More Python

CS156

Chris Pollett

Sep 10, 2014

Outline

- Lists, Tuples, Sets, and Dictionaries
- Iteration and Looping
- Functions, Generators, Co-routines
- Objects and Classes

Lists

- On Monday, we began talking about Python -- the programming language we are going to use in this AI course.
- We talked about installing it, running programs with it, variables, conditionals, File I/O, and strings.
- Today, we begin by looking at Python lists... Lists can be declared using square brackets like:

```
my_list = ['YO', 1.2, 7, "HI", ["scary nested list", "watch out!"]]
my_empty_list = [] # or list()
```

- We can access parts of a list in a variety of ways:

```
b = my_list[0] # b is now: 'YO'
c = my_list[4][1] # c is now: 'watch out!'
d = my_list[1:3] # d is [1.2, 7]
e = my_list[2:] # e is [7, 'HI', ['scary nested list', 'watch out!']]
```

- Lists can also be easily manipulated to make new lists:

```
my_list.append("an end") #my_list now
    #['YO', 1.2, 7, 'HI', ['scary nested list', 'watch out!'], 'an end']
my_list.insert(2, 3) # my_list now
    #['YO', 1.2, 3, 7, 'HI', ['scary nested list', 'watch out!'], 'an end']
a = [1, 2, 3] + [4, 5] #a is [1, 2, 3, 4, 5]
```

Example Using Lists and Command Line

```
import sys
if(len(sys.argv)) != 2:
   #notice sys.argv is a list of command-line args and we found its length
   print "Please supply a filename"
   raise SystemExit(1) # throw an error and exit
f = open(sys.argv[1]) #program name is argv[0]
lines = f.readlines() # reads all lines into list one go
f.close()

#convert inputs to list of ints
ivalues = [int(line) for line in lines]

# print min and max
print "The min is ", min(ivalues)
print "The max is", max(ivalues)
```

Tuples

- You can pack a collection of objects into a single object in Python using a tuple

```
a = ( 1, "hello", 3)
b = ( some, where)
c = "6 scared of 7", "as 7 8 9" #notice can omit paren's
d = () # 0-tuple
e = 'yo', #one tuple
f = ('yo',) #same one tuple
g = (d,) # g is ((),)
```

- Can access parts of tuple with the same notation as lists a[1] gives 'hello' for instance. Or we can do things like:

```
c = (4, 5)
a, b = c
```

- Tuples cannot be appended or inserted to, but save memory.

Sets

- A **set** is used to contain an unordered collection of objects.
- Sets can be created, viewed, and check for membership using statements like:

```
my_set = set([3, 9, 2, 6])
another_set = set("goodness") # set of unique chars
print another_set # set(['e', 'd', 'g', 'o', 'n', 's'])
if 'e' in another_set:
    print "it's in there"
```

- Sets support various operations:

```
a = my_set; b = another_set;
c = a | b # union of sets
c = a & b # intersection of sets
c = a - b # difference of sets
c = a ^ b # symmetric difference of sets
another_set.add('y') # adds a single element to set
another_set.update([6,7,8]) # add multiple elements
my_set.remove(3) # removes the number 3 from my_set
```

- A frozenset is like a set but cannot be changed

Dictionaries

- Python's notion of an associative array (a table of key-value pairs) is called a **dictionary**.
- Dictionary literal make use of curly braces:

```
person = {
    "name" : "bob",
    "age" : 27,
    "sex" : "Male"
}
empty_dict = {} #an empty dictionary # or use dict()
```

- We can access and modify elements of a dictionary with statements like:

```
name= person["name"]
person["age"] = 28
person["address"] = "somewhere" #this would add a key-value pair
del person["age"] # removes key value associated with 'age'
```

- Membership of a key in a dictionary can be tested with 'in':

```
if "name" in person:
   name = person["name"]
else:
   name = "no one"
#the above conditional can be shortened to:
name = person.get("name", "no one")
```

- Aside: Python has built in functions locals() and globals() and the syntax: x in locals() or x in globals() might be used to check if a variable is set.
- The list of keys in a dictionary can be gotten with:

```
keys = list(person) #keys is ['address', 'name',  'sex']
#could also do person.keys()
#person.values() would get list of values
#person.len() gives the number of keys in dictionary
```

Iteration and Looping

- Python supports while loops like most languages:

```
while condition:
    statement1
    statement2
    ...
```

- Python for loops always behave as a foreach over a sequence:

```
for n in [1,2,3,4,5,6,7,8,9]:
    print "2 to the %d  is %d" % (n, 2**n)
```

- To make this less cumbersome, Python allows you to declare ranges such as range(1, 10):

```
for n in range(1,10):
    print "2 to the %d  is %d" % (n, 2**n) # same as before
```

- The syntax for ranges is: range(start_inclusive, end_exclusive [, stride]):

```
a = range(5) # can omit start to get a = 0, 1, 2, 3, 4
b = range(1,8) # b = 1, 2, 3, 4, 5, 6, 7
c = range(0, 13, 2) # c = 0, 2, 4, 6, 8, 10, 12
d = range(7, 2, -1) # d = 7, 6, 5, 4, 3
```

- In Python 2, range actually creates a list which can be quite memory intensive for big ranges. If you didn't want a list but just something to iterate over you used xrange. In Python 3, xrange is no more and range() have the default behavior of xrange.

Examples of things can iterate over with for

- As we've seen in some of our earlier examples, for can be used with strings, lists, dictionaries, file objects, etc:

```
a = "Get rich quick"
for b in a:
    print b

c = ["now", "I", "know"]
for d in c:
    print c

person = {
    "name" : "bob",
    "age" : 27,
    "sex" : "Male"
}
for key in person:
    print key, person[key]

f = open("my_file.txt")
for line in f
    print line
```

Functions

- The keyword **def** is used to create functions in Python:

```
def smaller_value( a, b):
   if a < b: return a # notice single line if can be same line
   else: return b
```

- Function invocation is similar to most languages:

```
print smaller_value(8, 4)
```

- Functions can be recursive and are allowed to take as arguments and return tuples, lists, etc:

```
def reverse_list(list):
    if list == []: return []
    return reverse_list(list[1:]) + [list[0]]
print reverse_list([1, 2, 3, 4]) # prints [4, 3, 2, 1]

def divide(a,b):
   q = a // b
   r = a - q*b
   return (q, r)
quotient, remainder = divide(2373, 16)
```

- We can also supply default values to functions:

```
def expify(a, b=2):
   return b**a #if don't give a second argument b will be 2
```

Function Scoping, Functions as Variables

- By default variables used in a function definition are local.

```
i = 5
def printi():
   i=4
   print i
printi() # outputs 4
print i #outputs 5
#note without the i=4 assignment would get i=5
```

- To be able to modify a variable in global scope we must use the keyword global:

```
def assign_i():
  global i
  i=3
assign_i()
print i #now get 3
```

- Function names can be treated as references to code in memory, so we can assign them to variables. i.e., We have function pointers and functions can be passed to other functions, etc:

```
a = printi
a() # prints 4
```

- Python also supports anonymous function definitions:

```
Consider:
def f(x): return x**3
f(10) #returns 1000

g = lambda x: x**3 #notice don't use return with lambda
g(10) #returns 1000

def make_adder (n): return lambda x: x + n
f = make_adder(2)
g = make_adder(6)
print f(42), g(42)
```

- Anonymous function are often used to write "one-off" functions like callback handlers in GUI's and things.

Generators

- We can generate sequences one element at a time using the **yield** construct:

```
def countdown(n):
    while n > 0:
       yield n
       n -=1
c = countdown(5)
print c.next() #prints 5
print c.next() #prints 4
print c.next() #prints 3
```

- The next() method runs to the next yield statement
- Here's how we could get a function which monitors a log file and returns lines as they appear:

```
import time
def tail_log(f):
    f.seek(0, 2) # 2nd arg: 0 means start of file, 1 means current pos, 2 means end of fi
    while True:
        line = f.readline()
        if not line:
            time.sleep(0.1) # sleep 1/10 of a second
            continue
        yield line
```

Coroutines

- Generators are functions which output a sequence of values when their next() method is called
- One can imagine the opposite situation where a function receives a sequence of values one at a time and computes something.
- Such a function is called a **coroutine** and can also be defined using yield:

```
def print_matches(text):
    print "Trying to find", text
    while True:
        line = (yield)
        if text in line:
            print line
p = print_matches("hello")
p. next()  #prints 'Trying to find hello'
p.send("lalalala la") #doesn't print anything
p.send("hello world") #prints hello world
```

Objects and Classes

- We have already seen that lists, dicts, etc are objects and we can invoke methods on them much as we do in other languages. For instance, my_list.append("hello")
- Given an object we can see its methods by calling dir(my_obj). For example, a=[]; dir(a); gives:

```
['__add__', '__class__', '__contains__', '__delattr__', '__delitem__',
'__delslice__', '__doc__', '__eq__', '__format__', '__ge__', '__getattribute__',
'__getitem__', '__getslice__', '__gt__', '__hash__', '__iadd__', '__imul__',
'__init__', '__iter__', '__le__', '__len__', '__lt__', '__mul__', '__ne__',
'__new__', '__reduce__', '__reduce_ex__', '__repr__', '__reversed__', '__rmul__',
'__setattr__', '__setitem__', '__setslice__', '__sizeof__', '__str__',
'__subclasshook__', 'append', 'count', 'extend', 'index', 'insert', 'pop',
'remove', 'reverse', 'sort']
```

- You can see append in the list. Special methods such as operator overloads have '__' around them. For example, __add__ in the above corresponds to the + operator. We can actually use the + operator in a cumbersome way as: a.__add__([1, 2]).
- If you were writing your own class overriding __add__ would give you operator overloading of "+" for operations of the form my_obj + something_else. To override something_else + my_obj you would need to implement __radd__. To override += use overrider __iadd__, etc.
- __init__ represents the constructor.
- To define new classes we using the keyword class. For example,

```
class Stack(object): #this says stack inherits from object
    a_class_variable = 5 # this var behaves like a Java static var
    def __init__(self): #self = this in Java
        self.stack = [] #now stack is a field variable of Stack
        #in general using self.field_var is how we declare and
        #instantiate a instance variable
    def push(self, object): #the first argument of any method
        self.stack.append(object) # is the object itself
    def pop(self):
        return self.stack.pop()
    @property #properties are computed attributes
    def length(self):
        return len(self.stack)
    #(where an attribute is like a field) of a class
    # Can use dot notation with or without parentheses to invoke
```

- To instantiate an instance of this class and use it one could do:

```
my_instance = Stack()
my_instance.push("hello")
print my_instance.length
print isinstance(my_instance, object) #returns True
print issubclass(Stack, object) #returns True
#type(my_instance) returns Stack
...
```

```
        #etc
```