Lists, Functions, & Input/Output in Python

L445/L515
Autumn 2011
Lists

Python has a number of (compound) data types for combining other values, most notably **lists**

```python
a = ['hotel','motel',100]
```

- Lists are concatenated and sliced in the same way that strings are

```python
>>> aa = ['word']
>>> bb = ['up']
>>> cc = aa + bb # cc = ['word', 'up']
>>> cc[1]
'up'
```

- `len` gets the length of the list: `len(a)` equals 3
- `sort` and `reverse` do what you pretty much expect them to
Out of bounds errors

A common mistake is to try to index a part of the list that isn’t there

```python
>>> a = ['hotel', 'motel']
>>> a[2]
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
IndexError: list index out of range
```
A new way to iterate

A for loop allows you to iterate directly over the items in a list

```python
>>> a = ['hotel', 'motel', 'inn']
>>> for item in a:
...    print "Are you staying in a(n) " + item + "?"
...
Are you staying in a(n) hotel?
Are you staying in a(n) motel?
Are you staying in a(n) inn?
```
Another iteration method

List method: `pop()` removes the last item

- So, we can pop items from a list until there is no more list:

```python
>>> a = ['hotel', 'motel', 'inn']
>>> while a:
...     print "Are you staying in a(n) " + a.pop() + "?"
...     
... Are you staying in a(n) inn?
Are you staying in a(n) motel?
Are you staying in a(n) hotel?
>>> a
[]
```

NB: Only do this if you don’t need the list contents when you’re done
And yet another ...

The built-in function `range` takes an integer `i` and returns a list of integers from 0 to `i-1`.

- So, we can use this in combination with `len`.

```python
>>> for i in range(len(a)):
...     print "Are you staying in a(n) " + a[i] + "?"
... 
Are you staying in a(n) hotel?
Are you staying in a(n) motel?
Are you staying in a(n) inn?
```

```python
>>> a
['hotel', 'motel', 'inn']
```

Allows you to access both the index (i) and the list value at that index (a[i]).
List operations

- **append**—add list item to back of list
  
  ```python
  a.append('inn')  # a = ['hotel','motel',100,'inn']
  ```

- **insert**—add item at a given position
  
  ```python
  a.insert(0,'hostel')  # a = ['hostel','hotel','motel',100,'inn']
  ```

- **pop**—remove item at a given position
  
  ```python
  # a = ['hostel','hotel','motel',100,'inn']
  a.pop()  # a = ['hostel','hotel','motel',100]
  a.pop(0)  # a = ['hotel','motel',100]
  ```

- **index**—get index of element
  
  ```python
  a.index('motel')  # 1
  ```
Functions

Let’s say we have input text line-by-line and for each line, we do the same operation. It’s extremely useful to use a function in this case.

In one part of code (traditionally, near the beginning):

```python
def get_word(x):
    # x is in the form 'word tag'
    # split takes a string and turns it into a list
    text = x.split()
    return text[0]
```

In another part of code, where needed, we put:

```python
word = get_word(line)
```

Even though it takes two steps within the function, it's one conceptual idea, and now it takes up less space in the main part of the code, where we need it.
Functions, piece by piece

Let’s re-look at that function we just saw:

```python
def get_word(x):
    return text[0]
```

Two things are happening here:

- `def` is being used to indicate that a function is being defined. You give the name to the function, in this case, `get_word`.
- `x` is an *argument*, which means that it’s a variable and can take on any value.
  - When we call this function with `get_word(line)`, the value of the variable `line` in the main part of the program gets assigned to `x`.
  - In other words, `x` is *local* only to the function, but gets passed values from other parts of the program.
- `return` is used to send a value (or values) back to the part that called the function.
Default arguments

Here's a function that calculates $x^n$, but by default it calculates $x^2$

```python
def power(x, n=2):
    result = 1
    for i in range(n):
        result *= x
    return result
```

```python
x = power(3, 4)  # x = 81
y = power(3, 2)  # y = 9
z = power(3)    # z = 9
```
Modularity

In general, functions allow programming to be very **modular**

- e.g., I can ask you to write one function, and you won’t have to worry at all about how it’s being used

- This work as long as each function’s **specifications** are clear
  - What precisely is the expected input?
  - What precisely is the intended output?
Basic Input/Output

A very basic way to get input from a user is interactively, e.g.,

\[ x = \text{raw_input}("Please give me a number: ") \]

Likewise, a very basic way to display output is through the `print` statement

\[ \text{print } "word" \]

But we would like to interact with large text files
File Input

There are many ways to read in file input (e.g., `read`)

A very simple way to iterate is to use the following convention:

```python
filename = "whateveritscalled.txt"
for line in open(filename):
    print line
```

This gives us access to each line from the file

- You can think of `open(filename)` as giving you a *list* of lines from the file
File Output

The open method by default reads a file, but can be altered to write to a file (using the 'w' option):

```python
words = ['a', 'rose', 'is', 'a', 'rose', 'is',
        'a', 'rose']
myfile = open('newfile.txt', 'w')

for word in words:
    myfile.write(word)

myfile.close()
```