CS 5142
Scripting Languages
11/08/2013
Python
Outline

• Comprehensions, Generators
• Modules
• Decorators
• Functools
• Django
class MyIter:
    def __init__(self):
        self.curr = -1
        self.end = 5

    def __iter__(self):
        return self

    def next(self):
        if self.curr >= self.end:
            raise StopIteration
        else:
            self.curr += 1
            return self.curr

for i in MyIter():
    print i
List Comprehensions

- Concise syntax for generating lists:
  
  \[
  \text{listCompr} ::= [\text{expr forClause comprClause}*]
  \]

  \[
  \text{forClause} ::= \text{for id in expr}
  \]

  \[
  \text{comprClause} ::= \text{forClause} | \text{ifClause}
  \]

  \[
  \text{ifClause} ::= \text{if expr}
  \]

- Example:
  
  \[
  l = [1,2,3,4]
  \]

  \[
  t = 'a', 'b'
  \]

  \[
  c1 = [x \text{ for } x \text{ in } l \text{ if } x \% 2 == 0]
  \]

  \[
  c2 = [(x, y) \text{ for } x \text{ in } l \text{ if } x < 3 \text{ for } y \text{ in } t]
  \]

  \[
  \text{print } \text{str(c1)} \ # \ [2, 4]
  \]

  \[
  \text{print } \text{str(c2)} \ # \ [(1,'a'),(1,'b'),(2,'a'),(2,'b')]
  \]
Generators

```python
#!/usr/bin/env python

def myGenerator(x):
    x = x + 3
    yield x
    x = x + 3
    yield x
    x = x + 3
    yield x

myCoroutine = myGenerator(1)
print '1st call:'
print myCoroutine.next()
print '2nd call:'
print myCoroutine.next()
print '3rd call:'
print myCoroutine.next()
print 'after 3rd call'

Python can also treat a generator result as an iterator:

```
Generator Expressions

• Creates an anonymous generator function

\[
\text{listCompr} ::= (\text{expr forClause comprClause}*)
\]
\[
\text{forClause} ::= \text{for id in expr}
\]
\[
\text{comprClause} ::= \text{forClause | ifClause}
\]
\[
\text{ifClause} ::= \text{if expr}
\]

• Example:

```python
def gen(l):
    for x in l:
        if (x % 2 == 0):
            yield x

g = gen(l)
```

```python
l = [1, 2, 3, 4]
g = (x for x in l if x % 2 == 0)
print str(g.next()) # 2
```
### Using Modules

<table>
<thead>
<tr>
<th>Import Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>import M</code></td>
<td>Import M. Refer to things defined in M with <code>M.name</code>.</td>
</tr>
<tr>
<td><code>from M import *</code></td>
<td>Imports M, creates reference to all public objects in the current namespace. Refer to things with <code>name</code>.</td>
</tr>
<tr>
<td><code>from M import name</code></td>
<td>Imports M, creates reference to name in the current namespace. Refer to it with <code>name</code>.</td>
</tr>
</tbody>
</table>
class Fruit:
    def __init__(self, weight):
        self.weight = weight
    def pluck(self):
        return 'fruit(' + self.weight + 'g)'
    def prepare(self, how):
        return how + 'd ' + self.pluck()

import fruit
f = fruit.Fruit(150)
print f.prepare('squeeze')
def square(x):
    return x * x

def main():
    print "in the main"

if __name__ == '__main__':
    main()

• Use __name__ variable to test how the code is being used
  Will either be the name of the module, or “__main__”
Decorator Pattern

- See “Gang of Four” design patterns book
- Add/remove responsibilities to objects at runtime
- Avoid feature clutter high in the hierarchy
#!/usr/bin/env python

class dec(object):
    def __init__(self, f):
        print "dec.__init__()"
        print "inside g()"
        print "Finished decorating g()"
        print "dec.__call__()"

    def __call__(self):
        print "dec.__call__()"

@dec
def g():
    print "inside g()"

print "Finished decorating g()"
g()
Decorators for Debugging

#!/usr/bin/env python

class trace(object):
    def __init__(self, f):
        self.f = f
    def __call__(self):
        print "called", self.f.__name__
        self.f()
        print "exited", self.f.__name__

@trace
def g():
    print "inside g()"

@trace
def h():
    print "inside h()"

g()
h()
#!/usr/bin/env python

from functools import partial

def sum(x, y):
    return x + y

incr = partial(sum, 1)
print incr(3)  # 4

- Provides higher-order functions and operations
- Makes programming in Python similar to programming in functional languages
$ pip install Django

$ python -c "import django; print(django.get_version())"

$ django-admin.py startproject mysite

$ python manage.py runserver
$ pip install Django

$ python -c "import django; print(django.get_version())"

$ django-admin.py startproject mysite

$ python manage.py runserver
Django

$ cat mysite/views.py
from django.http import HttpResponse
def index(request):
    return HttpResponse("Hello, world.")

$ cat mysite/urls.py
from django.conf.urls import patterns, include, url
from django.contrib import admin
admin.autodiscover()

urlpatterns = patterns('',
    url(r'^hello/', 'mysite.views.index')
)
Last Slide

• Office hours moved to Wednesday.

• Today’s lecture
  – Python

• Next lecture
  – TouchDevel