CS 5142
Scripting Languages

11/08/2013

Python

Outline

- Comprehensions, Generators
- Modules
- Decorators
- Functools
- Django

Iterator Object

```
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
```

- __iter__() returns the iterator object
- next() returns the next item

Concepts

List Comprehensions

- Concise syntax for generating lists:

  listCompr ::= [*expr forClause comprClause**]
  forClause ::= **`for`** *id* **`in`** *expr*
  comprClause ::= *forClause* | *ifClause*
  ifClause ::= **`if`** *expr*

- Example:

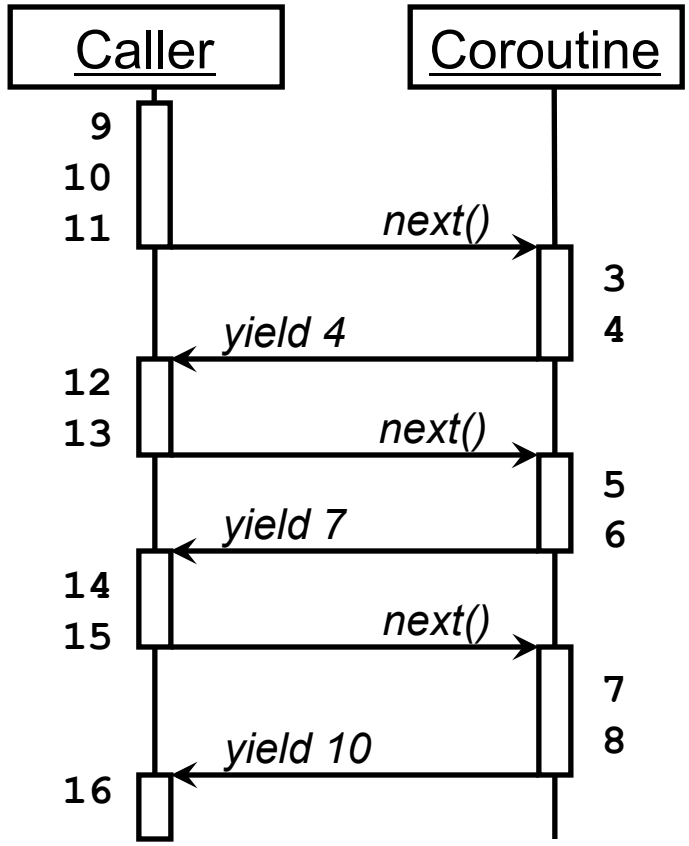
```
l = [1,2,3,4]
t = 'a', 'b'
c1 = [x for x in l if x % 2 == 0]
c2 = [(x, y) for x in l if x < 3 for y in t]
print str(c1) # [2, 4]
print str(c2) # [(1,'a'),(1,'b'),(2,'a'),(2,'b')]
```

```
c2 = []
for x in l:
  if x < 3:
    for y in t:
      c2.append((x,y))
```

Python

Generators

```
#!/usr/bin/env python          # 1
def myGenerator(x):            # 2
    x = x + 3                  # 3
    yield x                    # 4
    x = x + 3                  # 5
    yield x                    # 6
    x = x + 3                  # 7
    yield x                    # 8
myCoroutine = myGenerator(1)   # 9
print '1st call:'             #10
print myCoroutine.next()       #11
print '2nd call:'             #12
print myCoroutine.next()       #13
print '3rd call:'             #14
print myCoroutine.next()       #15
print 'after 3rd call'        #16
```



Python can also treat a generator result as an iterator:

`for y in myGenerator(1): print y`

Concepts

Generator Expressions

- Creates an anonymous generator function

  listCompr ::= (*expr* *forClause* *comprClause**)

  forClause ::= **`for`** *id* **`in`** *expr*

  comprClause ::= *forClause* | *ifClause*

  ifClause ::= **`if`** *expr*

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

- Example:

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

Using Modules

`import M`	Import M. Refer to things defined in M with *M.name*.
`from M import *`	Imports M, creates reference to all public objects in the current namespace. Refer to things with *name.*
`from M import name`	Imports M, creates reference to name in the current namespace. Refer to it with *name.*

Python

Defining Modules

```
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')
```

Modules or Standalone?

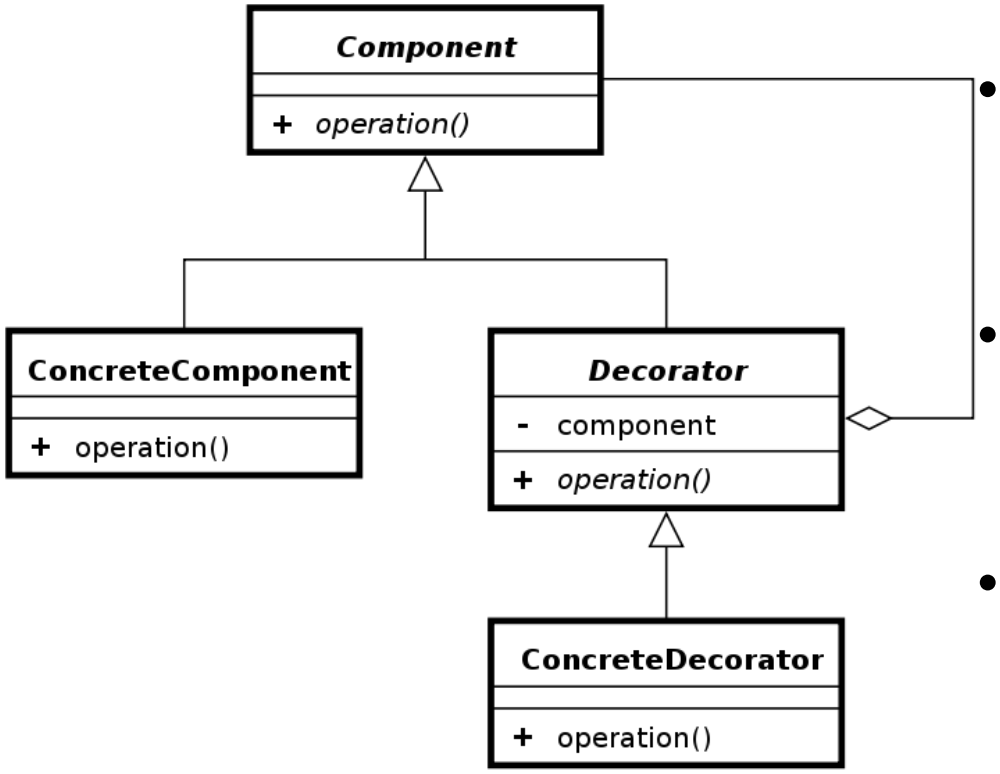
```
def square(x):
    return x * x

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

Python Decorators

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

class dec(object):
    def __init__(self, f):
        print "dec.__init__()”

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

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

print "Finished decorating g()"
g()
```

```
 ./decorator.py
dec.__init__()
Finished decorating g()
dec.__call__()
```

Python

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()
```

Functools

```
#!/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

Django

```
$ pip install Django

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

$ django-admin.py startproject mysite

$ python manage.py runserver
```

Django

```
$ pip install Django

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

$ django-admin.py startproject mysite

$ python manage.py runserver
```

Python

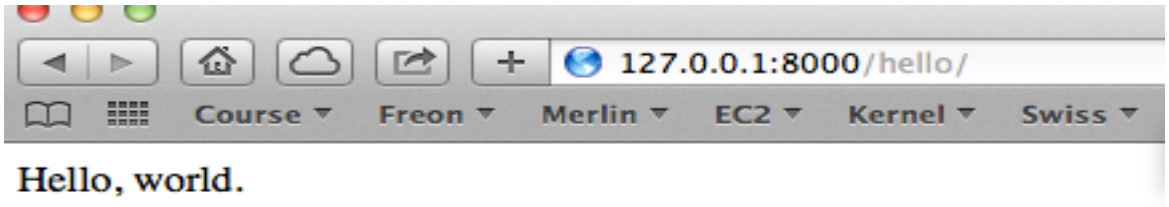
Django

```
$ 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