __add__, __div__, ...
  - séquence : __len__, __iter__, ...

Surcharge d'opérateurs

```
class Vecteur2D:
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def __add__(self, other):    # addition vectorielle
        return Vecteur2D(self.x + other.x, self.y + other.y)

    def __repr__(self):          # pour l'affichage
        return "Vecteur(%g, %g)" % (self.x, self.y)
```

- Méthode __del__ : appelée quand un objet est détruit ;
- c'est le "garbage collector" qui décide la destruction quand il n'y a plus de références sur l'objet ;
- un objet peut être aussi détruit par l'instruction del ;
- la méthode __del__ prend un seul paramètre self.

- on peut invoquer le “constructeur” de la super-classe :

Constructeur

```
class Base:
    def __init__(self):
        self.x = 1

class Derivee(Base):
    def __init__(self):
        Base.__init__(self)
        self.y = 1
```

Méthode statique ou de classe

```
class A(object):
    def f():
        print('coucou')
    f = staticmethod(f)

A.f()

a = A()
a.f()

class B(object):
    def f(cls):
        print(cls.__name__)
    f = classmethod(f)

A.f()

a = A()
a.f()
```

Singleton

```
class Singleton(object):
  _singletons = {}

  def __new__(cls):
    if not cls._singletons.has_key(cls):
      cls._singletons[cls] = object.__new__(cls)
    return cls._singletons[cls]
```

Question

La méthode __new__ est invoquée lors de la création d'une instance et elle retourne une instance.

Plan

Exceptions

- capture des exceptions : `try ... except`

Exceptions

```
from math import sin

for x in xrange(-3, 4):
    try:
        print "%.3f" % (sin(x)/x), # cas normal
    except:
        print 1.0,             # gère l'exception en 0
```

Exceptions

- levée d'une exception : `raise`

Exceptions

```
if (x < 0) or (x > 1):
    raise ValueError, "la valeur n'est pas dans [0 .. 1]"
```

- il existe une liste d'exceptions prédéfinies :
  `http://docs.python.org/api/standardExceptions.html`

Exceptions

- définir sa propre exception :

Exceptions

```
class MyError(Exception):
    def __init__(self, value):
        self.value = value

    def __str__(self):
        return repr(self.value)
```

Plan

Réflexivité

Définition

La réflexion ou l'introspection est la capacité d'un programme à examiner, et éventuellement à modifier, ses structures internes de haut niveau (par exemple ses classes, ses objets, ...) lors de son exécution.

Instance en Python

Une instance en Python se résume en un dictionnaire contenant : les attributs et les méthodes.

Réflexivité

- Python permet :
  - d'interroger les classes et les objets ;
  - de modifier dynamiquement la structure et le comportement des classes et des objets ;

Définition

```
class C:
    y=0

    def __init__(self):
        self.x = 0

a=C()
```

Réflexivité

- dir(item) : retourne la liste des méthodes et des attributs d'une classe ou d'un objet

dir

```
print dir(a) # ['__doc__', '__init__', '__module__', 'x']
print dir(C) # ['__doc__', '__init__', '__module__', 'y']
```

- type(item) : retourne le type d'une classe ou d'un objet

type

```
print type(a)       # <type 'instance'>
print type(a.x)     # <type 'int'>
print type(C)       # <type 'classobj'>
```

- callable(attribut) : retourne True si l'attribut est une méthode.

Réflexivité

- hasattr(objet, nom d'attribut) : retourne True si l'objet (ou la classe) possède un attribut dont le nom est donné ;
- getattr(objet, nom d'attribut) : retourne une référence sur l'attribut dont le nom est donné s'il existe ;
- setattr(objet, nom d'attribut, valeur) : affecte l'attribut.

Métaclasse

- une métaclasse est une classe dont les instances sont des classes ;
- sous Python, la métaclasse de base est `type` ;
- quelque soit l'objet manipulé, l'exécution de monObjet.__class__.__class__ retourne `type` ;
- pour créer une nouvelle métaclasse, il suffit de créer une sous-classe de `type`.

Métaclasse

- lors de la création d'une instance, les méthodes __new__() et __init__() sont invoquées ;
- __new__() est une méthode de classe qui "alloué" l'instance et prend quatre paramètres :
  - la métaclasse elle-même,
  - le nom de la classe instanciée,
  - ses superclasses sous forme d'une liste,
  - un dictionnaire contenant l'ensemble de ses méthodes et attributs ;
- __init__() est une méthode d'initialisation des instances. Ses paramètres sont les mêmes que ceux de __new__() hormis le premier.

Métaclasse

Métaclasse (fr.wikipedia.org - Métaclasse)

```
from types import FunctionType
class Tracer(type):
  def __new__(metacls, name, bases, dct):
      def _wrapper(method):
          def _trace(self, *args, **kwargs):
              print "(call␣%s␣with␣*%s␣**%s)"
               % (method.__name__, str(args), kwargs)
              return method(self, *args, **kwargs)
          _trace.__name__ = method.__name__
          _trace.__doc__  = method.__doc__
          _trace.__dict__.update(method.__dict__)
          return _trace

      newDct = {}
      for name, slot in dct.iteritems():
          if type(slot) is FunctionType:
              newDct[name] = _wrapper(slot)
          else:
              newDct[name] = slot
      return type.__new__(metacls, name, bases, newDct)
```

Métaclasse

- définition d'une fonction _wrapper de telle sorte qu'elles affichent un message avant de continuer leur exécution ;
- cette fonction est un pattern Decorator (elle prend en entrée une fonction et renvoie en sortie cette fonction avec un comportement modifié);

Métaclasse

- proposer une forme de d'éfinition de classe où l'on peut spécifier la liste des attributs de la classe avec leur initialisation.

Métaclasse (en.wikipedia.org - Metaclass)

```
class AttributeInitType(type):
  def __call__(self, *args, **kwargs):
      """ Create a new instance. """
      # First, create the object in the normal default way.
      obj = type.__call__(self, *args)
      # Additionally, set attributes on the new object.
      for name in kwargs:
          setattr(obj, name, kwargs[name])
      # Return the new object.
      return obj
```

- la méthode __call__ est invoquée lorsqu'une instance est appelée comme une fonction. Par exemple, lorsque l'on écrit : a = C().

Métaclasse

Métaclasse (en.wikipedia.org - Metaclass)

```
class Car(object):
  __metaclass__ = AttributeInitType
  __slots__ = ['make', 'model', 'year', 'color']

  @property
  def description(self):
     """ Return a description of this car. """
     return "%s %s %s %s"
      % (self.color, self.year, self.make, self.model)

cars = [
 Car(make='Toyota', model='Prius', year=2005, color='green'),
 Car(make='Ford', model='Prefect', year=1979, color='blue')]
```

- __slots__ est une façon de réserver l'espace mémoire pour les attributs des instances.

Plan

Assertion

- les assertions permettent d'insérer du code de contrôle en mode debug :

assert

```
if __debug__:
  if not expression: raise AssertionError
```

- en mode normal, les assertions sont activées ;
- l'option -O supprime les assertions.

Tests unitaires

- le processus de développement basé sur les tests :
  - on écrit un test ;
  - le test ne passe pas ;
  - on écrit la fonctionnalité au plus simple ;
  - le test passe.
- les tests unitaires sont utilisés dans un processus de développement comme eXtrem Programming (XP).

Tests unitaires

pyUnit

- les tests s'organisent autour de la classe TestCase du package unittest ;
- il suffit de définir une classe héritant de TestCase et de développer la fonction runTest :

TestCase

```
class MonTest(unitest.TestCase):
  def runTest(self):
    assert ma_fonction(2,5) == 0
```

- le test vérifie que ma_fonction retourne 0 si les paramètres sont égaux à 2 et 5 ;
- si le test échoue alors l'assertion est fausse et une exception (AssertionError) est levée ;
- l'exception permet d'identifier quel test a échoué.

Tests unitaires

pyUnit

- si un ensemble de tests nécessite les mêmes initialisations, la méthode setUp permet d'isoler les initialisations ;

TestCase

```
class MonTest(unitest.TestCase):
  def setUp(self):
    self.x = 10

class Test1(MonTest):
  def runTest(self):
    assert ma_fonction(self.x,self.x)

class Test2(MonTest):
  def runTest(self):
    assert ma_fonction(self.x,8)
```

- la méthode setUp peut aussi être utilisé dans un test unique ... juste pour séparer les initialisations et les tests.

- l'exécution des tests passent par une instance de TestSuite qui construit la suite des tests à réaliser ;

TestSuite

```
t = unittest.TestSuite()
t.addTest(MonTest('runTest'))
```

Lien entre la suite et le test

Le constructeur du test admet en paramètre le nom de la fonction à appeler dans le test. Il est donc possible d'avoir plusieurs fonctions dans la même classe de test.

Tests unitaires

pyUnit

- il faut ensuite instancer le moteur d'exécution de la suite des tests ;

RunnerTest

```
m = unittest.TextTestRunner()
m.run(t)
```