Python

Jérôme Petazzoni
<jp@enix.fr>

Qu'est-ce que c'est ?

Langage de script « évolué »

Nombreux types de base : entiers, chaînes, listes, tables de hachage

Orientation objet : classes, héritage, introspection

Gestion transparente de la mémoire par *référence comptage*

On ne manipule pas de pointeurs mais des références (=Java)

On peut définir des fonctions anonymes (~lambda-calcul, Caml)

Syntaxe concise, claire et cohérente (je vous jure que si!)

```
(lambda t_debut, octets, t_fin: octets/(t_fin-t_debut))(
   time.time(),
   len(urllib.urlopen(myurl).read()),
   time.time())
```

Petit historique...

- Première version : février 1991
- Version 2.0 en 2000
- Aujourd'hui : version 2.4

- versions embarquées
  (pour téléphone portable, pour Palm, en Java...)

- *stackless* Python
  (utilisé pour le moteur du jeu EVE, entre autres)

Langage interprété ou compilé ?

- Langage interprété
  comme BASIC, Caml, PHP, TCL, Perl...
- ...mais aussi compilé
  comme Caml, Java, C, C++...
- Machine virtuelle à *bytecode*
  comme Caml, Java
- Génération du *bytecode* automatique :
  lorsqu'on exécute toto.py, le fichier toto.pyc est créé à la volée
- Machine virtuelle JIT disponible (psyco)

Typage fort ou faible ?

- Typage fort : C, Java
- définition très stricte des variables, des fonctions...
- généricité :
  par héritage
- “super” généricité :
  bof (void*, Object)

- Typage faible : PHP, Perl
- les chaînes sont des entiers et inversement proportionnel
- avantage :
  code plus compact
- inconvénient :
  porte ouverte à toutes les fenêtres

Que valent, dans vos langages favoris :

"2" + 4	"2" * 4
4 + "2"	4 * "2"

Le typage en Python

- Typage faible :

```
>>> def add(a,b): return a+b
```

- Mais fort quand même :

```
>>> print type(12), type(12.0), type(“12.0”)
<type 'int'> <type 'float'> <type 'str'>
```

- Tout n'est pas permis :

```
>>> 12+”12” → TypeError
```

```
>>> 2*2 → 4
```

```
>>> 2*”2” → “22”
```

Un petit exemple

```
jpetazzo@skalinka:~$ cat coin.py
#!/usr/bin/python
import sys
print reduce(lambda a,b:a+b,
             [int(x) for x in sys.stdin],
             0)
```

Illisible à souhait, n'est-ce pas ?

Les types : entiers et flottants

```
>>> 2+3*4
14
>>> 1<<8
256
>>> 5**3
125
>>> 42%10
2
>>> 2**30
1073741824
>>> 2**31
2147483648L
```

```
>>> import math
>>> math.cos(math.pi/3)
0.50000000000000011
>>> 2**(1.0/12)
1.059463094359295 3
>>> 2**(1/12)
1
>>> int(“0123”)
123
>>> 2.0**64
1.8446744073709552e+19
```

Les types : chaînes

```
>>> “abc”+”def”
'abcdef'
>>> “abc”*2
'abcabc'
>>> “abc%sdef%d”%(“toto”,4)
'abctotodef4'
>>> “%08X”%42069
'0000A455'
>>> “%%%s”%d%12
12
>>> “of” in “lord of the rings”
True
```

```
>>> len(“toto”)
4
>>> “toto”.upper()
'TOTO'
>>> “      x yz   “.strip()
'x yz'
>>> “albert”.startswith(“al”)
True
>>> “salamalec”.replace(“al”,”ila”)
'silaamilaec'
>>> "l'an de l'ananas".count("an”)
3
>>> “     piano\n”.strip()
'piano'
```

Les types : tuples & listes

```
>>> [1,2,4,8]+[16,32]
[1, 2, 4, 8, 16, 32]
>>> (1,2,3)*2
(1, 2, 3, 1, 2, 3)
>>> liste=[“toto”,42,True]
>>> liste[0]=123
>>> liste
[123, 42, True]
>>> 123 in liste
True
>>> 123 not in liste
False
```

```
>>> pasliste=(1, False)
>>> pasliste[0]
1
>>> pasliste[0]=69
TypeError: object doesn't
support item assignment
>>> list(pasliste)
[1, False]
>>> range(4)
[0, 1, 2, 3]
>>> tuple(range(4))
(0, 1, 2, 3)
```

Les types : tables de hachage

```
>>> bazar={“cac40”:857.4, “jaune”: (255,255,0), “vrai”:
True, 42: “quarante-deux”, “liste”: [4, 2, {False:
“faux”}], (0,0):”origine”}
>>> bazar[“liste”]
[4, 2, {False: “faux”}]
>>> bazar[“liste”][2][False]
'faux'
>>> bazar.has_key(“truc”)
False
>>> bazar[“truc”]=”chose”
>>> “truc” in bazar
True
>>> del bazar[“truc”]
>>> dict([(“k1”,10),(“k2”,”v”)])
{'k1': 10, 'k2': 'v'}
```

Massacre à la tronçonneuse

```
>>> “abcdefgihj”[4]
'e'
>>> “abcdefghij”[4:7]
'efg'
>>> [1,2,4,8,16][2:]
[4, 8, 16]
>>> s=“the knights who say ni”
>>> s[-2:]
'ni'
```

```
>>> s = s.split(“ “)
>>> s
['the', 'knights', 'who',
'say', 'ni']
>>> s[1]='lemmings'
>>> s[:-1]=['oh', 'no']
>>> s.join(“ “)+'!'
'the lemmings who say oh no!'
```

```
dict([kv.split(“=”) for kv in
      open(“toto.conf”).read().split(“\n”)])


dict([kv.split(“=”,1) for kv in
      open(“toto.conf”).read().split(“\n”)
      if “=” in kv and not kv.strip().startswith(”#”)])
```

Aide-mémoire (chaînes)

```
>>> dir('')
['__add__', '__class__', '__contains__', '__delattr__',
'__doc__', '__eq__', '__ge__', '__getattribute__',
'__getitem__', '__getnewargs__', '__getslice__', '__gt__',
'__hash__', '__init__', '__le__', '__len__', '__lt__',
'__mod__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmod__', '__rmul__',
'__setattr__', '__str__', 'capitalize', 'center', 'count',
'decode', 'encode', 'endswith', 'expandtabs', 'find',
'index', 'isalnum', 'isalpha', 'isdigit', 'islower',
'isspace', 'istitle', 'isupper', 'join', 'ljust', 'lower',
'lstrip', 'replace', 'rfind', 'rindex', 'rjust', 'rstrip',
'split', 'splitlines', 'startswith', 'strip', 'swapcase',
'title', 'translate', 'upper', 'zfill']
```

Aide-mémoire (listes)

```
>>> dir([])
['__add__', '__class__', '__contains__', '__delattr__',
'__delitem__', '__delslice__', '__doc__', '__eq__',
'__ge__', '__getattribute__', '__getitem__',
'__getslice__', '__gt__', '__hash__', '__iadd__',
'__imul__', '__init__', '__iter__', '__le__', '__len__',
'__lt__', '__mul__', '__ne__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__rmul__', '__setattr__',
'__setitem__', '__setslice__', '__str__', 'append',
'count', 'extend', 'index', 'insert', 'pop', 'remove',
'reverse', 'sort']
```

Aide-mémoire (tables de hachage)

```
>>> dir({})
['__class__', '__cmp__', '__contains__', '__delattr__',
'__delitem__', '__doc__', '__eq__', '__ge__',
'__getattribute__', '__getitem__', '__gt__', '__hash__',
'__init__', '__iter__', '__le__', '__len__', '__lt__',
'__ne__', '__new__', '__reduce__', '__reduce_ex__',
'__repr__', '__setattr__', '__setitem__', '__str__',
'clear', 'copy', 'fromkeys', 'get', 'has_key', 'items',
'iteritems', 'iterkeys', 'itervalues', 'keys', 'pop',
'popitem', 'setdefault', 'update', 'values']
```

Typage pas si faible que ça

```
>>> type(42)
<type 'int'>

>>> type (“toto”) == type (“titi”)
True

>>> class Cbase: pass
>>> class Cderivee: pass
>>> Ibase=CBase() ; Iderivee=Cderivee()
>>> isinstance(Ibase,Cbase)
True
>>> isinstance(Iderivee,Cbase)
True
>>> isinstance(Ibase,Cderivee)
False

>>> callable(42), callable(Cbase)
(False, True)
```

Délimitation des blocs de code - 1

- Avec des tabulations (non? si!)

```
if condition1 or condition2:
   faire_truc()
   faire_autre_truc()
else:
   faire_la_vaisselle()
```

- C'est l'équivalent en C ou Java, de :

```
if (condition1 or condition 2) {
   faire_truc() ;
   faire_autre_truc() ;
} else {
   faire_la_vaisselle();
}
```

Délimitation des blocs de code - 2

- Le point-virgule en fin de ligne est optionnel
- On peut l'utiliser pour mettre plusieurs instructions par ligne :

```
faire_ci() ; faire_ca() ; faire_autre_chose()
```

- On peut utiliser des barres obliques inverses (a.k.a. *backslash*) pour continuer une instruction sur la ligne suivante

```
print 2+3*4/27% \
42+37
```

- Il est parfois plus simple d'utiliser des parenthèses :

```
print (2+3*4/27
       %42+37)
```

Structures conditionnelles

```
if condition: si_vrai()
elif: autre_truc()
else: si_faux()

while condition:
   iteration()
   if fini: break
   if pouet: continue
   autre_chose()

for element in liste:
    manip(element)
    if foo(element): break
    if quux(element): continue
    chose(element)
```

Exceptions - 1

```
try:
   buffer += socket.read(4096)
   # ...
except OSError,e:
   if e.errno!=errno.EAGAIN: raise
except DiskFullException:
   print “Le disque est plein.”
except:
   save_the_work()
   do_something_with(sys.exc_traceback)
```

Exceptions - 2

```
try:
   du_code()
   raise Exception # exception générique
   #raise Exception('message explicatif') # un peu mieux
   #raise NetworkException(clientsocket, errno)
                    # par exemple...
except NetworkException:
   ...
except Exception:
   ...
```

Exceptions - 3

```
try:
   du_code()
   raise Exception(“bonjour”)
finally:
   # sera exécuté dans tous les cas
   # (et avant les éventuels handlers d'exception)
   save_work_in_progress()
   print “au revoir”
```

Définition de fonctions

```
def aire(x1,y1,x2,y2):
    delta_x=x2-x1
    delta_y=y2-y1
    return abs(delta_x*delta_y)

def aire(a,b,x2=None,y2=None):
    if x2: x1,y1=a,b
    else: (x1,y1),(x2,y2)=a,b
    return abs((x2-x1)*(y2-y1))
```

- cette deuxième version marche avec

  `aire(1,2,3,4) mais aussi aire((1,2),(3,4))`

Encore plus de fonctions

```
def wget(host,port=80,uri="/"):
    return urllib.urlopen("http://%s:%d%s"%(host,port,uri)).read()

wget("google.com")

wget("google.com",80,"/search?q=perdu")

wget("google.com",uri="/search?q=perdu")

wget(uri="/search?q=perdu",host="google.com")
```

- Interdit : fonction(param=valeur,autreparam)

```
SyntaxError: non-keyword arg after keyword arg
```

Fonctions variadiques

- Fonctions variadiques :

```
def printf(format_string, *arguments):
    print format_string%arguments

printf(“%s: %d”,”trois”,3)

liste=[“riri”, “fifi”, “loulou”, 17]
printf(“%s + %s + %s = %d”,*liste)

toto(a,b,c) = toto(*[a,b,c])
```

Arguments nommés arbitraires

```
def apply_font(font=”Helvetica”, size=12, **styles):
    print “Font name is %s.”%font
    print “Font size is %d.”%size
    print “Extra attributes:”
    for attrname,attrvalue in styles.items():
        print “\t%s: %s”%(attrname,str(attrvalue))
```

```
>>> apply_font(size=24,color=”red”,background=”black”)
Font name is Helvetica.
Font size is 24.
Extra attributes:
    background: black
    color: red
```

```
>>> apply_font(**{“color”: “red”, “background”: “black”})
```

Les noms des paramètres doivent bien sûr être des chaînes.

Commentaires et *docstrings*

```
# avec des dièses

“ou bien des chaînes littérales”

“””
ou encore, des triple-quotes
“””


def racine_carree(x):
   “””Cette fonction calcule la racine carrée
du nombre fourni comme paramètre.”””
   return x**0.5

help(racine_carree)
help.__doc__
```

L'itération revisitée

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> [x*x for x in range(10)]
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> [x*x for x in range(10) if x%2]
[1, 9, 25, 49, 81]

>>> map(lambda x:x*x, range(10))
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
>>> filter(lambda x:x%2,[1, 4, 2, 3, 7, 8, 13, 42])
[1, 3, 7, 13]

>>> h={“un”:1, “deux”:2, ”trois”:3}
>>> for key in h: print key,
un trois deux
```

Une fonction peut en cacher une autre

```
>>> daltons=[{"nom":"joe","taille":140,"caractere":"teigneux"},
...          {"nom":"jack","taille":155,"caractere":"idiot"},
...          {"nom":"william","taille":170,"caractere":"stupide"},
...          {"nom":"averell","taille":185,"caractere":"abruti"}]

>>> def cmp_attr(attr):
...     def my_cmp(d1,d2): return cmp(d1[attr],d2[attr])
...     return my_cmp
>>> daltons.sort(cmp_attr(“nom”))

>>> def compose(f,g):
...     def fg(*l): return g(f(*l))
...     return fg
>>> def plus4(x): return x+4
>>> def fois2(x): return x*2
>>> h=compose(plus4,fois2)
>>> h(10)
28
```

Une fonction lambda

```
>>> compose=lambda f,g:lambda x:g(f(x))
>>> plus4fois2=compose(lambda x:x+4,lambda x:x*2)
>>> plus4fois2(5)
18
```

- Guido Von Rossum regrette d'avoir introduit en Python les “lambda”, “map”, “filter” et “reduce” ; et aimerait bien s'en débarrasser dans les versions futures de Python
- “map” et “filter” sont remplacées par les “list comprehensions” [f(x) for x in s if p(x)]
- “reduce” est très peu usité
- introduction probable de all(s) et any(s)

Fonctions génératrices

- Problème concret : engendrer les carrés inférieurs à une valeur donnée
- Solution “directe” :

```
>>> def squares_up_to(x):
...     return [n*n for n in range(int(math.sqrt(x)))]
```

- Solution génératrice :

```
>>> def squares_up_to(x):
...     i=0
...     while i*i<x:
...         yield i*i
...         i+=1
```

- Dans les deux cas, on peut faire :

```
>>> for x in squares_up_to(20): print x,
```

Petit tour à la bibliothèque

- http://doc.python.org/lib/
- sys : sys.argv, sys.stdin... principalement
- os : fonctions “bas niveau” (=appels système)
- os.path : manipulations de noms de fichier
- re : expressions régulières
- math : manipulations de nombres flottants, voire complexes
- pickle : (dé)sérialisation
- shelve : file-backed hashtable

Manipulations de fichiers

```
>>> f=open(filename,mode="r")
```

- Les modes sont les mêmes que pour la fonction fopen(3)

```
>>> f.read() # lit tout le fichier
>>> f.read(1024) # lit 1024 octets
>>> f.readline() # lit une ligne (avec le \n)
>>> f.write("toto")
>>> f.close()
>>> import sys
```

- On peut utiliser sys.stdin, sys.stdout, sys.stderr

```
>>> for zorglub in sys.stdin: # itère sur les lignes
>>> sys.stdout = open("/tmp/log.txt","a")
```

Manipulations de répertoires

```
>>> import os
>>> print os.listdir(os.environ[“HOME”])

>>> for name in os.listdir(os.environ[“HOME”]):
...     path=os.path.join(os.environ[“HOME”],name)
...     print “%s: %d octets”%(name,os.stat(path).st_size)

>>> for dirpath, dirnames, filenames in os.walk(“/”):
...     for dirname in dirnames:
...         print os.path.join(dirpath,dirname)
...     for filename in filenames:
...         print os.path.join(dirpath,filename)

>>> os.open(“/tmp/toto”,os.O_RDWR|os.O_CREAT,0644)
3
>>> os.write(3, “coin\n”)
5
```

Expressions régulières : la théorie

- import re
- re.match(regex, chaîne)
- re.search(regex, chaîne)
- retournent (en cas de match) un objet de type Match
- en cas de recherches multiples : utiliser re.compile
- un truc pratique : les raw strings

```
>>> "abc\n\1def"
'abc\n\x01def'
>>> r"abc\n\1def"
'abc\\n\\1def'

>>> re.search(r”t(.)t\1”, “le tutu”)
<_sre.SRE_Match object at 0xb7de5de0>
```

Expressions régulières : la pratique

```
>>> m = re.match(“(le|la|les) (.*) (est|sont) (.*)”,
...                “le ciel est bleu et dégagé”)
>>> m.groups()
('le', 'ciel', 'est', 'bleu et dégagé')

>>> m = re.match(“(le|la|les) (?P<sujet>.*) (est|sont)
(?P<attribut>.*)”, “le soleil est haut dans le ciel”)
>>> m.groupdict()
{'sujet': 'soleil', 'attribut': 'haut dans le ciel'}
```

- Nombreuses extensions folkloriques
- Cas d'école : *log parser* avec plusieurs centaines de regex par ligne...
- → Un appel de fonction, ça coûte cher !

Pickle (mise en boîte – de conserve)

- Sauvegarde d'éléments :

```
>>> import pickle
>>> f=open(“/tmp/toto”,”w”)
>>> pickle.dump(47,f)
>>> pickle.dump(sys.argv,f)
>>> pickle.dump({“canard”:”coin”, “chat”:”miaou”},f)
```

- Rechargement des données :

```
>>> f=open(“/tmp/toto”)
>>> try:
...     while 1: print pickle.load(f)
... except EOFError: pass
```

- On ne peut pas tout sauver (sockets, modules...)
- dumps et loads permettent de (dé)sérialiser des données.

Shelve

- Table de hachage stockée sur disque
- Restrictions : les mêmes que pour pickle
- De plus, les clés doivent être des chaînes

```
>>> import shelve
>>> s=shelve.open("/tmp/s")
>>> s[str(4)]="quatre"
>>> s["quatre"]=4
>>> s["h"]={1:2, 3:4}
>>> "absent” in s
False
```

Le langage le plus classe du monde

```
class networkoutput:
    """Redirects file output to a remote server"""
    def __init__(self, host, port):
        """host and port to connect (with a TCP socket)"""
        from socket import *
        self.socket=socket(AF_INET,SOCK_STREAM)
        self.socket.connect((host,port))
    def write(self, data):
        self.socket.send(data)


sys.stdout=networkoutput("logger.domain.com",4000)
```

- Constructeur : méthode __init__
- Toutes les méthodes prennent “this” comme premier argument (qu'on appelle traditionnellement “self”, mais ça n'est pas obligatoire)

Champs spéciaux

```
def f(): pass
f.__name__

from os import *
walk.__module__

class c(): pass
i=c()
i.__class__
i.__class__.__name__

class xy:
    def __init__(self,x,y): self.x,self.y=x,y
o=xy(0,0)
o.__dict__

→ {'y': 0, 'x': 0}
```

Chirurgie interne

```
class c:
    def __init__(self,v): self.v=v
    def f(self,x): print x
    def g(self,x): print x, x
    def getv(self): print self.v
i=c(1)
j=c(2)

c.__dict__[“f”](i, 10)

c.__dict__[“g”](None, 20)

i.getv=j.getv
i.getv()
→ 2
```

Smooth operators

- Comme en C++, on peut redéfinir de nombreux opérateurs :
- __add__ __sub__ __mul__ __div__ __neg__ __pos__
- __eq__ __ne__ __ge__ __gt__ __le__ __lt__
- __abs__ __pow__ __mod__ __lshift__ __rshift__
- __and__ __or__ __xor__ __invert__
- __repr__ __str__ __int__ __oct__ __hex__ __long__
- __cmp__ __hash__ __nonzero__ __coerce__
- __getattribute__ __setattr__ __delattr__
- __getitem__ __setitem__ __delitem__
- __getslice__ __setslice__ __delslice__
- __len__ __contains__ __call__

Exemple – classe mylist

```
class mylist:
    def __init__(self, *l):
        self.l = l
    def __call__(self):
        return self.l
    def __len__(self):
        return len(self.l)
    def __getitem__(self,i):
        print 'élément',i
        return self.l[i]
    def __add__(self,l):
        return mylist(*self.l+l())
    def __repr__(self):
        return "mylist(%s)"%(",".join([repr(i)
                                       for i in self.l]))
    def __str__(self):
        return ",".join([str(i) for i in self.l])
```

Exemple – classe groupe

```
class mygroup:
    def __init__(self, modulo, n=0):
        self.value = n ; self.modulo = modulo
    def __add__(self,x):
        return mygroup(self.modulo,(self.value+x)%self.modulo)
    def __sub__(self,x):x
        return mygroup(self.modulo,(self.value-x)%self.modulo)
    def __radd__(self,x):
        return mygroup(self.modulo,(x+self.value)%self.modulo)
    def __rsub__(self,x):
        return mygroup(self.modulo,(x-self.value)%self.modulo)
    def __neg__(self):
        return mygroup(self.modulo)-self
    def __repr__(self):
        return “mygroup(%d,%d)”%(self.value,self.modulo)
        return "mygroup(%(value)d,%(modulo)d)”%self.__dict__
    def __str__(self):
        return str(self.value)
    def __int__(self):
        return self.value
```

Exemple – table de hash (1)

```
class myhash:
    def __init__(self, *l, **kv):
        self.h = {}
        for k,v in l: self.h[k]=v
        self.h.update(kv)
    def __getattr__(self, k):
        if k in self.h: return self.h[k]
        raise AttributeError(k)
    def __repr__(self):
        return repr(self.h)
    def __str__(self):
        return str(self.h)
```

Exemple – table de hash (2)

```
class myhash:
    def __init__(self, *l, **kv):
        self.__dict__["h"] = {}
        for k,v in l: self.h[k]=v
        self.h.update(kv)
    def __getattr__(self, k):
        if k in self.h: return self.h[k]
        raise AttributeError, k
    def __setattr__(self, k, v):
        self.h[k]=v
    def __repr__(self):
        return repr(self.h)
    def __str__(self):
        return str(self.h)
```