CSCI 1300

Intro to Computing

Gabe Johnson

Lecture 19 Feb 27, 2013

Python Review

Lecture Goals

1. Exam 2 Plan of Action
2. Review All of Python

Upcoming Exam

Test #2 **Friday, March 1**

Exam 2: Most of Python

The exam is 3 pages, worth 30 points, and is mostly short answer and multiple choice.

At the start of class, get a test from the top of the stairs, take a seat, don't chat with people about it, and get to it right away. *We won't deliver you a test this time since we'll be handing things out atop the stairs.*

We also have to cut it off after 50 minutes this time. Bring finished tests up front.

Keywords, Literals, Variables

A Python program is composed of three things:

- **Keywords**: these are words that Python looks for that cause special action. Examples: `def`, `class`, `for`, `while`, `return`.
- **Literals**: A literal is data that Python directly reads. Examples: `4.7`, `“Hello”`, `True`, `None`, `[ ]`, `{ }`
- **Variables**: These are named items that refer to some specific value. They can be whatever we want, as long as Python doesn’t confuse it with a literal or a keyword. Examples: `foo`, `bar`, `ret`, `my_list`.

Expressions

An expression is one or more statements that give (or combine to give) a value. These are all expressions:

Statement	Value
2	2
2 + 4	6
8 * “Foo”	FooFooFooFooFooFooFooFoo'
10 + 2 > 5	True

Types

All statements that have value have a *type*. This is what kind of thing it is. An item's type determines how we can use it. For example, if we divide an integer by another integer, we are guaranteed to get an integer as a result. But if we divide an integer with a floating point number, the result is a floating point number.

Expressions also have types. 5+8 is an expression that yields an integer. "foo" + "bar" is an expression that involves two strings, and results in a string.

Type	Description	Example 1	Example 2
integer	counting number	0	84
float	continuous number	0.0	84.5
string	textual data	“Hello”	My dog is cool’
boolean	truth value	TRUE	False
None	represents nothing	None	--
dictionary	a map of key / value pairs	{ }	{ ‘name’: ‘Sputnik’, ‘age’: 4 }
list	sequential data	[ ]	[‘sputnik’, 42, None, False, “Hello World”]
class	a blueprint for making object instances	class Slug: pass	class House: color = “Blue”
object	a particular, independent instance of a class	s = Slug()	my_house = House()

Use type(x) to get x's type

If you want to find out what type of item something is, you can always use the type( ) function:

```
x = 4
print type(4)
```

this says: <type 'int'>

Same thing works for expressions:

```
happy = True
know_it = True
print type(happy and know_it)
```

this says: <type 'bool'>

For-loops

A *for-loop* lets us cycle through a sequence of things, giving us a chance to work with each item in order.

```
numbers = [5.0, 7, 3.0, 3 ]
for val in numbers:
    print val, "/", 2, "is:", (val / 2)
```

This prints →

```
5.0 / 2 is: 2.5
7 / 2 is: 3
3.0 / 2 is: 1.5
3 / 2 is: 1
```

(think about why 7 / 2 is 3!)

For-loops, again

```
for val in numbers:
    x = val * val + 3
    print x
```

Here, ‘numbers’ is something we can iterate through, like a list or a dictionary. It goes through all the items, and each item gets one run through the body of the loop. During an item’s turn, it is put into the variable called ‘val’. We can use it any way we want to. Here we calculate val squared, plus three.

While-Loops

Another loop is a *while-loop*. This will execute it's associated loop body as long as some boolean expression is true.

```
count = 4
while count > 0:
      print count, "..."
      count = count - 1
print "Blast off!"
```

```
4 ...
3 ...
2 ...
1 ...
Blast off!
```

While-loops, again

This while-loop's boolean expression is *count > 0*.

```
count = 4
while count > 0:
      print count, "..."
      count = count - 1
print "Blast off!"
```

As long as count is larger than zero, it will execute the loop body from top to bottom. Be careful: if the condition never becomes false, it will loop forever, until you forcibly stop your program. In this case, we decrement count by one every time.

Dictionaries

A dictionary lets us use one piece of information to look up some other piece of information. I know that there's this word, *cromulent*, but I need the definition. So I can use a physical dictionary and find the page where *cromulent* is defined, and I can find out what it means.

Here, *cromulent* is the *key*, and the definition (whatever that is) is the *value*.

Dictionaries, again

A dictionary can be initialized with or without key/ value pairs. Here, ‘blank’ is an empty dictionary:

```
blank = { }
```

And ‘words’ has a couple words in it:

```
words = { ‘splendid’ : ‘better than good’,
          ‘terrible’ : ‘worse than bad’ }
```

More Dictionaries

To access a value, use the dictionary and the key:

```
print words['splendid']
terrible_defined = words['terrible']
```

To assign a new value or replace an existing one:

```
words['ok'] = 'neither good nor bad'
words['splendid'] = 'fantastic!'
```

Variables can be keys too

On the last page we used a literal string as the key:

```
words['ok'] = 'neither good nor bad'
```

It is also just as legitimate to use a variable as a key:

```
my_key = 'ok'
words[my_key] = 'neither good nor bad'
```

Iterating through things

We can iterate through lists, dictionaries, and other things (but we haven't the others yet).

Say we iterate through the 'words' dictionary.

```
for foo in words:
  print "Key:", foo, "Value:", words[foo]
```

Here, 'foo' is a variable that is filled with a different key from 'words' each time through the cycle.

List/Dictionary Length

You can use built-in Python functions to do many tasks, like convert a string into a number, or a number into a string. One neat thing you can do with lists and dictionaries is get its length using the `len` function:

```
beatles = [‘John’, ‘Paul’, ‘George’, ‘Ringo’]
print “There are”, len(beatles), “Beatles.”
print “There are “ + str(len(beatles)) + “ Beatles.”
```

Note these print statements give the same output.

Functions

```
def get_cube(num):
    print “Getting cube of “ + str(num)
    cube = num * num * num
    print “Returning “ + str(cube)
    return cube
```

Often we have to do the same task many times, possibly using different data. A function performs this work for us. Functions have names (`get_cube`), accept input (`num`), perform calculations (e.g. fills in `cube`), and *very importantly*, returns values (e.g. `return cube`).

Functions

Note that the ‘return’ keyword causes the function to *stop what it is doing and bail out from the function completely.*

Functions might have more than one return function. I guarantee you that if program execution reaches one return statement, it will stop, and it definitely *not* reach any other return statement.

A note on ‘print’

In this function:

```
def get_cube(num):
  print “Getting cube of “ + str(num)
  cube = num * num * num
  print “Returning “ + str(cube)
  return cube
```

... the ‘print’ statements help us debug. But *they don’t actually change how the function works.* They aren’t involved with calculating the result, or returning it.

Python Classes

Python has many kinds of things: integers, strings, and so on. But even among integers, there are different values they can take on. So, 4 and 283 are both integers, but they are different values.

A ‘class’ is sort of like a template for building many copies of the same *kind* of special thing, and you get to define what that thing is. Like 4 and 283 are unique values among integers, we can have unique instances (a.k.a *objects*) of a class.

Defining a Class

```
class Robot:
    weight = 10
    height = 4
    num_eyes = 2
    num_death_rays = 1

r1 = Robot()
r1.height = 7
r2 = Robot()
r2.num_eyes = 20
```

At left we define a new class—a factory that we can use to make unique robots. By default, all robots have the same weight, height, number of eyes, and number of death rays.

Next, we create two instances. We customize the r1 object to be taller, and r2 to have more eyes.

Parameterizing new Objects

```
class Robot:
    weight = 10
    height = 4
    num_eyes = 2
    num_death_rays = 1

    def __init__(self, weight, height,
                 eyes, rays):
        self.weight = weight
        self.height = height
        self.num_eyes = eyes
        self.num_death_rays = rays

r1 = Robot(4, 7, 10, 2)
r2 = Robot(10, 5, 0, 10)
```

I modified the Robot class so it now has an *initialize* method.

This lets us give default values to customize our robot when we make them (at the bottom).

Object Behaviors

A method is a special kind of function that is found *inside* a class definition, and it *must* take a parameter called ‘self’ in the first slot. Like this:

```
class Bird:
   name = “Timmy”
   def fly(self, how_far):
      print self.name, “flies“, how_far, “miles”
```

`fly` is a method because it is inside the `Bird` class, and it takes `self` as input.

Object Behaviors

```
class Robot:
    num_death_rays = 1

    def __init__(self, rays):
        self.num_death_rays = rays

    def use_death_rays(self):
        print "The robot warms up its", \
            self.num_death_rays, "death rays..."
        for dr in range(self.num_death_rays):
            print "Robot fires death ray #" + str(dr)

r1 = Robot(2)
r2 = Robot(4)
r1.use_death_rays()
r2.use_death_rays()
```

‘self’ is the instance

```
def use_death_rays(self):
    print "The robot warms up its", \
        self.num_death_rays, "death rays..."
    for dr in range(self.num_death_rays):
        print "Robot fires death ray #" + str(dr)
```

```
The robot warms up its 2 death rays...
Robot fires death ray #0
Robot fires death ray #1
The robot warms up its 4 death rays...
Robot fires death ray #0
Robot fires death ray #1
Robot fires death ray #2
Robot fires death ray #3
```

‘self’ is the robot instance

```
def use_death_rays(self):
    print "The robot warms up its", \
        self.num_death_rays, "death rays..."
    for dr in range(self.num_death_rays):
        print "Robot fires death ray #" + str(dr)
```

If we want to access a variable that is associated with the robot instance we have to use “self.*something*” where “something” is whatever we need. So inside a robot method:

```
print num_death_rays # wrong!
print self.num_death_rays # right!
```

Boolean Arithmetic

Also known as “doing math with true and false”.
This is all about keywords **`or`** and **`and`**.

```
if 8 < 10 or 100 >= x:
    print “This will always print because 8 < 10.”

if 8 > 10 or 100 >= x:
    print “This will only print when x < 100.”
```

Boolean Arithmetic, p2

Also known as “doing math with true and false”.
This is all about keywords **`or`** and **`and`**.

```
if 8 < 10 and 100 >= x:
    print “This will only print when x < 100.”

if 8 > 10 and 100 >= x:
    print “This will never print because 8 < 10.”
```

Booleans can be variables

We can store the results of a boolean expression in a variable. If I am only happy when it rains *and* when it is complicated:

```
rain = false
complicated = true
is_happy = rain and complicated
print “Am I happy?” + str(is_happy)
```

It says I am not happy. Why?

Also why do I need the str() in there?