Python Basics

The Crash Course

If you choose, you can hold a conversation with the Python interpreter, where you speak in expressions and it replies with evaluations. There's clearly a **read**-**eval**-**print** loop going on just as there is in the Kawa environment.

```
bash-3.2$ python
Python 2.5.1 (r251:54863, Oct  5 2007, 21:08:09)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 4 + 15
19
>>> 8 / 2 * 7
28
>>> x = 12
>>> x ** 2
144
>>> y = 9 + 7 * x
>>> y
93
>>> ^D
bash-3.2$
```

Unlike purely functional languages, Python doesn't require that every single expression print a result, which is why you don't see anything hit the console in response to an assignment statement. The above examples involve just whole numbers, and much of what you expect to be available actually is. There's even built-in exponentiation with `**`, though `++` and `--` aren't included. To launch interactive mode, you type **`python`** on the command line, talk for a while, and then type **`Control-D`** when you're done to exit.

Booleans

The Boolean constants are **`True`** and **`False`**, and the six relational operators work on all primitives, including strings. `!`, `||`, and **`&&`** have been replaced by the more expressive **`not`**, **`or`**, and **`and`**. Oh, and you can chain relational tests—things like **`min < mean < max`** make perfect sense.

```
>>> 4 > 0
True
>>> "apple" == "bear"
False
>>> "apple" < "bear" < "candy cane" < "dill"
True
>>> x = y = 7
>>> x <= y and y <= x
True
>>> not x >= y
False
```

Whole Numbers

Integers work as you'd expect, though you're insulated almost entirely from the fact that small numbers exist as four-byte figures and super big numbers are managed as longs, without the memory limits:

```
>>> 1 * -2 * 3 * -4 * 5 * -6
-720
>>> factorial(6)
720
>>> factorial(5)
120
>>> factorial(10)
3628800
>>> factorial(15)
1307674368000L
>>> factorial(40)
815915283247897734345611269596115894272000000000L
```

When the number is big, you're reminded how big by the big fat **`L`** at the end. (I defined the **`factorial`** function myself, because it's not a built-in. We'll start defining functions shortly.)

Strings

String constants can be delimited using either double or single quotes. Substring selection, concatenation, and repetition are all supported.

```
>>> interjection = "ohplease"
>>> interjection[2:6]
'plea'
>>> interjection[4:]
'ease'
>>> interjection[:2]
'oh'
>>> interjection[:]
'ohplease'
>>> interjection * 4
'ohpleaseohpleaseohpleaseohplease'
>>> oldmaidsays = "pickme" + interjection * 3
>>> oldmaidsays
'pickmeohpleaseohpleaseohplease'
>>> 'abcdefghijklmnop'[-5:] # negative indices count from the end!
'lmnop'
```

The quirky syntax that's likely new to you is the slicing, ala **`[start:stop]`**. The **`[2:6]`** identifies the substring of interest: character data from position 2 up through but not including position 6. Leave out the start index and it's taken to be 0. Leave out the stop index, it's the full string length. Leave them both out, and you get the whole string. (Python doesn't burden us with a separate character type. We just use one-character strings where we'd normally use a character, and everything works just swell.)

Strings are really objects, and there are good number of methods. Rather than exhaustively document them here, I'll just illustrate how some of them work. In general, you should expect the set of methods to more or less imitate what strings in other object-oriented languages do. You can expect methods like **`find`**, **`startswith`**, **`endswith`**, **`replace`**, and so forth, because a string class would be a pretty dumb string class without them. Python's string provides a bunch of additional methods that make it all the more useful in scripting and WWW capacities—methods like **`capitalize`**, **`split`**, **`join`**, **`expandtabs`**, and **`encode`**. Here's are some examples:

```
>>> 'abcdefghij'.find('ef')
4
>>> 'abcdefghij'.find('ijk')
-1
>>> 'yodelady-yodelo'.count('y')
3
>>> 'google'.endswith('ggle')
False
>>> 'lItTle ThIrTeEn YeAr OlD gIrl'.capitalize()
'Little thirteen year old girl'
>>>
>>> 'Spiderman 3'.istitle()
True
>>> '1234567890'.isdigit()
True
>>> '12345aeiuo'.isdigit()
False
>>> '12345abcde'.isalnum()
True
>>> 'sad'.replace('s', 'gl')
'glad'
>>> 'This is a test.'.split(' ')
['This', 'is', 'a', 'test.']
>>> '-'.join(['ee','eye','ee','eye','oh'])
'ee-eye-ee-eye-oh'
```

Lists and Tuples

Python has two types of sequential containers: lists (which are read-write) and tuples (which are immutable, read-only). Lists are delimited by square brackets, whereas tuples are delimited by parentheses. Here are some examples:

```
>>> streets = ["Castro", "Noe", "Sanchez", "Church",
               "Dolores", "Van Ness", "Folsom"]
>>> streets[0]
'Castro'
>>> streets[5]
'Van Ness'
>>> len(streets)
7
>>> streets[len(streets) - 1]
'Folsom'
```

The same slicing that was available to us with strings actually works with lists too:

```
>>> streets[1:6]
['Noe', 'Sanchez', 'Church', 'Dolores', 'Van Ness']
>>> streets[:2]
['Castro', 'Noe']
>>> streets[5:5]
[]
```

Coolest feature ever: you can splice into the middle of a list by identifying the slice that should be replaced:

```
>>> streets
['Castro', 'Noe', 'Sanchez', 'Church', 'Dolores', 'Van Ness', 'Folsom']
>>> streets[5:5] = ["Guerrero", "Valencia", "Mission"]
>>> streets
['Castro', 'Noe', 'Sanchez', 'Church', 'Dolores', 'Guerrero',
 'Valencia', 'Mission', 'Van Ness', 'Folsom']
>>> streets[0:1] = ["Eureka", "Collingswood", "Castro"]
>>> streets
['Eureka', 'Collingswood', 'Castro', 'Noe', 'Sanchez', 'Church',
 'Dolores', 'Guerrero', 'Valencia', 'Mission', 'Van Ness', 'Folsom']
>>> streets.append("Harrison")
>>> streets
['Eureka', 'Collingswood', 'Castro', 'Noe', 'Sanchez', 'Church',
 'Dolores', 'Guerrero', 'Valencia', 'Mission', 'Van Ness', 'Folsom', 'Harrison']
```

The first splice states that the empty region between items `5` and `6`—or in `[5, 5)`, in interval notation—should be replaced with the list constant on the right hand side. The second splice states that `streets[0:1]`—which is the sublist `['Castro']`—should be overwritten with the sequence `['Eureka', 'Collingswood', 'Castro']`. And naturally there's an `append` method.

Note: lists need not be homogenous. If you want, you can model a record using a list, provided you remember what slot stores what data.

```
>>> prop = ["355 Noe Street", 3, 1.5, 2460,
            [[1988, 385000],[2004, 1380000]]]
>>> print("The house at %s was built in %d." % (prop[0], prop[4][0][0])
The house at 355 Noe Street was built in 1988.
```

The list's more conservative brother is the tuple, which is more or less an immutable list constant that's delimited by parentheses instead of square brackets. It's supports read-only slicing, but no clever insertions:

```
>>> cto = ("Will Shulman", 154000, "BSCS Stanford, 1997")
>>> cto[0]
'Will Shulman'
>>> cto[2]
'BSCS Stanford, 1997'
>>> cto[1:2]
(154000,)
>>> cto[0:2]
('Will Shulman', 154000)
>>> cto[1:2] = 158000
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support slice assignment
```

Defining Functions

In practice, I'd say that Python walks the fence between the procedural and object-oriented paradigms. Here's an implementation of a standalone **`gatherDivisors`** function. This illustrates **`if`** tests, **`for`**-loop iteration, and most importantly, the dependence on white space and indentation to specify block structure:

```
# Function: gatherDivisors
# ------------------------
# Accepts the specified number and produces
# a list of all numbers that divide evenly
# into it.

def gatherDivisors(num):
    """Synthesizes a list of all the positive numbers
    that evenly divide into the specified num."""
    divisors = []
    for d in xrange(1, num/2 + 1):
        if (num % d == 0):
            divisors.append(d)
    return divisors
```

The syntax takes some getting used to. We don't really miss the semicolons (and they're often ignored if you put them in by mistake). You'll notice that certain parts of the implementation are indented one, two, even three times. The indentation (which comes in the form of either a tab or four space characters) makes it clear who owns whom. You'll notice that **`def`**, **`for`**, and **`if`** statements are punctuated by colons: this means at least one statement and possibly many will fall under the its jurisdiction.

Note the following:

- The **`#`** marks everything from it to the end of the line as a comment. I bet you figured that out already.
- **None** of the variables—neither parameters nor locals—**are strongly typed**. Of course, Python supports the notion of numbers, floating points, strings, and so forth. But it doesn't require you state why type of data need be stored in any particular variable. Identifiers can be bound to any type of data at any time, and it needn't be associated with the same type of data forever. Although there's rarely a good reason to do this, a variable called **`data`** could be set to **`5`**, and reassigned to **`"five"`**, and later reassigned to **`[5, "five", 5, [5]]`** and Python would approve.
- The triply double-quote delimited string is understood to be a string constant that's allowed to span multiple lines. In particular, if a string constant is the first expression within a **`def`**, it's taken to be a documentation string explanation the function to the client. It's not designed to be an implementation comment—just a user comment so they know what it does.
- The **`for`** loop is different than it is in other language. Rather than counting a specific numbers of times, **`for`** loops iterate over what are called **iterables**. The iterator

(which in the **gatherDivisors** function is **d**) is bound to each element within the iterable until it's seen every one. Iterables take on several forms, but the list is probably the most common. We can also iterate over strings, over sequences (which are read-only lists, really), and over dictionaries (which are Python's version of the C++ **hash_map**)

Packaging Code In Modules

Once you're solving a problem that's large enough to require procedural decomposition, you'll want to place the implementations of functions in files—files that operate either as modules (sort of like Java packages, C++ libraries, etc) or as scripts.

This **gatherDivisors** function above might be packaged up in a file called **divisors.py**. If so, and you launch **python** from the directory storing the **divisors.py** file, then you can import the **divisors** module, and you can even import actual functions from within the module. Look here:

```
bash-3.2$ python
Python 2.5.1 (r251:54863, Oct  5 2007, 21:08:09)
[GCC 4.0.1 (Apple Inc. build 5465)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import divisors
>>> divisors.gatherDivisors(54)
[1, 2, 3, 6, 9, 18, 27]
>>> gatherDivisors(216)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
NameError: name 'gatherDivisors' is not defined
>>> from divisors import gatherDivisors
>>> gatherDivisors(216)
[1, 2, 3, 4, 6, 8, 9, 12, 18, 24, 27, 36, 54, 72, 108]
>>> "neat"
'neat'
```

If everything you write is designed to be run as a standalone script—in other words, an independent interpreted program—then you can bundle the collection of meaningful functions into a single file, save the file, and mark the file as something that's executable (i.e. **chmod a+x narcissist.py**).

Here's a fairly involved program that prints out the first 15 (or some user-supplied number of) narcissistic numbers (just Google narcissistic numbers if you miss the in class explanation):

```
#!/usr/bin/env python
# encoding: utf-8
# Here's a simple script (feels like a program, though) that prints out
# the first n narcissistic numbers, where n is provided on the command line.
import sys

def numDigits(num):
    """Returns the number of digits making
    up a number, not counting leading zeroes,
    except for the number 0 itself."""
    if (num == 0): return 1
    digitCount = 0
    while (num > 0):
        digitCount += 1
        num /= 10
    return digitCount

def isNarcissistic(num):
    """Returns True if and only if the
    number is a narcissistic number."""
    originalNum = num
    total = 0
    exp = numDigits(num)
    while (num > 0):
        digit = num % 10
        num /= 10
        total += digit ** exp
    return total == originalNum

def listNarcissisticNumbers(numNeeded):
    """Searches for an prints out the first 'numNeeded'
    narcissistic numbers."""
    numFound = 0;
    numToConsider = 0;
    print "Here are the first %d narcissistic numbers." % numNeeded
    while (numFound < numNeeded):
        if (isNarcissistic(numToConsider)):
            numFound += 1
            print numToConsider
        numToConsider += 1
    print "Done!"

def getNumberNeeded():
    """Parses the command line arguments to the extent necessary to determine
    how many narcissistic numbers the user would like to print."""
    numNeeded = 15;    # this is the default number
    if len(sys.argv) > 1:
        try:
            numNeeded = int(sys.argv[1])
        except ValueError:
            print "Non-integral argument encountered... using default."
    return numNeeded

listNarcissisticNumbers(getNumberNeeded())
```

The slash-bang is usually the first line of a script, and it tells us what environment to run the script in. The encoding thing is optional, but standard.

Required so that we can parse the command line via variables defined by the sys module.

One-liner slave expressions can be on the same line as their owner, like this.

The equivalent of System.out.println, but with printf's substitution strategy. The exposed % marks the beginning of the expressions that should fill in the %d and %s placeholders.

No ++ ☹

An exposed function call, which gets evaluated as the script runs. This is effectively your main program, except you get to name your top-level function in Python.

View the script on the previous page as a module with five expressions. The first four are **`def`** expressions—function definitions—that when evaluated have the side effect of binding the name of the function to some code. The fifth expression is really a function call whose evaluation generates the output we're interested in. It relies on the fact that the four expressions that preceded it were evaluated beforehand, so that by the time the Python environment gets around to the **`listNarcissisticNumbers`** call, **`listNarcissisticNumbers`** and **`getNumbersNeeded`** actually mean something and there's code to jump to.

Quicksort and List Comprehensions

Here's an implementation of a familiar sorting algorithm that illustrates an in-place list initialization technique:

```
# Illustrates how list slicing, list concatentation, and list
# comprehensions work to do something meaningful.
# This is not the most efficient version of quicksort available, because
# each level requires two passes instead of just one.

def quicksort(sequence):
    """Classic implementation of quicksort using list
    comprehensions and assuming the traditional relational
    operators work.  The primary weakness of this particular
    implementation of quicksort is that it makes two passes
    over the sequence instead of just one."""

    if (len(sequence) == 0): return sequence
    front = quicksort([le for le in sequence[1:] if le <= sequence[0]])
    back = quicksort([gt for gt in sequence[1:] if gt > sequence[0]])
    return front + [sequence[0]] + back

>>> from quicksort import quicksort
>>> quicksort([5, 3, 6, 1, 2, 9])
[1, 2, 3, 5, 6, 9]
>>> quicksort(["g", "b", "z", "k", "e", "a", "y", "s"])
['a', 'b', 'e', 'g', 'k', 's', 'y', 'z']
```

The **`[le for le in sequence[1:] if le <= sequence[0]]`** passed to the first recursive call is called a **list comprehension**, which is a quick, one line way to create one list out of another piece of data. You can include an arbitrary number of iterations in a list comprehension, as with:

```
>>> [(x, y) for x in xrange(1, 3) for y in xrange(4, 8)]
[(1, 4), (1, 5), (1, 6), (1, 7), (2, 4), (2, 5), (2, 6), (2, 7)]
>>> [(x, y, z) for x in range(1, 5)
                for y in range(1, 5)
                for z in range(1, 6) if x < y <= z]
[(1, 2, 2), (1, 2, 3), (1, 2, 4), (1, 2, 5), (1, 3, 3), (1, 3, 4),
 (1, 3, 5), (1, 4, 4), (1, 4, 5), (2, 3, 3), (2, 3, 4), (2, 3, 5),
 (2, 4, 4), (2, 4, 5), (3, 4, 4), (3, 4, 5)]
```

Here's a more serious script that starts to approximate the functionality of the Unix **`find`** routine. Our **`find`**, when given a directory name and a file name, searches through the

entire file system at and below the specified directory and lists the full paths of all files with the specified name.

```
#!/usr/bin/env python
# encoding: utf-8
#
# Simple imitation of the Unix find command, which search sub-tree
# for a named file, both of which are specified as arguments.
# Because python is a scripting language, and because python
# offers such fantastic support for file system navigation and
# regular expressions, python is very good at for these types
# of tasks

from sys import argv
from os import listdir
from os.path import isdir, exists, basename, join

def listAllExactMatches(path, filename):
    """Recursive function that lists all files matching
    the specified file name"""
    if (basename(path) == filename):
        print "%s" % path
    if (not isdir(path)):
        return
    dirlist = listdir(path)
    for file in dirlist:
        listAllExactMatches(join(path, file), filename)

def parseAndListMatches():
    """Parses the command line, confirming that there are in fact
    three arguments, confirms that the specified path is actually
    the name of a real directory, and then recursively searches
    the file tree rooted at the specified directory."""
    if (len(argv) != 3):
        print "Usage: find <path-to-directory-tree> <filename>"
        return
    directory = argv[1]
    if (not exists(directory)):
        print "Specified path \"%s\" does not exist." % directory
        return;
    if (not isdir(directory)):
        print "\"%s\" exists, but doesn't name an actual directory." % directory
    filename = argv[2]
    listAllExactMatches(directory, filename)

parseAndListMatches()
```

Here's a list of all of the **rss-news-search.c** files I got as Assignment 4 submissions last spring quarter:

```
jerry> find.py /usr/class/cs107/submissions/hw4/Jerry/ rss-news-search.c
/usr/class/cs107/submissions/hw4/Jerry/izaak-1/rss-news-search.c
/usr/class/cs107/submissions/hw4/Jerry/ajlin-1/rss-news-search.c
/usr/class/cs107/submissions/hw4/Jerry/taijin-1/rss-news-search.c
/usr/class/cs107/submissions/hw4/Jerry/sholbert-1/rss-news-search.c
/usr/class/cs107/submissions/hw4/Jerry/hmooers-1/rss-news-search.c
/usr/class/cs107/submissions/hw4/Jerry/msmissyw-1/rss-news-search.c
/usr/class/cs107/submissions/hw4/Jerry/jorelman-1/rss-news-search.c
/usr/class/cs107/submissions/hw4/Jerry/jdlong-1/rss-news-search.c
```

The **from**/**import** statements should tell you where the new functions are coming from. Most of the functions are self-explanatory, and you can intuit what the others must do based on what you know the script is trying to accomplish.

- **basename** returns the last part of a path:
  - **random.c** from **/Users/jerry/code/rsg/random.c**
  - **Makefile** from **/usr/class/cs107/assignments/assn-1-rsg/Makefile**
  - **usr** from **/usr**

- **join** returns the concatenation of one or more paths using the path separator appropriate from the operating system. If any of the paths are absolute, then all previous paths are ignored.
  - **join("/usr/class/cs107", "bin", "submit")** returns **"/usr/class/cs107/bin/submit"**
  - **join("/usr/ccs/", "/usr/bin", "ps")** returns **"/usr/bin/ps"**

- The other functions from the **os** and **os.path** modules should be self-explanatory.

Why am I including this script? Because this is the type of things that scripts do, and because Python, with its support for file system navigation and regular expression matching, is perfect for this type of thing.

Pulling and Navigating XML content

```
#!/usr/bin/env python
# encoding: utf-8

from xml.dom import *
from xml.dom.minidom import parse
from urllib2 import urlopen
from sys import argv

# Overall program illustrates the glorious support
# Python has for XML.  The xml.dom.minidom module
# provides the parse method, which knows how to
# pull XML content through an open internet connection
# and build an in-memory, tree version of the document.
# The full xml.dom package is what defines the Document
```

```
# class and all of the helper classes to model a XML
# document as a tree.

def listAllArticles(rssURL):
   """Lists all of the titles of the articles identified
   by the specified feed"""
   conn = urlopen(rssURL)
   xmldoc = parse(conn)
   items = xmldoc.getElementsByTagName("item")
   for item in items:
      titles = item.getElementsByTagName("title")
      title = titles[0].childNodes[0].nodeValue
      print("Article Title: %s" % title.encode('utf-8'))

def extractFeedName():
   """Pulls the URL from the command line if there is one, but
   otherwise uses a default."""
   defaultFeedURL = "http://feeds.chicagotribune.com/chicagotribune/news/"
   feedURL = defaultFeedURL
   if (len(argv) == 2):
      feedURL = argv[1]
   return feedURL

listAllArticles(extractFeedName())
```

Why am I including this particular script? Because Python's library set is modern and sophisticated enough that current-day web technology needs—things like HTTP, XML, SOAP, SMTP, and FTP—are supported by the language. Assignment 4 and 6 required two full **`.h`** and **`.c`** files to manage **`URL`**s and **`URLConnection`**s. Python takes care of all that with **`urlopen`**.

Dictionaries

We know enough to start talking about Python's Holy Grail of data structures: the dictionary. The Python dictionary is little more than a hash table, where the keys are strings and the values are anything we want. Here's the interactive build up of a single dictionary instance modeling the house I grew up in:

```
>>> primaryHome = {} # initialize empty dictionary, add stuff line by line
>>> primaryHome["phone"] = "609-786-06xx"
>>> primaryHome["house-type"] = "rancher"
>>> primaryHome["address"] = {}
>>> primaryHome["address"]["number"] = 2210
>>> primaryHome["address"]["street"] = "Hope Lane"
>>> primaryHome["address"]["city"] = "Cinnaminson"
>>> primaryHome["address"]["state"] = "New Jersey"
>>> primaryHome["address"]["zip"] = "08077"
>>> primaryHome["num-bedrooms"] = 3
>>> primaryHome["num-bathrooms"] = 1.5
>>> primaryHome
{'num-bathrooms': 1.5, 'phone': '609-786-06xx', 'num-bedrooms': 3, 'house-
type': 'rancher', 'address': {'city': 'Cinnaminson', 'state': 'New Jersey',
'street': 'Hope Lane', 'number': 2210, 'zip': '08077'}}
>>> primaryHome["address"]["street"]
'Hope Lane'
```

You can think of this as some method-free object that's been populated with a bunch of properties. Although, building up a dictionary like this needn't be so tedious. If I wanted, I could initialize a second dictionary by typing out the full text representation of a dictionary constant:

```
>>> vacationHome = {'phone': '717-581-44yy', 'address': {'city': 'Jim
Thorpe', 'state': 'Pennsylvania', 'number': 146, 'street':'Fawn Drive',
'zip': '18229'}}
>>> vacationHome["address"]["city"]
'Jim Thorpe'
```

Usually the dictionaries are built up programmatically rather than by hand. But before we go programmatic, I can show you what RSG looks like in a Python setting, where the grammar is hard coded into the file as a dictionary:

```
#!/usr/bin/env python
# encoding: utf-8
# Script that generates three random sentences from the
# hard-code grammar.  In general, the grammar would be
# stored in a data file, but to be honest, it would likely
# be encoded as a serialized dictionary, since that would make it
# trivial to deserialize the data.

import sys # for sys.stdout
from random import seed, choice

grammar = { '<start>':[['The ', '<object>', ' ', '<verb>', ' tonight.']],
            '<object>':[['waves'], ['big yellow flowers'], ['slugs']],
            '<verb>':[['sigh ', '<adverb>'], ['portend like ', '<object>']],
            '<adverb>':[['warily'], ['grumpily']] }

# Expands the specified symbol into a string of
# terminals.  If the symbol is already a terminal,
# then we just print it as is to sys.stdout.  Otherwise,
# we look the symbol up in the grammar, choose a random
# expansion, and map the expand function over it.
#
# Note the mapping functionality that comes with map.
# Scheme has its influences, people!

def expand(symbol):
    if symbol.startswith('<'):
        options = grammar[symbol]
        production = choice(options)
        map(expand, production)
    else:
        sys.stdout.write(symbol)

# Seeds the randomization engine, and the
# procceds to generate three random sentences.
# We use sys.stdout so we have a little more
# control over formatting.

def generateRandomSentences(numSentences):
    seed()
    for iteration in range(numSentences):
        sys.stdout.write(str(iteration + 1));
```

Here I'm spelling out a dictionary literal, which maps strings to sequences of string sequences. Note I implant the white space needs directly into the expression of the full grammar.

The choice built-in (from the random module) takes a sequence and returns a randomly selected element, where all elements are equally likely to be chosen. This is how we generate a random production.

*Look, mom! **Mapping**!*

```
        sys.stdout.write('.) ')
        expand('<start>')
        sys.stdout.write('\n')

generateRandomSentences(3)
```

And here are the test runs:

```
bash-3.2$ rsg.py
1.) The slugs portend like waves tonight.
2.) The big yellow flowers portend like big yellow flowers tonight.
3.) The big yellow flowers portend like waves tonight.
bash-3.2$ rsg.py
1.) The big yellow flowers sigh warily tonight.
2.) The slugs sigh grumpily tonight.
3.) The slugs portend like big yellow flowers tonight.
bash-3.2$ rsg.py
1.) The waves portend like waves tonight.
2.) The big yellow flowers sigh grumpily tonight.
3.) The slugs sigh warily tonight.
```

Defining Objects

Here's a simple **lexicon** class definition:

The __init__ method is the Python equivalent of a constructor. It's optional, but since there's typically at least one attribute that needs to be initialized (else why do we have a class?), it's unusual to not have a constructor for any meaningful object

```
from bisect import bisect
class lexicon:
    def __init__(self, filename = 'words'):
        """Constructs a raw lexicon by reading in the
        presumably alphabetized list of words in the
        specified file.  No error checking is performed
        on the file, though."""
        infile = open(filename, 'r')
        words = infile.readlines() # retains newlines
        self.__words = map(lambda w: w.rstrip(), words)

    def containsWord(self, word):
        """Implements traditional binary search on the
        lexicon to see if the specified word is present."""
        return self.words[bisect(self.words, word) - 1] == word
```

Here's a normal method. Note that all methods (and the special __init__ method) all take an exposed self pointer.

```
    def wordContainsEverything(self, word, characterset):
        """Returns True if and only if the specified word
        contains every single character in the specified
        character set."""
        for i in range(len(characterset)):
            if (word.find(characterset[i]) < 0):
                return False
        return True

    def listAllWordsContaining(self, characterset):
        """Brute force lists all of the words in the lexicon that
        contain each and every character in the character set."""
        matchingWords = []
        for word in self.words:
            if (self.wordContainsEverything(word, characterset)):
                matchingWords.append(word)

        if (len(matchingWords) == 0):
            print "We didn't find any words that contained all those characters."
            print "Try a less constraining character set."
            return

        print "Listing all words with the letters \"%s\"" % characterset
        print ""
        for word in matchingWords:
            print "\t%s" % word
        print ""
```

Source: http://see.stanford.edu/materials/icsppcs107/35-Python-Basics.pdf