

Python Programming

Lists and Dictionaries

Outline

- Special entities
- Lists
 - List comprehensions
- Dictionaries

Return values -- and None

- Functions that don't return a value return **None**
- **None**:
 - A special entity (object) in python
 - Represents the concept of nothing
 - Type **None** (no quotes) into the interpreter

```
>>> None
>>>
```
 - **None** evaluates to ... Nothing. (No value printed.)

Boolean values - True, False

- Operators that perform tests return boolean values
 - For example, `==` tests whether its operands are equal in value:

```
>>> A = [1,2,3] # assignment
>>> B = [1,2,3] # assignment
>>> A == B # boolean returned
True
```
- In python, 0 and None are False:

```
>>> 0 == False
True
>>> None == False
True
```
- All other entities are True:

```
>>> 'something' == True
True
```

Lists

- Lists are sequences.
- You do not have to declare ahead of time what types of objects they will contain.
- They can contain anything, including other lists.
- They are *mutable* – which means, you can modify them.

Creating new lists

- Square brackets:

```
>>> lst = [] # an empty list
>>> lst = [2,3,'a']
>>> lst = [2,3,4+5,'z']
```
- Using `list`:

```
>>> list()
```
- Using `range`:

```
>>> range(1,5)
[1,2,3,4]
```

Note: the result does not include 5

Modifying lists

- Add an element to a list using “append”

```
>>> lst = [2,3,'abc']
>>> lst.append(5.0)
>>> lst
>>> [2,3,'abc',5.0]
```

- Add an element to a list using “insert”

```
>>> lst = range(1,5)
>>> lst
[1,2,3,4]
>>> lst.insert(2,2.5)
>>> lst
>>> [1,2,2.5,3,4]
```

- Or use assignment operator:

```
>>> lst[2] = 23
>>> lst[2]
>>> 23
```

List comprehension

- Syntax:

– $L = [expression \text{ for } variable \text{ in } sequence]$

- Example:

```
>>> L1 = [x*x for x in range(1,3)]
```

variable evaluates to a sequence
keywords

- Example:

```
>>> L2 = [[L1[i],L1[i+1]] for i in range(0,len(L1))]
```

Sorting lists

- Lists have a sort method
`>>> lst.sort(f) # f is optional (default is cmp)`
- `f` -- a function that:
 - Accepts two arguments (x,y).
 - Returns a **negative** value if `x < y`.
 - Where x goes first in the list sorted in ascending order.
 - Returns a **positive** value if `x > y`.
 - Where x goes last in a list sorted in ascending order.
 - Returns 0 if x & y have the **same** value.
 - We can't decide which should go first.
- The sort happens *in place*
 - The order of items in the list changes.

```
def cmp(x,y):  
    if x<y:  
        return -1  
    elif x>y:  
        return 1  
    else:  
        return 0
```

Example: custom comparison method (using instead of `cmp`)

- Sort strings by size only. Shorter strings should be closer to the front of the list.

```
>>> def scmp(x,y):  
...     return len(x)-len(y) ← len returns the number of  
...                                     characters in a string  
...  
>>> test = ['atcgatccat', 'atc', 'ggg', 'atta']  
>>> scmp(test[0],test[1])  
6  
>>> scmp(test[0],test[0])  
0  
>>> test.sort(scmp) ← This invocation of sort will use  
>>> test                                     the default cmp method.  
['atc', 'ggg', 'atta', 'atcgatccat']  
>>> test.sort() ←  
>>> test  
['atc', 'atcgatccat', 'atta', 'ggg']
```

Sorting lists, con't

- Built-in method `sorted` returns a *copy* of the list, but with elements sorted in an order.

```
>>> L = [1,2,3,4]
>>> L.reverse()
>>> L
[4,3,2,1]
>>> sorted(L)
[1,2,3,4]
```

- `sorted` takes an optional second argument
 - a comparison function (same as for `L.sort`)
 - `cmp` is the default sorting method

Filtering lists

- `filter` - tests all the members of a list or other type of sequence, and returns a new list

– Syntax

- `filter(function , sequence)`

- Example:

```
>>> L = range(1,100,10)
>>> L
[1, 11, 21, 31, 41, 51, 61, 71, 81, 91]
>>> def test(x): return x > 50
...
>>> filter(test,L)
[51, 61, 71, 81, 91]
>>> []
```

lambda lets you define functions in one line (easier to type)

- Syntax:
 - lambda arglist: expression
- The return value is whatever the last evaluated expression returned

```
>>> lambda x,y:x+y
<function <lambda> at 0x68db0>
>>> g = lambda x,y:x+y
>>> g(4,5)
9
>>> filter(lambda x:x>50,L)
[51, 61, 71, 81, 91]
```

lambda expressions return functions

Use them as arguments to filter.

Another way to do list comprehensions:

- map applies a function to all elements in a sequence:
 - Syntax: map(function, sequence)
- Example:

```
>>> L = range(1,10)
>>> L
[1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> map(str,L)
['1', '2', '3', '4', '5', '6', '7', '8', '9']
>>> map(lambda x:str(x)+'.',L)
['1.', '2.', '3.', '4.', '5.', '6.', '7.', '8.', '9.']
>>>
```

Dictionaries

- Store values by key, where keys are typically strings, but can be any *hashable* object
- In other languages, called hashtables

```
>>> d = {}
>>> d['Ann']='Lorraine'
>>> d
{'Ann': 'Lorraine'}
>>> d['Ann']
'Lorraine'
>>> L = range(1,10)
>>> d[L] = 'a list'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: list objects are unhashable
>>> import re
>>> d[re]='foo'
>>> type(re)
<type 'module'>
>>> []
```

Dictionary methods

- Use `keys` to retrieve a list of all the keys in a dictionary.

```
>>> D = {}
>>> D['foo']='bar'
>>> D['FOO']='BAR'
>>> D.keys()
['foo', 'FOO']
```
- Use `values` to retrieve a list of all the values in a dictionary.

```
>>> D.values()
['bar', 'BAR']
```
- Don't count on items in either list appearing in any particular order.

Dictionary methods

- Use `has_key` to find out if a given key exists in a dictionary:

```
>>> D.has_key('foo')
True
```

- Using a non-existent key generates an error:

```
>>> D = {'FOO': 'foo', 'BAR': 'bar'}
>>> D['FOO']
'foo'
>>> D['baz']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'baz'
```

Homework

- Problems - list comprehension, side effects/return values
- Mini-project:
 - Answering some questions about genes, publications
 - Uses: Entrez Gene database.