Functions and Functional Programming

CS 339R (Python) — Chapter 6
Outline

- Function Parameters
- Scope
- Higher-Order, Nested, and Anonymous Functions
- Decorators
- Functional Programming
- Generators Redux
- `eval` and `exec`
Function Parameters

- Pass by "Assignment of Object Reference"
  - Like Java and C#

- Default Arguments

- Keyword Arguments
  - parameter names act as keywords in the function call

- Variable-length Arguments Lists
  - accomplished with tuples and/or dictionaries
Arguments are **Assigned** toParms

- That is, the *object reference* in the calling argument is *assigned* to the local parameter

- For *immutable* objects, this is similar to *pass-by-value*
  - Because no local parameter changes can propagate back to the caller

- For *mutable* objects, this is similar to *passing objects* in Java and C#
  - Applies to lists, dictionaries, class instances, etc.
  - You can change an object’s attributes, as usual

- See *args0.py*
Default Arguments

* Like in C++

* Default args should come after non-default ("positional") args

* The default value is stored when the def statement is first processed!
  * So it should usually be immutable

* See next slide for a “gotcha”

```python
gotcha_code
```
Beware Mutable Default Args

```python
def h(x, y = []):
    y.append(x)
    return y

print h(1)  # [1]
print h(2)  # [1, 2]
print h(3, [7])  # [7, 3]
print h(4)  # [1, 2, 4]
```
Passing Arguments by Keyword

- You can use the actual parameter names as keywords in the calling context.
- Allows ignoring the order the parameters appear in the function.
- Useful with large arg lists that have default args.

```python
>>> def f(a,b):
...     print 'a =',a,'b =',b
...     print '...

>>> f(b=10,a=20)
a = 20 b = 10
```
The sorted Function

* Often called with keyword arguments:
  
  ```python
  sorted(iterable, cmp=None, key=None, reverse=False)
  ```

* To sort in descending order:
  
  ```python
  sorted(stuff, reverse=True)
  ```

* From Homework 2:
  
  ```python
  for w, c in sorted(counts.iteritems(), key=operator.itemgetter(1), reverse=True):
  ```
Variable-length Argument Lists

* An arbitrary number of *positional* arguments can be collected into a *tuple*

* Just use an *asterisk* with the tuple *parameter*

```python
>>> def avg(*stuff):
...     return sum(stuff)/float(len(stuff))
...     return sum(stuff)/float(len(stuff))
...
>>> avg(1,2,3)
2.0
>>> avg(1,2,3,4)
2.5
```
Implementing a vararg max Function

# mymax.py: Illustrates variable-length arg lists

def mymax(*stuff):
    themax = None
    for x in stuff:
        if x > themax:
            themax = x
    return themax

print mymax()       # None
print mymax(1,2,3)  # 3
Interlude: and, or, not and None

- Logical connectives can apply to almost any value
- Empty and zero values are False
- Non-empty and non-zero are True
- This affects compound logical expressions
- None is False and less than anything
- See connectives.py
Unpacking Tuples at the Call Site

- Use an asterisk in the function call

  - Note: this only works in the context of a function call

```python
>>> def f(a,b,c):
...     print 'a =',a,'b =',b,'c =',c
...
>>> stuff = ('one',2,333)
>>> f(*stuff)
a = one b = 2 c = 333
```
Variable-length Keyword Arg Lists

* Received as a dictionary

* Use ** with the receiving parameter

```python
>>> def f(**kwargs):
    ...     print kwargs
    ...
...  
>>> f(a=1, b='two', c=[3])
{'a': 1, 'c': [3], 'b': 'two'}
```
Unpacking Pairs at the Call Site

* Use ** in the function call

* The key-value pairs in the dictionary become keyword arguments in the call

  * They must use the same names as expected by the function

```python
>>> def f(a,b,c):
...     print 'a =',a,'b =',b,'c =',c
... >>>

>>> args = {'a':'one', 'b':2, 'c':[3]}
>>> f(**args)  # Same as f(a='one',b=2,c=[3])
a = one b = 2 c = [3]
```
Putting it All Together

* Don’t mix parameter-passing styles unless necessary

* Order is important:
  * Fixed
  * Default
  * Optional non-keyword
  * Keyword

* See args2.py
A Very Flexible Function

# allargs.py: Accepts arbitrary number of positional and/or keyword args

def f(*args, **kwargs):
    for arg in args:
        print arg
    for key in kwargs:
        print key, '=', kwargs[key]

f(1,2,t=3,f=4)

# Output:
1
2
t = 3
f = 4
Scope

- Scope refers to the *region* where an identifier is *accessible*
  - Affects how values bound to names are found
- Python manages scopes with the concept of *namespaces*
  - A namespace is just a *dictionary* attached to a scope
  - Named `__dict__`, available through `vars()`
- The following constitute a scope: *modules, functions, classes, objects*
- See `scope.py`
The LEGB Rule

• Names are resolved in the following order:
  • The Local scope (`vars()`)  
  • Enclosing scopes (could be a function, class, or superclass (recursively))  
  • The Global (current module) scope  
  • Built-ins (shared by all modules)

• The dictionary of each namespace, in the above order, is searched until the name of the identifier is found
  • Then the associated value is returned

• But *changing* non-local bindings is another matter...
The global Statement

* Allows assignment to a global variable
  * instead of creating a new local variable as it normally would
* Whether a name is global or local is determined at *definition* time

```python
>>> a = 10
>>> def f():
...     global a  # Required!
...     a += 1
...
>>> print a
10
>>> f()
>>> print a
11
```
Changing Non-Locals (Python 2)

- You can’t!
- Just globals, with the `global` keyword
Changing Non-locals in Python 3

```python
# nonlocal3.py

def g():
    a = 10
    def f():
        nonlocal a
        a += 1  # Changes g's a
        print(a)  # 10
        f()
    print(a)  # 11

    g()
```
Decorators

• A way of adding functionality to an existing function without changing the function
  
  • “Before/after” logic

• Merely wrap the original function in another function (or any callable)

• The wrapper should be quite generic
  
  • To account for various types of argument passing (*args,**kwargs)

• Code example on the next slide
Decorator code

```python
def trace(func):
    if enable_tracing:
        def callf(*args, **kwargs):
            print func.__name__, args, kwargs
            r = func(*args, **kwargs)
            print "returned", r
            return r
        return callf
    else:
        return func
```

**new syntax**

```python
@trace
def square(x):
    return x * x
```

**old-fashioned way**

```python
def square(x):
    return x * x
square = trace(square)
```

**output:**

```
square(3)  calls callf, which calls the original square function
```

* See: `decorate1.py` (old-fashioned way), `decorate2.py` (new syntax), `decorate3.py` (composing decorators)
Higher-order Functions

* A fancy term for functions that have other functions as parameters or return values

* Possible, because in Python, functions are real objects

* Functions passed as parameters can customize the host function
  * Like a custom compare function passed to `sort()`
  * Hence, `sort` is a higher-level function

* See next slide...
Customizing list.sort()

```python
emps = [('Joe',100,2500),('Jill',200,1200)]

# Sort by product of last two fields, ascending order
def cmp1(rec1,rec2):
emps.sort(cmp1)
print emps

# Sort by product of last two fields, descending order
def cmp2(rec1,rec2):
emps.sort(cmp2)
print emps

''' Output:
[('Jill', 200, 1200), ('Joe', 100, 2500)]
[('Joe', 100, 2500), ('Jill', 200, 1200)]
'''```
Anonymous Functions

- aka “lambda expressions”
- You can define a function as an expression with the `lambda` keyword
- The function itself can only compute a single expression
  - Can’t have multiple statements
  - The “return” is implied
  - See next slide...
Using lambda Expressions

```python
emps = [('Joe',100,2500),('Jill',200,1200)]

# Sort by product of last two fields, ascending order
emps.sort(lambda r1,r2: r1[1]*r1[2] - r2[1]*r2[2])
print emps

# Sort by product of last two fields, descending order
emps.sort(lambda r1,r2: r2[1]*r2[2] - r1[1]*r1[2])
print emps

''' Output:
[('Jill', 200, 1200), ('Joe', 100, 2500)]
[('Joe', 100, 2500), ('Jill', 200, 1200)]
'''
```
Creating a Custom Sorter

```python
>>> def cmpsort(yourcmp):
...     return lambda stuff: sorted(stuff, cmp=yourcmp)
... 
>>> f = cmpsort(lambda x, y: y - x)  # Descending order
>>> f([1, 2, 3])
[3, 2, 1]
```
A style of programming built upon functions

Typically, functions have no *side effects*
  - They just return *values*
  - They should effect no *non-local* changes

A functional program is just a starting function that calls other functions
Returning Functions

- In functional programs, higher-level functions are often used as factories returning other functions

- Similar to classes instantiating executable objects

- Both take initial data to customize the returned callable

- “Why return a value when you can return a function that returns the value” :-)

- Example: an addx function...
An Adding-function Factory

```python
# addx.py

def addx(x):
    # A nested function:
    def f(y):
        return x + y
    return f

add5 = addx(5)
print add5(1)       # 6
add10 = addx(10)
print add10(1)      # 11
```
# addx3.py

class Addx(object):
    def __init__(self,x):
        self.x = x
    def __call__(self,y):
        return self.x + y

add5 = Addx(5)
print add5(1)       # 6
add10 = Addx(10)
print add10(1)      # 11
Using a lambda Expression

```
# addx-lambda.py

def addx(x):
    return lambda y: x+y

add5 = addx(5)
print add5(1)       # 6
add10 = addx(10)
print add10(1)     # 11
```

When is `addx` “finished”?
Hey, Wait a Minute!

- The function **addx** has *returned* before we ever call the function it gave us
- How can we still access the x parameter of **addx**?
Closures

- When a function is passed as a parameter or returned as a value, it is invoked later in a different environment!
  - Could even be another module!

- But it still needs to access its original definition environment!
  - It may need to access its non-locals (aka “free variables”), for example!

- Therefore, functions always have a dictionary containing the needed bindings from their definition environments (including nonlocals)!

- The packaging of a function with its definition environment is called a closure (and is faster than using classes and __call__)!
Inspecting the Closure

# addx-lambda2.py: Inspects the closure

def addx(x):
    return lambda y: x+y

f = addx(5)
print f.func_closure[0].cell_contents, f(1)  # 5 6

g = addx(10)
print g.func_closure[0].cell_contents, g(1)  # 10 11

See also: dynclosure.py
Closures Are Dynamic

# Requires Python 3.x!

def fibonacci():
    cur = 0
    nxt = 1
    def func():
        nonlocal cur, nxt
        result = cur
        cur, nxt = nxt, cur + nxt
        return result
    return func

f = fibonacci()
for i in range(10):
    print(f(), end = ' ')

Output: 0 1 1 2 3 5 8 13 21 34
A Useful Higher-order Function

* **reduce** - processes sequence elements using a passed *binary* function
* Note: In Python 3.x this is in the **functools** module
* **Two calling signatures:**
  * **reduce**(*f*, *sequence*)
  * **reduce**(*f*, *sequence*, *init*)
* Very useful as a **factory**

```python
>>> reduce(lambda x,y: x+y, [1,2,3,4])  # 1+2+3+4
10
>>> reduce(lambda x,y: x*y, [1,2,3,4])  # 1*2*3*4
24
```
What reduce Does

# reduce.py: Implements reduce (2-arg version)

def myreduce(f, seq):
    assert seq  # Can't be empty
    if len(seq) == 1: return seq[0]
    result = f(seq[0], seq[1])
    for x in seq[2:]:
        result = f(result, x)
    return result

print reduce(lambda x, y: x+y, [1, 2, 3, 4])  # 10
print reduce(lambda x, y: x*y, [1, 2, 3, 4])  # 24
Reduce with an Initial Value

# reduce2.py: Implements reduce with initial value

def myreduce(f, seq, init):    # seq can be empty now
    result = init
    for x in seq:
        result = f(result, x)
    return result

print reduce(lambda x, y: x+y, [1, 2, 3, 4], 0)   # 10
print reduce(lambda x, y: x+y, [], 0)            # 0
print reduce(lambda x, y: x*y, [1, 2, 3, 4], 1)  # 24
print reduce(lambda x, y: x*y, [], 1)            # 1
The operator Module

* Defines common operators as functions

```python
# reduce3.py: Uses the operator module
import operator

print reduce(operator.__add__, [1, 2, 3, 4], 0)  # 10  
print reduce(operator.__add__, [], 0)            # 0   
print reduce(operator.__mul__, [1, 2, 3, 4], 1) # 24  
print reduce(operator.__mul__, [], 1)           # 1
```
Using reduce

# Find sum of squares of a sequence
seq = [1,2,3,4,5]
print reduce(lambda acc,elem:acc+elem*elem,seq,0)  # 55

def myany(seq):
    return reduce(lambda acc,elem:acc or elem,seq,False)

myall = lambda seq: reduce(lambda acc,elem: acc and elem,seq,True)

print myany([1,0])  # 1
print myany([1,1])  # 1
print myany([0,0])  # 0

print myall([1,0])  # 0
print myall([1,1])  # 1
print myall([0,0])  # 0
map and filter

- Preceded list comprehensions
  - Nowadays we usually just use comprehensions or generator expressions

- $\text{map}(f, \text{stuff}) = [f(x) \text{ for } x \text{ in } \text{stuff}]$
  - >>> map(lambda x: -x, [1, 2, 3])
    [-1, -2, -3]

- $\text{filter}(f, \text{stuff}) = [x \text{ for } x \text{ in } \text{stuff} \text{ if } f(x)]$
  - >>> filter(lambda x: x % 2 == 1, [1, 2, 3])
    [1, 3]
Function Composition

• A chain of function calls:
  • $f(g(h(x)))$
  • The output of one is the input of the next, enclosing function
  • Input and output types must be compatible (duh!)

• Handy to store a composition as a single function
  • $k(x) = f(g(h(x)))$
  • But how?
Implementing Composition

- Store the sequence of functions
- Return a function waiting for x; call in reverse order
- Chain the calls in an inside-out, pipeline fashion:

```python
def compose(*funs):
    def doit(x):
        result = x
        for f in reversed(funs):
            result = f(result)
        return result
    return doit

def f(x): return x+2
def g(x): return x*x
def h(x): return (x+1)/2

k = compose(f,g,h)
print k(4)  # 6  (= (4+1/2)^2 + 2)
print k(5)  # 11 (= (5+1/2)^2 + 2)
```
A More Functional Approach

def compose(*funs):
    return lambda x: reduce(lambda acc, f: f(acc), reversed(funs), x)

5 lines shorter!
Partial Function Evaluation

* A technique that allows calling a function with fewer args than expected

* Returns a function waiting for more args

* When all are finally supplied, the result is computed

* aka “currying”
Using `functools.partial`

```python
# partial.py
import functools

# Store the accumulator function
add = functools.partial(reduce, lambda x, y: x+y)
print add([1, 2, 3], 0)  # 6
print add([4, 5, 6], 100)  # 115

# Add the list
add123 = functools.partial(add, [1, 2, 3])
print add123(0)  # 6
print add123(100)  # 106
```
functools Handles Keyword args

```python
>>> def f(a, b):
...     print 'a =', a, 'b =', b
...
>>> f(b=1, 2)
File "<stdin>", line 1
SyntaxError: non-keyword arg after keyword arg

>>> g = functools.partial(f, b=1)
>>> g(2)  # Provide “a” last!
a = 2 b = 1
```
A Currying Decorator

Class Version

class curry(object):
    def __init__(self, f, *args):
        self.f = f
        self.args = args  # Initially empty when decorated
        self.nargs = f.func_code.co_argcount
    def __call__(self, *more_args):
        all_args = self.args + more_args
        return curry(self.f, *all_args) \
            if len(all_args) < self.nargs else self.f(*all_args)

@curry
def f(a,b,c):
    return a+b*c

# All calls print 7:
g = f(1)
print g(2,3)
A Currying Decorator

Function Version

def curry(f):
    nargs = f.func_code.co_argcount
    def curried(*args):
        return f(*args) if len(args) >= nargs \
        else lambda *more_args: curried(*args+more_args)
    return curried

@curry
def f(a,b,c):
    return a+b*c

# All prints print 7:
g = f(1)
print g(2,3)
h = g(2)
print h(3)
print f(1,2,3)
print f(1)(2)(3)
print f(1,2)(3)
print f(1)(2,3)

See currysort.py
Programming with Streams

* Streams are *unbounded* sequences evaluated only on demand
  * Accomplished with *generators*
  * Each iteration of the generator just computes the next item (duh)
  * We’ve seen these already (*primes, count*)
* Streams can be *composed*
* Example: *stream.py*
Numerical Approximations

- Newton’s Method
  - An iterative method for finding roots (zeroes) of equations

- Taylor Series
  - For calculating transcendental functions (sin, cos, exp)

- Two parts:
  - 1) The stream produces a sequence of approximations
  - 2) A control loop is the consumer of the stream
    - Quits when the approximations are “good enough”

- See newton.py, iterate.py, stream2.py, taylor.py
Coroutines

- Coroutines are generators that can yield output and receive input
  - Every `yield` is activated by a `next/send`; the coroutines exchange information
  - This is known as `cooperative multitasking` (aka “green threads”)
- Allows `multiple entry/exit` to/from a function during its lifetime
  - State is saved `internally` between exit/re-entry
  - Used heavily in `simulations`
- `Functions` are merely a `special case` of coroutines that have a `single` entry and exit per activation!
Coroutine Examples

* Note:
  * `yield <with no expression> == yield None`
  * `next() = send(None)`
  * Review: coroutine.py

* Producer-Consumer architecture (cooperative multitasking)
  * coroutine2.py

* Sending input to a generator
  * coroutine3.py

* “Push” equivalent to stream2.py
  * coroutine4.py
Coroutines and State Machines

- Coroutines allow a straightforward implementation of state machines
- Just consume until you get what you need
- Your “state” is where you are in the code
- Example: Comment Extractor
  - See next slide and comment.py
A $...$ Comment Extractor
**eval() and exec()**

- You can form expressions and statements at *runtime*
- By passing strings to:
  - **eval()** for *expressions*
  - **exec** for *statements*
eval/exec Example

```python
# eval_exec.py
x = input("Enter an integer value for x: ")
expression = raw_input("Enter an expression with x: ")
print eval(expression)
assign_y = raw_input("Enter an assignment for y: ")
exec assign_y
print y
```

$ python eval_exec.py
Enter an integer value for x: 10
Enter an expression with x: x*x+2
102
Enter an assignment statement for y: y = 'hello'
hello

See also: dynamic_call.py