

Programming Basics (for Python)

Linguistics 445/515
The Computer and Natural Language
Autumn 2008

Programming

When we talk about programs and algorithms in this class, it'll help to know more about how they work.

- How do we go from describing how something works to actually making it work?
- To what level of precision does a program need to be defined?

We'll examine one programming language in particular, Python, and you'll learn:

- the basics of converting the idea of an algorithm into program code
- some fundamental concepts for writing good programs
- the basic capabilities of Python
- how to write short programs for text processing

What (not) to expect

Expect to learn:

- how to think algorithmically & improve such thinking
- how precise one has to be in writing programs

Expect not to learn:

- how to be a programmer

Important point: the programming aspects of this course are meant to supplement the core content

- Programming is not the main material of the course!

What is a program?

At an abstract level, a program is a sequence of commands, which produces an output for a given input.

Example 1:

1. Input: a text file containing all of *Ulysses*
2. Program: stuff happens (Input \mapsto Output)
3. Output: every bigram with its associated frequency

Example 2:

1. Input: your income information
2. Program: stuff happens (Input \mapsto Output)
3. Output: how much tax you have to pay

Algorithms

As mentioned, a program is basically an **algorithm**, i.e., a sequence of commands.

Here's what a sketch of an algorithm for printing out a text's unigrams might look like:

1. Read in each word from the text
 - (a) Store each word
 - (b) Add to the count of each word, storing (word,count) pairs in some storage device
2. Read through the storage device
 - (a) Print each word with its count

But how do we “read in” something or “store” things?

- That's where different programming languages differ.

Programming Languages

Programming languages share a lot in common:

- They require you to use explicit syntax. Some examples:
 - Only well-defined functions can be used, i.e., you need to know what things are/aren't allowed by a programming language
 - * `exec` is a legitimate command in Python
 - * `evac` is not a legitimate command
 - The language forces you to follow particular formats
 - * In Python, you have to indent within a `for` loop
 - * In Perl, you have to enclose the contents of a loop within brackets.

But the languages differ in the specifics of the syntax

- Luckily, it's very easy to get a small program working in Python.

Python

So, now we're ready to start investigating Python ... Why Python?

- It's quick: It is very good for writing short scripts and for text processing.
 - Python is somewhat of a “cousin” to Perl in those respects.
- It's powerful: At the same time, Python has much support for turning small programs into much larger projects (such as object-oriented programming)
- It's easy: Function names are (arguably) rather transparent in Python.
- It's free and available across systems (code is generally portable across platforms)
- It's marketable: organizations like Google, Pixar, and the NSA use Python

Resources on Python

Books:

- *Learning Python* by Mark Lutz
- *Beginning Python: From Novice to Professional* by Magnus Lie Hetlund
- *Think Python* by Allen Downey (freely available online): <http://www.greenteapress.com/thinkpython/thinkpython.pdf>
- *Dive Into Python* by Mark Pilgrim (also available online; for experienced programmers): <http://www.diveintopython.org/>

Online resources:

- Guido van Rossum's Python Tutorial: <http://www.python.org/doc/current/tut/>
- Python-forum.org: <http://python-forum.org/pythonforum/index.php>
- Or use your favorite search engine to find more about a particular point ...

Using a command line

Let's step back from Python for just one second and talk about using a command line

Instead of navigating through your files by clicking on things (in Windows or the Aqua interface on Macs), you can navigate by typing

- Windows: open a Command Prompt
 - Start → Programs → Accessories → Command Prompt
- Mac: open a Terminal
 - Applications → Utilities → Terminal

See the contents of a directory:

- Windows: `dir`
- Mac (Unix): `ls`

Moving around the terminal

The important command for us (on both platforms) is `cd` (“change directory”), since we’ll have to get to the directory which contains our python files.

- `cd courses/08/445/` puts me into the directory `courses/08/45/` instead of my home directory
- Note on Windows that directories can have spaces in their names—in those situations, use double quotes (“): `cd "Documents and Settings\md7\Desktop\"`
- Also note that slashes go in opposite directions, depending on the platform.

How to avoid repetitive typing:

- Hitting the tab key will complete what you’re typing
- Hitting the up arrow will bring up the previous command(s) you typed

How to use Python

You can run python either at the command-line or from a file:

- Interactive: simply type `python` (or `python2.3` or `python2.5`) at the command line, and this will open up a session with python
 - Interactive sessions are very useful for practicing and testing out bits of code.
 - Note that variable values are automatically printed out (not true of files).
- Files: more often, you will want to write a program and call that program
 - This allows you to edit freely.
 - If your program's name is `program.py`, you will type `python program.py`
 - * Both Windows and Mac/Unix allow you to *redirect* your output to a file, if you want to look at it later.
 - * `python program.py > output.txt` stores the output in `output.txt` (in the same directory)

A basic python program

Here's a very basic python program, which will give you a flavor of python (but don't worry too much about how it works yet).

```
i = 1
while i < 10:
    # comments can be put after '#'
    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.
```

Creating/Editing Python files

How do we create such files, though? Where do they come from?

Python files are simply text files, so we just need a text editor. Some options:

- Windows: Notepad or Wordpad → Save as plain text
 - Sometimes Windows is set up s.t. it forces you to add a `.txt` extension to your file.
 - This isn't a problem, but to get rid of it, (I think) you need to save as "All files" and also change your desktop settings so that they show file extensions
- Mac: TextEdit → Under *Preferences*, be sure "Plain Text" is checked for Format
- Unix: pico, Emacs (or Aquamacs [which I use]), Vim, and probably others

IDLE

Some text editors offer **syntax highlighting**, which shows you variable names, indentation, etc. and can make coding much easier

There are also **Integrated Development Environments (IDEs)** which offer syntax highlighting, debugging features, streamlined code-running, etc.

- One IDE which comes with Python is IDLE (<http://www.python.org/idle/doc/idlemain.html>)
 - Windows: Once you've installed Python, this should be available from Start → Applications → Python25 → ...
 - Mac: This may or may not already be installed. For me, I did the following:
 - * Opened up a terminal
 - * Typed: `cd /System/Library/Frameworks/Python.framework/Versions/2.5/lib/python2.5/idlelib/`
 - * Typed: `python idle.py`

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

Obtaining Python

- The latest python is available for different platforms at: <http://www.python.org/download/>
- Mac: It should be pre-installed. Type `python` at a terminal to check.

Some notes for Windows users:

- On Windows it may not appear as if Python is installed: it could be installed, but it's only available in the directory where it was downloaded.
- To handle this, you can:
 - work in the directory where Python was installed
 - include the full path of Python when you run your programs, e.g.,
`C:\Python25\python program.py`
 - change the environment variable `PATH` (check under “Control Panel”) to include `C:\Python25`, so the Command Prompt can find `python` from any directory

Data Types

Every programming language has certain basic **types**, which are the building blocks from which everything else is built. In Python, some core data types are:

- Simple Types: numbers and strings ... more on these in a minute
 - numbers: 3, 12.443, 89, ...
 - strings: "hello", 'manny', "34", ...
- Complex Types: lists and dictionaries (& sets & tuples) ... covered another week
 - lists: [1,2,3], [1,2," a"], ["john", "george", "paul", "ringo"], ...
 - dictionaries: {"a":1, "b":16}, ...

Python is **dynamically typed**: you do not have to declare what type each variable is

Numbers

```
>>> 2+2
```

```
4
```

```
>>> 3/2
```

```
1
```

```
>>> 3/2.
```

```
1.5
```

Python has integers and floating point numbers (& complex numbers), and operations to convert between them:

```
>>> float(3)
```

```
3.0
```

```
>>> int(4.123)
```

```
4
```

You can print floating point numbers with different levels of precision, but we won't cover that here.

More on math

We'll talk more about **modules** later, but essentially they allow you to have additional functionality in your program

- The math module:

```
>>> import math
>>> math.sqrt(5)
2.2360679774997898
```

- The random module:

```
>>> import random
>>> random.choice([1,2,3,4,5,6,7,8,9,10])
8
>>> random.choice([1,2,3,4,5,6,7,8,9,10])
7
```

Variables

A variable stores some value for later use

- A variable can be almost any sequence of alphabetic characters (the underscore and digits can be in there, too, as long as they're not the first character of the variable)
- The only other constraint is that a variable cannot have the same name as a function in python. Bad variable names, thus, include: `for`, `in`, `class`, etc.
- It helps to give mnemonic names to variables (e.g., `name` preferred over `a`)

```
counter = 1  
name = "john"
```

Strings

- Many ways to write a string:
 - single quotes: 'string'
 - double quotes: "string"
 - can also use """ to write strings over multiple lines:

```
>>> """<html>
... <body>
... something
... </body>
... </html>
... """
'<html>\n<body>\nsomething\n</body>\n</html>\n'
```
- There are string characters with special meaning: e.g., \n (newline) and \t (tab)
- Get the length of a string by the len function

String indices & slices

You can use slices to get a part of a string

```
>>> s = "happy"
>>> len(s) # use the len function
5
>>> s[3] # indexed from 0, so 4th character
'p'
>>> s[1:3] # characters 1 and 2
'ap'
>>> s[:3] # first 3 characters
'hap'
>>> s[3:] # everything except first 3 characters
'py'
>>> s[-4] # 4th character from the back
'a'
```

String operations

- Concatenate strings with the + operator, reduplicate them with *

```
s = "happy" + "joy"    # s = "happyjoy"  
s = "happy"*3         # s = "happyhappyhappy"
```

- Convert to upper or lower case: upper / lower

```
>>> s = 'abcdefg'  
>>> s.upper()  
'ABCDEFG'  
>>> s  
'abcdefg'
```

String operations (2)

- The `strip` operator can be used to remove white space around the string.

```
>>> s = ' agbg\t'  
>>> s.strip()  
'agbg'
```

- The `split` operator take a string and converts it to a list, splitting it on whitespace (although, you can change this to split on a different delimiter)

```
>>> s = 'here is a toy sentence'  
>>> s.split()  
['here', 'is', 'a', 'toy', 'sentence']
```

`find` and `replace` are also useful operations for manipulating strings

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)

```
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.

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`.

This goes back to an example we saw earlier:

```
i = 1
while i < 10:
    # comments can be put after '#'
    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.

A common mistake: infinite loops

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

```
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 (see debugging discussion above)

Function Names and Feeling Lost

We know about some basic commands, like `print`, but what else is there?

Things to remember:

- There is always something you won't know!
 - If you're curious as to what's available for a particular data type, try typing `dir(s)` for a variable `s` ... This will list all possible methods
- Lots of things have been pre-written, so look around for functions you want to use
 - For example, I don't know how to reverse a string, but I feel like someone will have done that before.
- Familiarize yourself with the documentation
 - See <http://www.python.org/doc/>

The help command

```
>>> 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.

In-class exercise

Let's write a program that takes two strings and says what their shared prefix is, if any. e.g., *bring* and *behold* have a shared prefix *b*

- We'll sketch out the algorithm together, and write the skeleton of it together.
- You'll work for 15 minutes or so on your own/in pairs.
- We'll finish it up together.

Some useful concepts:

- `==` is the way to test if two strings or characters are equal, and `!=` to test if they are not equal.
- `break` exits you from a loop