Introduction to Python
Functions, Modules, and Sequences

Christopher Barker

UW Continuing Education

October 8, 2013

Table of Contents

Review of Previous Class

- Values and Types
- Expressions
- Intro to functions

Lightning Talks

Lightning talks today:

Jo-Anne Antoun

Omer Onen

Ryan Small

Catherine Warren

Homework review

Homework Questions?

My Solution

Stuff brought up by homework

Bytecode and `*.pyc`

Please send me code:

- enclosed in an email
- with your name at the beginning of the filename: `chris_problem1.py`

PEP 8

Repeating variable names in nested loops

Basics

It turns out you can't really do much at all without at least a container type, conditionals and looping...

if

When you need to make a decision:

```
if x > 500:
    print "x is big!"
else:
    print "x is small"
```

lists

A way to store a bunch of stuff in order

“array” in other languages

```
a_list = [2,3,5,9]

a_list_of_strings = ['this', 'that', 'the', 'other']
```

tuples

Another way to store an ordered list of things

```
a_tuple = (2,3,4,5)

a_list_of_strings = ('this', 'that', 'the', 'other')
```

Often interchangeable with lists, but not always...

for

When you need to do something to everything in a sequence

```
>> a_list = [2,3,5,9]

>> for item in a_list:
>>     print item
2
3
5
9
```

range and for

When you need to do something a set number of times

```
>>> range(4)
[0, 1, 2, 3]
>>> for i in range(6):
...     print "*",
...
* * * * * *
>>>
```

intricacies

This is enough to get you started.

Each of these have intricacies special to python

We'll get to those over the next couple classes

Functions: review

Defining a function:

```
def fun(x, y):
    z = x+y
    return z
```

x, y, z are local names

Functions: local vs. global

```
x = 32
def fun(y, z):
    print x, y, z

fun(3,4)
```

```
32 3 4
```

x is global, y and z local

Use global variables mostly for constants

Recursion

Recursion is calling a function from itself.

Max stack depth, function call overhead.

Because of these two(?), recursion isn't used **that** often in Python.

(demo: factorial)

Tuple Unpacking

Remember: `x,y = 3,4` ?
Really "tuple unpacking": `(x, y) = (3, 4)`
This works in function arguments, too:

```
>>> def a_fun( (a, b), (c, d) ):
...     print a, b, c, d
...
>>> t, u = (3,4), (5,6)
>>>
>>> a_fun(t, u)
3 4 5 6
```

(demo)

Lab: more with functions

write a function that:

- copnutes the distance between two points:
  dist = sqrt((x1-x2)**2 + (y1-y2)**2)
  using tuple unpacking...
- Take some code with functions, add this to each function:
  `print locals()`
- Computes the Fiboacci series with a recursive function:
  f(1) = 1
  f(n) = f(n-1) + f(n-2)
  0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Lightning Talks

Lightning Talks:

Jo-Anne Antoun

Omer Onen

if

Making Decisions...

```
if a:
    print 'a'
elif b:
    print 'b'
elif c:
    print 'c'
else:
    print 'that was unexpected'
```

if

Making Decisions...

```
if a:
    print 'a'
elif b:
    print 'b'

## versus...

if a:
    print 'a'
if b:
    print 'b'
```

switch?

No switch/case in Python

use `if..elif..elif..else`

(or a dictionary, or subclassing....)

Truthiness

What is true or false in Python?

- The Booleans: `True` and `False`
- "Something or Nothing"

`http://mail.python.org/pipermail/python-dev/2002-April/022107.html`

Truthiness

Determining Truthiness:

```
bool(something)
```

Boolean Expressions

`False`

- `None`
- `False`
- zero of any numeric type, for example, `0`, `0L`, `0.0`, `0j`.
- any empty sequence, for example, `''`, `()`, `[]` .
- any empty mapping, for example, `{}`.
- instances of user-defined classes, if the class defines a `__nonzero__()` or `__len__()` method, when that method returns the integer zero or bool value `False`.

`http://docs.python.org/library/stdtypes.html`

Boolean Expressions

Avoid:

```
if xx == True:
```

Use:

```
if xx:
```

Boolean Expressions

"Shortcutting"

```
x or y          if x is false,
                   return y,
                   else return x

x and y         if x is false,
                    return  x
                    else return y

not x           if x is false,
                  return True,
                  else return False
```

Boolean Expressions

Stringing them together

```
 a or b or c or d
```

```
a and b and c and d
```

The first value that defines the result is returned

(demo)

Boolean returns

From CodingBat

```
def sleep_in(weekday, vacation):
    if weekday == True and vacation == False:
        return False
    else:
        return True
```

Boolean returns

From CodingBat

```
def sleep_in(weekday, vacation):
    return not (weekday == True and vacation == False)
```

or

```
def sleep_in(weekday, vacation):
    return (not weekday) or vacation
```

bools are ints?

bool types are subclasses of integer

```
In [1]: True == 1
Out[1]: True

In [2]: False == 0
Out[2]: True
```

It gets weirder!

```
In [6]: 3 + True
Out[6]: 4
```

(demo)

Conditional expression

A common idiom:

```
if something:
    x = a_value
else:
    x = another_value
```

Also, other languages have a "ternary operator"
(C family: `result = a > b ? x : y ;`)

```
y = 5 if x > 2 else 3
```

PEP 308: (http://www.python.org/dev/peps/pep-0308/)

LAB

- Look up the % operator. What do these do?
  ```
  10 % 7 == 3
  14 % 7 == 0
  ```
- Write a program that prints the numbers from 1 to 100 inclusive. But for multiples of three print “Fizz” instead of the number and for the multiples of five print “Buzz”. For numbers which are multiples of both three and five print “FizzBuzz” instead.

LAB

Re-write a couple CodingBat exercises, using a conditional expression

Re-write a couple CodingBat exercises, returning the direct boolean results

(use whichever you like, or the ones in: `code/codingbat.rst`)

Lightning Talks

Lightning Talks:

Ryan Small

Catherine Warren

Code Structure

Python is all about namespaces – the "dots"

```
name.another_name
```

the "dot" indicates looking for a name in the namespace of the given object.
could be:

- name in a module
- module in a package
- attribute of an object
- method of an object

indenting and blocks

Indenting determines blocks of code

```
something:
    some code
    some more code
    another block:
        code in
        that block
```

But you need the colon too...

indenting and blocks

You can put a one-liner after the colon:

```
In [167]: x = 12

In [168]: if x > 4: print x
12
```

Only do this if it makes it more readable...

Spaces and Tabs

An indent can be:

- Any number of spaces
- A tab
- tabs and spaces:
  - A tab is eight spaces (always!)
  - Are they eight in your editor?

Use four spaces – really!

(PEP 8)

Spaces Elsewhere

Other than indenting – space doesn't matter

```
x = 3*4+12/func(x,y,z)

x = 3*4 + 12 /   func (x,   y, z)
```

Choose based on readability/coding style

PEP 8

Various Brackets

Bracket types:

- parentheses ( )
  - tuple literal: `(1,2,3)`
  - function call: `fun( arg1, arg2 )`
  - grouping: `(a + b) * c`
- square brackets [ ]
  - list literal: `[1,2,3]`
  - sequence indexing: `a_string[4]`
- curly brackets { }
  - dictionary literal: `{"this":3, "that":6}`
  - (we'll get to those...)

Sequences

Sequences are ordered collections of objects

They can be indexed, sliced, iterated over,...

They have a length: `len(sequence)`

Common sequences (Remember Duck Typing?):

- strings
- tuples
- lists

Indexing

square brackets for indexing: `[]`

Indexing starts at zero

```
In [98]: s = "this is a string"

In [99]: s[0]
Out[99]: 't'

In [100]: s[5]
Out[100]: 'i'
```

Indexing

Negative indexes count from the end

```
In [105]: s = "this is a string"

In [106]: s[-1]
Out[106]: 'g'

In [107]: s[-6]
Out[107]: 's'
```

Slices

Slicing: Pulling a range out of a sequence

```
sequence[start:finish]
```

```
indexes for which:
```

```
start <= i < finish
```

Slices

```
In [121]: s = "a bunch of words"
In [122]: s[2]
Out[122]: 'b'

In [123]: s[6]
Out[123]: 'h'

In [124]: s[2:6]
Out[124]: 'bunc'

In [125]: s[2:7]
Out[125]: 'bunch'
```

Slices

the indexes point to the spaces between the items

```
  X   X   X   X   X   X   X   X
|   |   |   |   |   |   |   |
0   1   2   3   4   5   6   7
```

Slices

Slicing satisfies nifty properties:

```
len( seq[a:b] ) == b - a

seq[a:b] + seq [b:c] == seq
```

Slicing vs. Indexing

Indexing returns a single element

```
In [86]: l
Out[86]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [87]: type(l)
Out[87]: list

In [88]: l[3]
Out[88]: 3

In [89]: type( l[3] )
Out[89]: int
```

Slicing vs. Indexing

Unless it's a string:

```
In [75]: s = "a string"

In [76]: s[3]
Out[76]: 't'

In [77]: type(s[3])
Out[77]: str
```

There is no single character type

Slicing vs. Indexing

Slicing returns a sequence:

```
In [68]: l
Out[68]: [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

In [69]: l[2:4]
Out[69]: [2, 3]
```

Even if it's one element long

```
In [70]: l[2:3]
Out[70]: [2]

In [71]: type(l[2:3])
Out[71]: list
```

Slicing vs. Indexing

Indexing out of range produces an error

```
In [129]: s = "a bunch of words"
In [130]: s[17]
----> 1 s[17]
IndexError: string index out of range
```

Slicing just gives you what's there

```
In [131]: s[10:20]
Out[131]: ' words'

In [132]: s[20:30]
Out[132]: ''
```

(demo)

Multiplying and slicing

from CodingBat: Warmup-1 – front3

```
def front3(str):
  if len(str) < 3:
    return str+str+str
  else:
    return str[:3]+str[:3]+str[:3]
```

or

```
def front3(str):
    return str[:3] * 3
```

Slicing

from CodingBat: Warmup-1 – `missing_char`

```
def missing_char(str, n):
  front = str[0:n]
  l = len(str)-1
  back = str[n+1:l+1]
  return front + back

def missing_char(str, n):
    return str[:n] + str[n+1:]
```

Slicing

you can skip items, too

```
In [289]: string = "a fairly long string"

In [290]: string[0:15]
Out[290]: 'a fairly long s'

In [291]: string[0:15:2]
Out[291]: 'afil ogs'

In [292]: string[0:15:3]
Out[292]: 'aallg'
```

Command Line Input

`input` evaluates the input:

```
In [265]: val = input("a message> ")
a message> 4.5
In [266]: type(val)
Out[266]: float
```

`raw_input` gives you the plain string:

```
In [265]: val = input("a message> ")
a message> 4.5
In [266]: type(val)
Out[266]: float
```

(demo)

LAB

`def count_them(letter):`

- prompts the user to input a letter
- counts the number of times the given letter is input
- prompts the user for another letter
- continues until the user inputs "x"
- returns the count of the letter input

`def count_letter_in_string(string, letter):`

- counts the number of instances of the letter in the string
- ends when a period is encountered
- if no period is encountered – prints "hey, there was no period!"

LAB

Write some functions that:

- return a string with the first and last characters exchanged.
- return a string with every other character removed
- return a string with the first and last 4 characters removed, and every other char in between
- return a string reversed (just with slicing)
- return a string with the middle, then last, then first third in a new order

Homework

Recommended Reading:

- Read Think Python: 9, 14
- extra: string methods: `http://docs.python.org/library/stdtypes.html#string-methods`
- extra: unicode: `http://www.joelonsoftware.com/articles/Unicode.html`

Do:

- Six more CodingBat exercises.
- LPTHW: for extra practice with the concepts – some of:

strings:	ex5, ex6, ex7, ex8, ex9, ex10
raw_input(), sys.argv:	ex12, ex13, ex14 (needed for files)
files:	ex15, ex16, ex17