More on Python

Isak Karlsson
isak-kar@dsv.su.se

Table of contents

Python Properties

- Dynamic, multi-paradigm (oo, imp, func, proc)
- Focus on clear syntax and “one-best-way” philosophy
- Very easy to learn(?)
- Large STD lib
- Available on all major platforms

Python Properties, cont'd

- Integrates with COM (legacy), .NET, Java
- Extensions may be written in C or C++

Python Properties, cont'd

- Parse tree (code) availible at runtime

```
import ast
class RewriteLookup(ast.NodeTransformer):
    def __init__(self, occ, rep):
        self.occ = occ; self.rep = rep
    def visit_Name(self, node):
        if self.occ = node.id:
            return ast.copy_location(ast.Name(id=self.rep,
                            ctx=node.ctx), node)
        else:
            return node

>> ast.dump(ast.parse("foo = 2"))
"Module(body=[Assign(targets=[Name(id='foo', ctx=Store())], value=Num(n=2))])"
>> tree = RewriteLookup("foo", "bar").visit(ast.parse("foo = 2"))
>> ast.dump(tree)
"Module(body=[Assign(targets=[Name(id='bar', ctx=Store())], value=Num(n=2))])"
>> eval(tree)
```

MetaMetaMeta

Who uses it?

- Google, NASA, YouTube
- CERN, Spotify, AstraZeneca
- EVE (online)
- Gimp, Canonical
- Autodesk Maya, Blender
- Etc...Pytho

Py2k vs. Py3k

- Python 3 – released february 13th 2009
- Not backward compatible
- Much legacy code incompatible with 3k (but there exist a rewrite tool)
- Current 2.7.1, and 3.2 (we'll be using 2.7.1 since assignment 1 is done using Jython)

Python syntax

```
module ::= stmt, {stmt}
stmt ::= funcdef | classdef | assign |
      print | for | while | if | with |
      raise | tryexcept | tryfinally |
      import | import_from | exec | expr |
      Pass
expr ::= boolop | binop | unaryop | lamda |
         ifexpr | dict | set | list | list-
         comp | set-comp | generator |
         yield | compare | call | repr |
         num | str | attr | subscript |
         name | tuple
```

MODULES

Beyond the interpreter

Lunar Module

Modules

- Module == source file == text file ending with .py
- Imported using `import` or `from .. import .. as ..` statements

Example module

```
# dog.py
x = "internal"
print "Importing dog."

def bark():
    print "Dog is barking."
```

```
# animals.py
import dog
print "Welcome dog."
dog.bark()
```

```
$: python animals.py
Importing dog.
Welcome dog.
Dog is barking.
```

Modules, cont'd

- As seen;
- Code at the top most a module is run on import. Why?
- `import dog` eqvivalent to `dog = __import__("dog")`, Why?
- Import can apper as any statement, and adhere to the current scope.

Modules, cont'd

```
# dog.py
x = "internal"
print "Importing dog."

def bark():
     print "Dog is barking."
```

```
>> import animals
What? Yes
Importing dog
Dog is barking.
```

```
# animals.py

t = input("What? ")
if t == "yes":
     import dog
     print "Welcome dog."
     dog.bark()
```

```
>> import animals
What? No
```

Module guards

```
# dog.py

if __name__ == "__main__":
     print "Running dog."

def bark():
     print "Dog is barking."
```

```
# animals.py
import dog
print "Welcome dog."
dog.bark()
```

```
$: python animals.py
Welcome dog.
Dog is barking.
$: python dog.py
Running dog.
```

Modules as test-cases

- Good idea: use this for test cases.

```
def square(x): return x * x

if __name__ == "__main__":
     import unittest
     class TestModule(unittest.TestCase):
          def test_square(self):
               self.assert_equal(square(2), 4)
     unittest.main()
```

Functions

$$\dot{X}(t) = -\frac{v}{B_0}\sum_j \sum_{n=-\infty}^{\infty} \int d^3k \exp\{in[\psi - \phi(t)] + i(k_\parallel v_\parallel - \omega_j)t\} \times \left[J_n(W)\left(a\frac{k_x}{k_\perp}\delta B_\parallel^j + b\,\delta B_x^j\right) - i\sqrt{1-\mu^2}\frac{k_y}{k_\perp}\delta B_\parallel^j J_n'(W)\right], \quad (15)$$

$$\dot{Y}(t) = -\frac{v}{B_0}\sum_{j=1}^{N} \sum_{n=-\infty}^{\infty} \int d^3k \exp\{in[\psi - \phi(t)] + i(k_\parallel v_\parallel - \omega_j)t\} \times \left[J_n(W)\left(a\frac{k_y}{k_\perp}\delta B_\parallel^j + b\,\delta B_y^j\right) + i\sqrt{1-\mu^2}\frac{k_x}{k_\perp}\delta B_\parallel^j J_n'(W)\right], \quad (16)$$

Quasi-linear Drift of Cosmic Rays in Weak Turbulent Electromagnetic Fields

Positional, keyword and default arguments

```
def f(a, b, c="c"):
    print a, b, c

>> f(b="b", a="a")
abc
```

Variable argument lists

```
def f(a, *bs):
    for b in bs: print a, b

>> f("a", "a", "b", "c")
aa
ab
ac
```

Dictionaries as arguments

```
def f(**bs):
    for key,value in bs.iteritems():
        print key,value

>> f(a="a", b="b")
aa
bb
```

NOTE: iteritems() is items() in Py3K

Unpacking arguments

```
def f(a,b):
    print a, b

>> f(*["a", "b"])
ab
```

Functions are objects

```
def f(a,b):
     print a, b

>> f
<function f at 0x7f6a053a0668>
>> c = f
>> c(*[1, 2])
12
```

Functions are objects

```
def square(a):
     return a * a

>> s = map(square, [2, 4])
>> s
[4, 16]
```

```
>> s = map(lambda x: x * x, [2, 4])
>> s
[4, 16]
```

Lambdas

- A `lambda` is a anonymus function
  - Consisting of an expression, _only_
  - So, no statements
- For more complex, use `def`
  - `def` is a statement and is followed by a list of statements, i.e. it's possible to nest them

Lambdas & Higher order functions

There are no switch statement – let's make one.

```
def square(a):
     return a * a

def switch(str, *args, **map):
     if map.has_key(str):
          return map[str](*args)

>> switch("f", 10,
           f = square,
           l = lambda x: x*x)
100
```

List comprehensions

```
def even(*numbers):
    """ Return even number from numbers """
    return [n for n in numbers if n % 2 == 0]
>> even(*range(1, 10))
```

List comprehensions

- **The Proposed Solution**
  - It is proposed to allow conditional construction of list literals using for and if clauses. They would nest in the same way for loops and if statements nest now.
- **Rationale**
  - List comprehensions provide a more concise way to create lists in situations where map() and filter() and/or nested loops would currently be used.

http://www.python.org/dev/peps/pep-0202/

List comprehensions

- **Syntax**
  - `[x,y for ...]` is disallowed
  - `[(x,y) for ...]` instead
  - `[... for x ... for y]` y – varies faster

```
>> xlist = [1,2,3]
>> ylist = [1,2,3]
>> [x + y for x in xlist for y in ylist]
[2, 3, 4, 3, 4, 5, 4, 5, 6]
```

http://www.python.org/dev/peps/pep-0202/

List comprehensions

Map and filter.

```
def is_even(n): return n % 2 == 0

def even(*numbers):
    """ Return even number from numbers """
    return list(filter(is_even, number))

>> even(*range(1, 10))
[2, 4, 6, 8, 10]

def square(*numbers):
    """ Return squared number from numbers """
    return list(map(lambda x: x * x, numbers))

>> square(*range(1, 10))
[2, 4, 9, 16, 25, 36, 49, 64, 82, 100]
```

Set comprehensions

Same, same, same... but {}, and obviously no duplicates

```
def is_even(n): return n % 2 == 0

>> {x % 2 == 0 for x in range(1, 10)}
set([False, True])

>> {is_even(x) for x in range(1, 10)}
set([False, True])
```

Magic methods

```
obj.__init__, obj.__add__, obj.__repr__, obj.__str__,
obj.__eq__, obj.__lt__, etc...
```

```
class n(object):
    def __init__(self, i):
        self.i = i
    def __add__(self, o):
        return self.i + o
    def __int__(self):
        return self.i
    def __str__(self):
        return str(i)

>> n = n(10)
>> int(n + n(10))
20
>> str(n)
"10"
>> int(n)
10
```

Example: `__call__`

```
square = lambda x: x * x

>> square(10)
>> square.__call__(10)
```

```
def square(x):
    return x * x

>> square(10)
>> square.__call__(10)
100
```

```
class squarer(object):
    def __call__(self, x):
        return x * x

>> square = squarer()
>> square(10)
>> square.__call__(10)
```

"Calling" is implemented using a "magic" function, as well as every expression operation

Example: `__add/mul__`

```
>> 10 + 10
20
>> 10 * 10
100
```

```
>> "isak" + "isak"
"isakisak"
>> "isak" * 3
"isakisakisak"
```

```
class num(object):
    def __init__(self, v):
        self.v = v
    def __add__(self, o):
        return num(self.v + o.v)
    def __mul__(self, o):
        return num(self.v * o.v)
    def __str__(self):
        return str(self.v)


>> num(10) + num(10)
'20'
>> num(10) * num(10)
'100'
```

Plus, minus, times, modulo, etc...

Example: `__contains__`

```
>> 10 in [1,2,10]
True
```

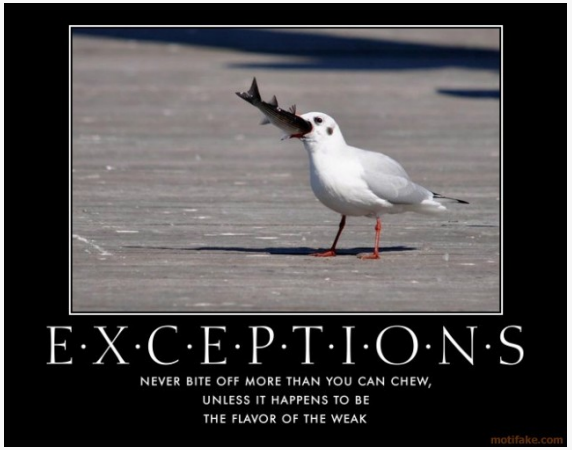
```
>> "isak" in {"a", "isak"}
True
```

```
class lst(object):
    def __init__(self, *args):
        self.l = args
    def __contains__(self, a):
        return a in self.l


>> 10 in lst(10, 20, 30)
True
```

Etc... etc... etc...

Exceptions



Exceptions

```
def print_list(lst, index):
    try:
        element = lst[index]
    except IndexError as e:
        print e
    else:
        print element

>> lst = range(1,10)
>> print_list(lst, 3)
4
>> print_list(lst, 100)
list index out of range
```

Exceptions

```
def open_close():
    try:
        file = open("file-that-dont-exist", "r")
    except IOError as e:
        print e
        raise e
    finally:
        print "closing..."
        if file is not None:
            file.close()
>> open_close()
No such file or directory: 'file-that-dont-exist'
"closing..."
Traceback (most recent call last):
  File "<stdin>", line 2, in <module>
IOError: [Errno 2] No such file or
directory: 'file-that-dont-exist'
```

Namespaces

Namespaces

- A namespace is a set of mappings between a name and an object
- In python this is implemented as dictionaries where the key is the variable and the value, is the value that the variable refers to
- Therefore namspaces can be used as dictionaries and vice versa

Namespaces

- When a value for a variable is asked for the interpreter searches:
  - Local namespace(s)
    - The inner most nesting level first
  - Global namespace
  - Builtin namespace
  - Else: a `NameError` is raised

Namespaces

- Assignment to a local variable (nested) introduces a new name in the local scope
- To make a variable looked up in the global scope `global` can be used

Namespaces

Making the local global

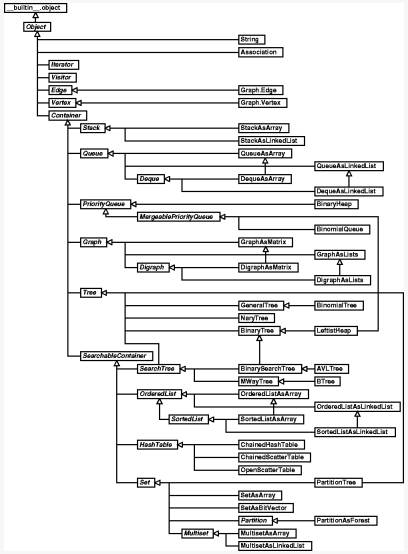
```
x = 10
def change_x():
    x = 2

>> change_x()
>> print x
10
```

```
x = 10
def change_x():
    global x # the global keyword
             # let a varible be looked up
             # in the global scope
    x = 20

>> change_x()
>> print x
20
```

Classes – instances and inheritance



Classes – background

- Prior to Python 2.2, user defined types where inherently different from built-in types (e.g. list, int etc.)
- You could not inherit from built-in types
- Python 2.2. was a first step towards type-class unification

Classes – background

- User defined types pre 2.2 are generally refered to as “classic classes”, “classic types” or plainly “old-style classes”
- In Python 2.2 “new-style classes” where introduced
- Old-style classes and new-style classes coexist in Python 2.2–2.7, but are removed in Python 3.0+

Classes – background

- A classic class is a class that is defined with a class statement that has no new style classes amongst its base classes, i.e. inherit from no other class, or only from classic classes, applying the definition recursively.
- A new-style class has the built-in **`object`** class as one of its base classes.

Classes – background

- A classic class is a class that is defined with a class statement that has no new style classes amongst its base classes, i.e. inherit from no other class, or only from classic classes, applying the definition recursively.
- A new-style class has the built-in **`object`** class as one of its base classes.

Classes – background

Remember: These definitions are equivalent in Py3.0+

Old-style classes

```
class A:
    <statment-1>
    ...
    ...
    <statement-n>
```

```
class A:
    pass

>> dir(A)
['__doc__', '__module__']
```

New-style classes

```
class A(object):
    <statment-1>
    ...
    ...
    <statement-n>
```

```
class A(object):
    pass

>> dir(A)
['__class__', '__delattr__', '__dict__', '__doc__',
'__format__', '__getattribute__', '__hash__',
'__init__', '__module__', '__new__', '__reduce__',
'__reduce_ex__', '__repr__', '__setattr__',
'__sizeof__', '__str__', '__subclasshook__',
'__weakref__']
```

Defining classes

- When entering a class definition, Pyton will introduce a new scope for that class.
- Attributes and function definitions will bind to this new namespace

```
class A(object):
    a = 10

>> A.a
10
```

Defining classes

```
class A(object):
     def __init__(self, name):
          self.name = name
     def hi(self):
          print self.name, " say hi!"

>> A.__init__
<unbound method A.__init__>
>> a = A("test")
>> a.__init__
<bound method A.__init__ of <__main__.A object at 0x7f1d1b7a4ed0>>
>> a.hi()
test say hi!
>> A.hi(a) # equivalent
test say hi!
```

Class objects

- When a class definition is ended (at the end of the block) a new class object is created
- This new object will be bound to the current scope (namespace)

```
# in file as.py
class A(object):
     def __init__(self, name):
          self.name = name
     def hi(self):
          print self.name, " say hi!"
>> import as
>> as.A.__module__
'as'
```

Class objects

- Basically, you can do two things with a class object:
  - Reference attributes (as a namespace)
  - Create instances

```
# in file as.py
class A(object):
     def __init__(self, name):
          self.name = name
     def hi(self):
          print self.name, " say hi!"
>> import as
>> as.A.__module__
'as'
```

Class objects

- Instances of a class can appear to be funtions

```
class A(object):
     def __init__(self, name):
          self.name = name
     def __call__(self):
          print self.name, " say hi!"

>> a = A("A")
>> a()
A say hi!
```

Class objects

- Instances of a class can appear to be arrays

```
class A(object):
     def __init__(self, name):
          self.name = name
     def __getitem__(self, val):
          print self.name, " say hi! With ", val

>> a = A("A")
>> a[10]
A say hi! With 10
```

And....

As we saw, methods are objects

```
class A(object):
     def __init__(self, name):
          self.name = name
     def hi(self):
          print self.name, " say hi!"

>> A.__init__
<unbound method A.__init__>
>> a = A("test")
>> a.__init__
<bound method A.__init__ of <__main__.A object at 0x7f1d1b7a4ed0>>
>> a.hi()
test say hi!
>> A.hi(a) # equivalent
test say hi!
```

Descriptors

- If an attribute is defined as a descriptor, it overrides access behaviour by methods in the descriptor protocol

```
descr.__get__(self, obj, type=None) ---> value
descr.__set__(self, obj, value) ---> None
descr.__delete__(self, obj, value) ---> None
```

Descriptors

- Simplified; a descriptor makes `x.a` do something else than just a dictionary lookup, i.e. `x.__dict__('a')`
- If `x` is a descriptor (i.e. implements the methods), Python will invoke the appropriate accessor method on the attribute
  - i.e. `x.__get__('a')` instead

```
class NameDescr(object):
     def __init__(self, name):
          self.name = name
     def __get__(self, obj, objtype=None):
          print "Getting name!"
          return self.name
     def __set__(self, obj, value):
          print "Setting name!"
          self.name = value

class Person(object):
     name = NameDescr("Isak")

>> isak = Person()
>> isak.name
Getting name!
'Isak'
```

Descriptors

- Used for various nifty things
- Properties, bound and unbound methods, super(), static methods, class methods
  - Class methods – class methods that can be overriden
  - Static methods – class methods that know nothing about the class or intances (use a class as a namespace)

Properties

- Build a data descriptor that invokes the related function when accessing the attribute (builds on the descriptor protocol)
  - `proprety(fget=None, fset=None, fdel=None fdoc=None)`

Properties

```
class Person(objecvt):
     def __init__(self, name)
          self._name = name
class Person(object):
    def __init__(self, name):
        self._name = name

    def set_name(self, name):
        if len(name.strip()) < 1:
            raise Error("To short name")
        self._name = name
    def get_name(self):
        return self._name

    name = property(fset=set_name, fget=get_name)


>> isak = Person()
>> isak.name = ""
Error("To short name")
```

Static methods

```
class Person(object):
     def __init__(self, name):
          self.name = name

     def make(name):
          return Person(name)

     make = staticmethod(make)

>> p = Person.make("Isak")
>> p.name
"Isak"
```

Equvialent

```
class Person(object):
     def __init__(self, name):
          self.name = name

     @staticmethod
     def make(name):
          return Person(name)

>> p = Person.make("Isak")
>> p.name
"Isak"
```

Class methods

```
class Person(object):
     def __init__(self, name):
          self.name = name

     def make(self, name):
          return self(name)

     make = classmethod(make)

>> p = Person.make("Isak")
>> p.name
"Isak"
```

Equvialent

```
class Person(object):
     def __init__(self, name):
          self.name = name

     @classmethod
     def make(self, name):
          return self(name)

>> p = Person.make("Isak")
>> p.name
"Isak"
```

Class method, cont'd

```
class Person(object):
     def __init__(self, name):
          self.name = name

     @classmethod
     def make(self, name):
          return self(name)

class Student(Person):
     @classmethod
     def make(self, name):
          s = self(name)
          s.degree = "Unknown"
          return s

>> s = Student.make("isak")
>> s.degree
"Unkown"
```

Fun with classes

```
class Thing(object):
     def __init__(self, on=False):
          self._on = on
     def on(self): return self._on
     def turn_on(self): self._on = True
     def turn_off(self): self._on = False
     def __str__(self):
          return "<%s> is %s" %
               (self.__class__.__name__,
                "On" if self._on else "Off")

>> t = Thing()
>> print t
Thing is Off
```

Fun with classes

- Each object has a `__class__` attribute that is assignable like any attr.

Fun with classes

- ... so, let's make things interesting.

Fun with classes

```
class OnThing(object):
     def on(self): return True
     def toggle(self): self.__class__ = OffThing
```

Fun with classes

```
class OffThing(object):
     def on(self): return False
     def toggle(self): self.__class__ = OnThing
```

Fun with classes

```
class OffThing(object):
     def on(self): return False
     def toggle(self):
          self.__class__ = OnThing
```

```
class OnThing(object):
     def on(self): return True
     def toggle(self):
          self.__class__ = OffThing
```

```
class Thing(object):
     def __init__(self, on=False):
          self.__class__ = OnThing if on else OffThing

     def __str__(self):
          return "<%s> is %s" %
               (self.__class__.__name__,
                "On" if self.on() else "Off")

>> t = Thing()
>> print t
OffThing is Off
>> t.toggle()
>> print t
OnThing is On
```

The End

```
from time import time
time = datetime.now()
if time.hour in range(13, 15)
  do_code_example()
present_references()
```

The End

- Old slides by Beatrice
- "Programming python" byt Mark Lutz
- python.org