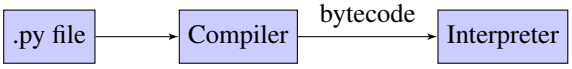
Python 2.7

Description
Python is a scripting language.

Contents

1. Introduction

1.1 Compile and Interpret[1] [2]



When we run a python script, first it's compiled. Not to machine code but to bytecode. This compilation make .pyc files. Then this code is interpreted in a VM. There are a lot interpreeters of python: CPython, Jython or IronPython.

2. Variables and Definitions[3]

2.1 Variables

There is not types. The definition is just the name and the value.

```
n = 24
s = 'Hola'
a = ['a', 'b', 'c']
>>> range(5, 10)
[5, 6, 7, 8, 9]
```

In while statement, continue statement continues with the next iteration.

2.2 Function

```
>>> def fib(n = 1):    # write Fibonacci series up to n
...     """Print a Fibonacci series up to n."""
...     a, b = 0, 1
...     while a < n:
...         print a,
...         a, b = b, a+b
...
>>> fib(2000)
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
>>> fib
<function fib at 10042ed0>
>>> f = fib
>>> f(100)
0 1 1 2 3 5 8 13 21 34 55 89
>>> f() # NEVER modify the default parameter (NEVER DO n
    =...)
1
```

2.2.1 Lambda functions

```
>>> def make_incrementor(n):
...     return lambda x: x + n
...
>>> f = make_incrementor(42)
>>> f(0)
42
>>> f(1)
43
```

2.3 Scope[4]

LEGB Rule: define the order of search variable.

- L. Local. (Names assigned in any way within a function (def or lambda)), and not declared global in that function.
- E. Enclosing function locals. (Name in the local scope of any and all enclosing functions (def or lambda), form inner to outer.
- G. Global (module). Names assigned at the top-level of a module file, or declared global in a def within the file.

[2] https://docs.python.org/2/library/dis.html
[3] https://docs.python.org/2/tutorial/index.html
[4] http://stackoverflow.com/questions/291978/short-description-of-python-scoping-rules

- B. Built-in (Python). Names preassigned in the built-in names module : open,range,SyntaxError,...

The for loop does not have it's own namespace. It would look in the LEGB order.

So first search local variables, then enclosing function locals, Global variable and finally built-in functions.

2.4 Modules

If we have the definition of fib function in fibo.py file, we can use it like a module definiing it like this.

```
>>> import fibo
>>> fibo.fib(1000)
1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
>>> from fibo import fib
>>> fib(500)
1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

2.5 Classes

```
class Dog:
  kind = 'canine'         # class variable shared
      # by all instances

  def __init__(self, name):
    self.name = name    # instance variable
      # unique to each instance

>>> d = Dog('Fido')
>>> e = Dog('Buddy')
>>> d.kind                  # shared by all dogs
'canine'
>>> e.kind                  # shared by all dogs
'canine'
>>> d.name                  # unique to d
'Fido'
>>> e.name                  # unique to e
'Buddy'
```

For private methods and functions, a leading underscore is conventionally added.

2.5.1 Multiple Inheritance

```
class DerivedClassName(Base1, Base2, Base3):
  <statement-1>
  ...
  <statement-N>
```

2.6 Error and exceptions

```
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."
except:
  print "Unexpected error:", sys.exc_info()[0]
raise
```

2.6.1 Raising exception

The raise statement allows the programmer to force a specified exception to occur.

```
>>> raise NameError('HiThere')
Traceback (most recent call last):
File "<stdin>", line 1, in ?
NameError: HiThere
```

3. The Python Standard Library[5]

3.1 Iterators and Generator[6]

```
>>> i = iter('abc')
>>> i.next()
'a'
>>> i.next()
'b'
>>> i.next()
'c'
```

Generator using yield. **Generators should be considered every time you deal with a function that returns a sequence or works in a loop.**

```
>>> def fibonacci():
...   a, b = 0, 1
...   while True:
...     yield b
...     a, b = b, a + b
>>> [fib.next() for i in range(10)]
[3, 5, 8, 13, 21, 34, 55, 89, 144, 233]
```

4. Magic methods

Reference: `http://www.rafekettler.com/magicmethods.html`

5. Design Patterns[7]

- **Singleton**: restricts instantiation of a class to one object.

```
>>> class Singleton(object):
...   def __new__(cls, *args, **kw):
...     if not hasattr(cls, '_instance'):
...       orig = super(Singleton, cls)
...       cls._instance = orig.__new__(cls, *args,
    **kw)
...     return cls._instance
...
>>> class MyClass(Singleton):
...   a = 1
...
>>> one = MyClass()
>>> two = MyClass()
>>> two.a = 3
>>> one.a
3
```

- **Adapter**: wraps a class or an object A so that it works in a context intended for a class or an object B.

```
>>> from os.path import split, splitext
>>> class DublinCoreAdapter(object):
...   def __init__(self, filename):
...     self._filename = filename
...   def title(self):
```

[5] `https://docs.python.org/2/library/`
[6] Tarek Zidae - Expert Python Programming
[7] Tarek Zidae - Expert Python Programming - Chapter 14

```
...       return splitext(split(self._filename)[-1])
      [0]
...     def creator(self):
...       return 'Unknown' # we could get it for real
...     def languages(self):
...     return ('en',)
>>> class DublinCoreInfo(object):
...     def summary(self, dc_ob):
...       print 'Title: %s' % dc_ob.title()
...       print 'Creator: %s' % dc_ob.creator()
...       print 'Languages: %s' % \
...         ', '.join(dc_ob.languages())
...
>>> adapted = DublinCoreAdapter('example.txt')
>>> infos = DublinCoreInfo()
>>> infos.summary(adapted)
Title: example
Creator: Unknown
Languages: en
```

- **Facade**: provides a high-level, simpler access to a subsystem. Facade is usually done on existing systems, where a package's frequent usage is synthesized in high-level functions. Usually, no classes are needed to provide such a pattern and simple functions in the _init_.py module are sufficient.

- **Observer**: This is used to notify a list of objects with a state change.

```
>>> class Event(object):
...     _observers = []
...     def __init__(self, subject):
...       self.subject = subject
...
...     @classmethod
...     def register(cls, observer):
...       if observer not in cls._observers:
...         cls._observers.append(observer)
...
...     @classmethod
...     def unregister(cls, observer):
...       if observer in cls._observers:
...         self._observers.remove(observer)
...
...     @classmethod
...     def notify(cls, subject):
...       event = cls(subject)
...       for observer in cls._observers:
...         observer(event)
>>> class WriteEvent(Event):
...     def __repr__(self):
...       return 'WriteEvent'
...
>>> def log(event):
...   print '%s was written' % event.subject
...
>>> WriteEvent.register(log)
>>> class AnotherObserver(object):
...     def __call__(self, event):
...       print 'Yeah %s told me !' % event
...
>>> WriteEvent.register(AnotherObserver())
>>> WriteEvent.notify('a given file')
a given file was written
Yeah WriteEvent told me !
```

- **Visitor**: helps in separating algorithms from data structures

```
>>> class Printer(object):
...     def visit_list(self, ob):
...       print 'list content :'
...       print str(ob)
...     def visit_dict(self, ob):
...       print 'dict keys: %s' % ','.join(ob.keys())
...
>>> def visit(visited, visitor):
...     cls = visited.__class__.__name__
...     meth = 'visit_%s' % cls
...     method = getattr(visitor, meth, None)
...     if meth is None:
...       meth(visited)
...
>>> visit([1, 2, 5], Printer())
list content : [1, 2, 5]
>>> visit({'one': 1, 'two': 2, 'three': 3}, Printer
    ())
dict keys: three,two,one
```

6. Advanced topics

6.1 Metaclass method[8]

```
>>> def method(self):
...   return 1
...
>>> klass = type('MyClass', (object,), {'method': method
    })
>>> instance = klass()
>>> instance.method()
1
```

7. Unit tests

- PyUnit `http://pyunit.sourceforge.net/pyunit.html`
- DocTest `https://docs.python.org/2/library/doctest.html`

8. Profiling[9]

```
python -m cProfile [-o output_file] [-s sort_order]
    myscript.py
```

Memory profiling: use Guppy [10]

9. Multiprocessing

Global Interpreter Lock (GIL) is a mechanism used in computer language interpreters to synchronize the execution of threads so that only one thread can execute at a time. An interpreter which uses GIL will always allow exactly one thread to execute at a time, even if run on a multi-core processor. Some popular interpreters that have GIL are CPython and Ruby MRI.

```
>>> from processing import Process
>>> import os
>>> def work():
...   print 'hey i am a process, id: %d' % os.getpid()
>>> ps = []
>>> for i in range(4):
... p = Process(target=work)
... ps.append(p)
... p.start()
...
hey i am a process, id: 27457
hey i am a process, id: 27458
hey i am a process, id: 27460
```

[8] Tarek Zidae - Expert Python Programming - Chapter 3
[9] `https://docs.python.org/2/library/profile.html`
[10] `http://guppy-pe.sourceforge.net/`

10. Useful Tools

- **Pylint**, a very flexible source code analyzer
- **CloneDigger**, a duplicate code detection tool

11. Books

11.1 Read:

- Tarek Zidae - Expert Python Programming

11.2 To read:

- Code Like a Pythonista: Idiomatic Python [11]
- Fluent Python - Luciano Ramalho

[11] http://python.net/~goodger/projects/pycon/2007/idiomatic/handout.html#don-t-reinvent-the-wheel