Programming language design and analysis

Mihai Oaida

15 December 2010

Agenda

- Background
- Data and Control structures , Syntax
- Procedural Python
- OOP
- Under the hood
- Functional Python
- Decorators
- Concurrency
- Who is using it?

Background

- invented by Guido Van Rossum in 1989
- Based on ABC
  - Focus on clear syntax
  - Nesting by indentation
  - Infinite lists and numbers
- Current 2.7 and 3.1 (backwards incompatible)

Philosophy

- ABC was used to teach programming
- basic constructs and grammar to describe what you want to do
- code that is as understandable as plain English

```
if "L" in "Linux":print "Hello World"
```

Implementations

- cPython – the original one
- Pypy – python in python
- Jython – for the JVM
- IronPython - .NET integration
- Tinypy – 64k of code

Simple types

```
>>> a = 200
>>> b = 2**0.5 # square root
>>> x = 3+4j
>>> name = "mihai"
>>> name = u"mi\u021b\u0103"
>>> print name
miță
>>>
```

Iterable types

```
>>> tuple = (1,2,3) #immutable
>>> print tuple
(1, 2, 3)
>>> list = [1,2,3]
>>> list.append(4)
>>> list[2:]+list[:2]
[3, 4, 1, 2]
>>len(list) #can use sum,max,min
4
>>> get = {"page":"new","id":3}
```

Tricks

```
print "Hi"*20

a,b=b,a

print "Linux"[1:-1]

newName = name[:]

#in operator can be used for iterables
1 in range(10) # True
```

Control structures

```
for number in range(1000):
    if number%5 == 0 and number%11 == 0:
        print number
```

- Also has: elif, while
- Does not have: switch

Functions - definition

```
def fact(n):
    if n ==0 :
        return 1
    else:
        return n*fact(n-1)

print fact(5)
```

Functions – reference passing

```
def compute(list,int):
    list.append(4)
    Int = 43

a_list=[1,2,3]

a_int = 3

compute(a_list,a_int)

print a_list # [1, 2, 3, 4]
print a_int  # 3
```

Functions – named arguments

```
def print_info(name,age,height=183):
    "keyword arguments and default arguments"
    print "%s is %d years old and %s cm tall"%
(name,age,height)

print_info(name= "John",age=23)

print print_info.__doc__
```

Functions – variable-length arguments

```
def multi_add(*vartuple):
    "variable-length arguments"
    return sum(vartuple)

print multi_add(1,2,3,4)
```

OOP

Multiple inheritance
No public, private, protected
Conventions are used
No interfaces
No overloading
Methods are virtual

this is self, and not reserved

Objects can be modified at runtime
- add members
- add methods

OOP - instance

```
class Person():
    name = None
    age = None

    def __init__(self,name,age):
        self.name = name
        self.age = age
    def show(self):
        print self.name,self.age


me = Person("Mihai",24)
me.show()
```

OOP – Multiple inheritance

```
class Kid(Mom,Dad):
    def cutTimber(self):
        Dad.cutTimber(self)
        #cut timber

    def cook(self):
        if "cook" in dir(Dad):
            Dad.cook(self)
        else:
            Mom.cook(self)
```

OOP – setters and getters

```
class D():
    x = None
    y = None
    privates = ["x","privates"]
    def __setattr__(self, name, val):
        if name not in self.privates:
            self.__dict__[name] = val
        else:
            raise Exception("%s is read only" %
(name))
```

OOP - Exceptions

```
try:
    print 10/0
except ZeroDivisionError,e:
    print e
finally:
    print "Execute anyway"
```

```
a=2
try:
   a=a*b
except NameError,e:
    print e#name 'b' is not defined
```

Modules

- Like modula-3
- Modules are files, with a separate namespace

```
#import module
import math
print math.pi


#import into current global namespace
from math import *
print pi
```

Under the hood

- assignments creates references
- for mutable objects use copy.copy
- dynamic typing, but when declared types are enforced
- duck typing
- objects have types
- integer vs float division
- type,isinstance , and __class__

Under the hood

- .pyc files – fast loading
- garbage collector
  - reference counting
  - reference cycles
- JIT – pysco, pypy

Functional programming

```
lambda x:x+1

print (lambda x,y:x**y)(2,10)#1024
```

```
print map(lambda a: a*a, [1,2,3,4,5])#[1, 4, 9,
16, 25]
```

```
numbers = [1,2,3,4,5]
numbers_under_4 = filter(lambda x: x < 4,
numbers)
print numbers_under_4#[1, 2, 3]
```

```
numbers = [1,2,3,4,5]
print reduce(lambda a,b: a*b, numbers)#120
```

List comprehension

- From Haskell
- Lists to generate lists
- Combine the power of for, if and lists in one line

```
[x for x in range(1000) if x%5 ==0 and
x%11==0]
```

```
sum([int(x) for x in str(2**1000)])
```

Generators

- For creating iterators

```
def double(list):
    for i in list:
        yield i
        yield i

for nr in  double(range(10)):
    print nr
```

Generators

- Yield saves state

```
def fib():
    p,q = 0,1

    while True:
        p,q=q,p+q
        yield p

seq = fib()
for i in range(10):print seq.next()
```

Decorators

- Functions that transform functions

```
def inc(func):
    return lambda x:x+1

@inc
def example(a):
    return a

print example(41)#42

print inc(example)(41)#42
```

Concurrency

- Threads
  - cPython has a GIL
  - Jython and ironPython do not
- Processes
  - CSP – using decorators

Who is using it?

- YouTube, Google, NASA , Linux, Bit torrent
- Used to teach programming
- Has influenced Cobra and Ocaml(twt) on nesting and syntax
- EcmaScript with iterators, generators and list comprehensions
- Go, Groovy, Boo with philosophy

Questions?