More Basics for Python

L445/L515
The Computer and Natural Language
Autumn 2011

A basic python program

```python
i = 1
while i < 10:
    print str(i) + " is my friend."
    i += 1
```

When we run this, we get the following output:

1 is my friend.
2 is my friend.
3 is my friend.
...
9 is my friend.

Control Structures

We can't really do a lot with our code yet, but we're going to want to:

- make choices (if)
- do the same thing several times (while and for)
- put a repeated part of the code in another part of the program (functions)

We'll look very briefly at if and while statements today

The if statement

For an if statement, a condition (e.g., x > 4) is evaluated to either True (run the code) or False (don't run it, or move on to the else block)

```python
if x > 4:
    print "x is greater than 4"
elif x < 4:
    print "x is less than 4"
else:
    print "Why, x must equal 4"
```

Note:
- 0 and "" (the empty string) evaluate to False
- Conditions (e.g., x < 4) can be put in parentheses, but do not need to be.

Tests

- x == y x equals y
- x < y x is less than y
- x > y x is greater than y
- x >= y x is greater than or equal to y
- x <= y x is less than or equal to y
- x != y x is not equal to y
- x is y x is the same object as y
- x is not y x is not the same object as y
- x in y x is a member of y
- x not in y x is not a member of y

- Caution: = and == are different:
  - = assigns a value
  - == compares values

The while loop

A while loop does a similar condition check, but then runs a block of code as long as that condition evaluates to True.

```python
i = 1
while i < 10:
    print str(i) + " is my friend."
    i += 1
```

- Run the block of indented code 10 times
- Each time, the variable i has a new value
Loop output

And again, here is the output we get from this code:
1 is my friend.
2 is my friend.
3 is my friend.
4 is my friend.
5 is my friend.
6 is my friend.
7 is my friend.
8 is my friend.
9 is my friend.

The code within the loop was repeated 10 times, but because i's value was different every time, the output changed.

Blocks of code

Python treats everything that is indented as a block of code.

- For while loops, this means that as long as the condition is true, the block of code is run.
- In other words, the same block of code is run multiple times ...
  ... usually with some of the variable values being different

It helps to trace your code to see this, i.e., observe what happens at every step.

Tracing a while loop

```python
i = 1
while i <= 10:
    # this indented block of code will run from i = 1 to i = 10
    print "Starting value of i: " + str(i)
    squared = i**2
    print i,
    print squared
    # remember to increment (i.e., avoid infinite loops):
    i = i + 1
    print "Ending value of i: " + str(i)
print
```

Trace output

Starting value of i: 1
1 1
Ending value of i: 2
Starting value of i: 2
2 4
Ending value of i: 3
...
Starting value of i: 10
10 100
Ending value of i: 11

Use #1: iteration

As we've just seen, while loops can be used to iterate over a sequence.

- This is most commonly done by iterating over integers, because integers easily count how many times you do something.
- You can change the way you iterate—e.g., i += 2 or i -= 1 or whatever

Use #2: until

Another, subtly different use is to perform the same actions until a certain condition is reached.

```python
user_input = ""
while len(user_input) < 10:
    user_input = raw_input("Please enter a long string: ")
print "Thank you for entering a long enough string!"
```
A common mistake: infinite loops

One common mistake when using while loops is to forget to iterate a variable, such as i

```python
i = 0
while (i < 10):
    print i
    # infinite loop b/c i is not iterated
```

If your code doesn’t stop:

- Kill it (Ctrl-C on Unix, Ctrl-Z on Windows)
- Examine your while loop and make sure the value is changing as you expect it to
  - In general, this is a good way to debug your code

Debugging (Dealing with Problems)

It is a fact that you will have errors in your code ... so, do not panic.

There are 2 main types of errors:

1. Syntax errors: these are errors which cause python not to be able to run your program, e.g.,
   - you type iff instead of if
   - you fail to indent something where you should have
   - you call a function which takes a different number of arguments
   - Python will give you an error message which gives a brief description and identifies the line number where the error occurs

2. Runtime errors: the program is well-formed python & runs, but not as intended
   - These logical errors can be much harder to track down

Debugging (cont.)

What to do:

- Track down the error:
  - Where exactly does the program break?
  - What happens if you add/remove/alter a line?
  - If you print out a variable (around the point where the program breaks), does it have the value you expect?
- Use a debugger (such as pdb), which can help answer the questions above
- Search online for what others have done with problems similar to yours
- Ask others for help in looking at your code: another set of eyes can provide a fresh perspective

The help command

```python
>>> s = 'abc'
>>> help(s.rfind)
Help on built-in function rfind:
rfind(...)
S.rfind(sub [,start [,end]]) -> int
Return the highest index in S where substring sub is found, such that sub is contained within s[start,end]. Optional arguments start and end are interpreted as in slice notation.
Return -1 on failure.
```

A “fun” exercise

Write your own while loop which only uses addition, in order to mimic multiplication

e.g., to multiply x and y:

- Start with a variable (e.g., value) set to 0
- Run a while loop y times
  - Each time, add x to the value