Introducere python

Ionel Măries Cristian

Python

- duck typed (php, javascript, ruby etc)
- masina virtuala care ruleaza bytecode, codul python este compilat in bytecode
- e un limbaj "general-purpose"
- 2 versiuni majore (2.x, 3.x)
- 3.x e incompatibil cu 2.x, majoritatea codului existent e pentru 2.x

Python

- accent pe claritate, corectitudine (ex: fara efecte secundare in expresii, gen `i++` sau `if (x=mumu()) {alert(x)}`)
- blocurile delimitate prin indentare
- sintaxa similara cu pseudocodul

Sintaxa: exemple

```
# comentariu

if bla:
  foo
elif:
  bar

while condition:
  bla
else:
  foo
```

Sintaxa: for

- functioneaza pe obiecte iterabile (liste, seturi, generatori, dictionare si alte structuri de date cu interfata de iterator)

```
for i in [0,1,2]:
  print i
```

sau

```
for i in range(3):
  print i
```

Sintaxa: try/except

polimorfic (tot ce mosteneste clasa Exception)

```
try:
  raise Exception("nasol")
except Exception, e:
  print e
```

grup de exceptii, tot polimorfic (tot ce mosteneste IndexError si KeyError)

```
try:
  raise IndexError("yeah")
except (IndexError, KeyError), e:
  print e
```

Sintaxa: try/except cont.

finally, folosit la cleanup etc.

```
f = None
try:
  f = file("FOO")
  f.read()
except IOError:
  print 'ceva pute'
finally:
  if f:
    f.close()
```

Sintaxa: with

```
with file("mumu.txt") as f:
   for line in f:
     print line
```

f este inchis automat la iesirea din bloc

- adaugat in python 2.5
- e un manager de context
- obiectele care implementeaza interfata de context manager pot face lucruri la intrarea si iesirea din bloc, tratand in mod elegant exceptiile.

Sintaxa: alte cuvinte cheie

- continue
- break
- pass (bloc gol, ca si `{ }` in C)
- assert
- del
- print
- si altele …

Tipuri de date

builtins: int, float, bool, list, dict, tuple, set, frozenset, string, unicode etc

```
barlist = [1,2,3]
somedict = {1:2, 'a':'b'}
mytuple = (1,2,3)
multime = set([1,2,3])
string1 = 'asdf'
string2 = 'asdf'
stringmultilinie1 = """asdf
bla
more bla"""
stringmultilinie2 = '''blabla"bla"asd'''
```

Tipuri de date cont. 1

Interpolare in siruri de caractere:

```
"Ana are %s." % "mere"
"Ana are %s si %s." % ("mere", "pere")
```

Metode utile la diverse tipuri de obiecte:

- liste: append, extend, remove, index, count, insert, pop, reverse, sort (inplace)
- string: split, splitlines, strip, replace, join, find, upper, lower, encode, decode
- dictionare: copy, clear, update, pop, popitem, keys, values, items etc.

Tipuri de date cont. 2

Operatori/functii builtin utili:

```
mylist[index]
mylist[index] = value
value in mylist
mylist[start:end:step] # slicing
mydict[key] = value
key in mydict
key not in mydict
del mydict[key]
del obj.attr
len(mydict), len(mylist), len(sequence_type)
str(obj), repr(obj), dir(obj)
```

etc.

Tipuri de date: unicode string

```
# -*- coding: utf8 -*-

a = "Ionel Măricş" # regular string (tip str)
b = u"Ionel Măricş" # unicode string (tip unicode)
assert a.decode('utf8') == b
assert b.encode('iso-8859-2') ==
a.decode('utf8').encode('iso-8859-2')
```

- unicode nu este un encoding, e un tip de date
- conversia de la str -> unicode se face prin *decodarea* printr-un anumit encoding
- conversia de la unicode -> str se face print *encodare*

Functii

```
def mumu(foo, bar):
    bla
```

la interpretorul interactiv:

```
>>> def mumu(a, b, *args, **kwargs):
...     print a, b, args, kwargs
...
>>> mumu(1, 2, 3, 4, foo=6, bar=6)
1 2 (3, 4) {'foo': 6, 'bar': 6}
```

Expresii: boolean

- `a or b` - intoarce primul care evalueaza ca si True, sau `b` in caz contrar
- `a and b` - intoarce `a` daca evalueaza la False, `b` altfel
- operator ternar (`cond?a:b`):
  - `cond and a or b` - ok ca si echivalent de operator ternar daca `a` nu evalueaza ca si fals
  - `a if cond else b` - introdus in python 2.5

Expresii: facilitati functionale

- `lambda foo, bar: foo+bar`
- functii builtin: filter, map
- list comprehension (in principiu o expresie de map+filter):

  [expr for name in sequence if condition]

Exemplu:

```
>>> [2*i for i in range(10) if i%2]
[2, 6, 10, 14, 18]
```

Modules

a.py

```
def foo():
    pass
```

b.py

```
import a
a.foo()
```

```
from a import foo
foo()

from a import foo as bar
bar()

from a import *
foo()
```

Module: pachete (packages)

Ex:

```
atelier/__init__.py
atelier/ciocane.py
atelier/cuie.py
```

- __init__.py a fost creat un mecanism de preventie - directoare de pe caile de import care nu contin module ar fi putut avea precedenta fara de modulele care vrem sa le importam - un lucru nedorit.
- __init__.py poate contine cod de initializare

mumu.py:

```
import atelier
```

- executa atelier/__init__.py

Module: pachete (packages) cont. 1

Ex:

```
atelier/__init__.py
atelier/ciocane.py
atelier/cuie.py
```

mumu.py:

```
from atelier import *
```

- executa atelier/__init__.py
- importa toate variabilele din __init__.py SAU daca exista __all__ importa simbolurile din lista respectiva

Module: pachete (packages) cont. 2

atelier/__init__.py

```
__all__ = ['ciocane', 'cuie']
import ciocane
import cuie
```

```
import bla, foo, bar
```

mumu.py

```
from atelier import * # va importa doar modulele ciocane si
cuie
```

Variabile

```
a = [1,2,3]
b = a
c = (1,2,3)
d = c
```

- toate atribuirile se fac prin referinte
- unele obiecte nu pot fi modificate (int, float, tuple, frozenset, string, unicode string) - obiecte imutabile
- restul sunt mutabile (list, dict, set etc)
- obiectele imutabile pot contine obiecte mutabile, deci imutabilitatea nu e acelasi lucru ca si o valoare constanta

Variabile cont. 1

- `id(bla)` - identitatea lui `bla`, pe cpython adresa obiectului (detaliu de implementare)
- comparatia de identitate se face cu operatorul `is` (daca simbolurile `a` si `b` refera la acelasi obiect)

```
>>> a = b = []
>>> c = []
>>> a is b
True
>>> a is c
False
```

Clase

```
class Foo:
  """Acesta este un 'docstring' - un string pentru
documentatie care va fi asociat acestei clase."""
  def __init__(self, a, b):
    self.a = a
    self.b = b

  def do_stuff(self):
    "Aceasta metoda face o chestie extrem de complicata."
    print self.a + self.b
```

- fiecare metoda ia instanta ca si parametru implicit (ex `foo = Foo(); foo.do_stuff()`)
- __init__ este echivalentul unui constructor
- `self` e doar o conventie de nume folosita de toata lumea
- nu exista distinctie intre metode sau atribute

Clase: mostenire

```
class Bar:
  def __init__(self, a, b):
    pass

class Foo(Bar):
  def __init__(self, a, b):
    Bar.__init__(self, a, b)
```

- nu exista supraincarcare de metode (method overloading) in python, metodele din subclase le suprascriu pe cele din ancestori chiar daca au parametrii diferiti
- comparativ cu alte limbaje (gen c++, java) toate metodele sunt virtuale

Clase: polimorfism

```
class Bar(object):
  def __init__(self, a, b):
    pass

class Foo(Bar):
  def __init__(self, a, b):
    super(Foo, self).__init__(a, b)
```

- clasele ce mostenesc `object` sunt numite *new style classes*, introduse in python 2.2
- pot folosi facilitati mai avansate ca si descriptori, proprietati calculate etc.
- `super` functioneaza cu clase *new style*, rezolva unele probleme cauzate de mostenirea multipla

Clase: proprietati calculate

```
class Bar:
  def _get_foo(self):
    return self._foo

  def _set_foo(self, val):
    self._foo = val

def _del_foo(self):
  del self._foo

foo = property(_get_foo, _set_foo, _del_foo, "Docstring")
```

Decoratori

```
def logger(func):
  def newfunc(*args, **kws):
    logging.log
    return func(*args, **kws)

def foo():
  pass
foo = logger(bar)

#sintaxa de decorator:

@logger
def bar():
  pass
```

Metode "magice"

`__nume_metoda__`

Obiectele pot avea niste metode speciale (conventia e ca incep si se termina cu __) pentru a oferi functionalitate ca si:

- suprascriere de operatori (__add__, __sub__, __mod__ etc.)
- getteri si setteri (__getattr__, __setattr__ etc.)
- acces de cheie/index, ex: foo[index] (__getitem__, __setitem__ etc.)
- tipurile builtin imutabile (int, tuple, string etc) implementeaza o interfata de hash si pot fi folosite ca si chei in dictionare (__hash__)
- reprezentarea obiectelor ca si string, similar cu toString in unele limbaje (__str__, __repr__)
- apelul (__call__)

Metode "magice" exemplu

```
>>> class plist(list):
...     def __iadd__(self, other):
...         self.append(other)
...         return self
...     def __and__(self, other):
...         new = self.__class__(self)
...         new.extend(other)
...         return new
...
>>> x = plist([1,2,3,4])
>>> x += 5
>>> x
[1, 2, 3, 4, 5]
>>> x & [6,7,8]
[1, 2, 3, 4, 5, 6, 7, 8]
>>> x
[1, 2, 3, 4, 5]
```

Generatori

Un generator este o functie care contine instructiunea `yield`

```
>>> def stuff():
...     yield 1
...     yield 2
...     yield 2
...
```

```
>>> for i in stuff():
...     print i
...
1
2
2
>>>
```

Iteratori

- un obiect poate intoarce un iterator prin metoda `__iter__`
- iteratorul implementeaza metodele `next` si `__iter__` (care va intoarce instanta curenta)
- o functie tip generator intoarce un obiect care are o interfata de iterator

```
>>> g = stuff()
>>> g is g.__iter__()
True
>>> g.next()
1
>>> g.next()
2
>>> g.next()
2
>>> g.next()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

- instructiunea `for` incearca sa apeleze interfata de iterator de pe obiect sau interfata de indexare (`obj[index]`)

Descriptori

- pot fi utilizati pe clase "new style"
- sunt niste obiecte implementate tot ca si clase new style
- care implementeaza __get__, __set__ si __del__
- pe scurt, accesul la atributele dintr-o clasa care sunt descriptori este delegat la metodele respective
- multa functionalitate din python este implementata ca si descriptori, in speta, functiile, metodele, proprietatile, metodele statice si de clasa, etc

Descriptori exemplu

```
>>> class ClassNameProperty(object):
...   def __get__(self, instance, owner):
...     return owner.__name__
...
>>> class FooBar(object):
...   myname = ClassNameProperty()
...
>>> a = FooBar()
>>> a.myname
'FooBar'
```

Metaclase

o solutie in cautare de probleme

- o clasa este compusa dintr-un dictionar (`__class__`) care contine rezultatul instructiunilor din blocul clasei - ca si un namespace, un nume, si un set de clase parinti
- clasele sunt instante de metaclase
- metaclasele pot suprascrie crearea clasei prin `__new__` si chiar si crearea instantelor de clasa prin suprascrierea operatorului de apel al clasei (`__call__`)
- instructiunea class e mai degraba 'syntactic sugar' pentru crearea clasei
- aplicatiile sunt nenumarate: logging, sincronizare, creare automata de proprietati, singletonuri (suprascrii __call__ in metaclasa) etc