A Quick Introduction to Python

Danny Yoo (dyoo@hkn.eecs.berkeley.edu)

March 15, 2001

1 Introduction

Hello! This is a small introduction for Python, a high level, programmer-friendly language. I'll assume that you've programmed in another languages like C, Java, or Scheme, in which case, I hope you will be pleasantly surprised by Python's elegance! If not, it's still my hope that most of this stuff will make sense for you.

So... Python! If you're trying to prototype a program or write a useful script, you may find Python one of the best tools for your job. More importantly, I think it's great fun to write Python programs. This handout will try to be a gentle but brief introduction to the Python language, so it might be a little slow for some, unfulfilling for others. I'm also definitely skipping some details here and there, so if you'd like a more detailed introduction, you may be interested in the official Python tutorial at `http://www.python.org/tut/index.html`, which is a dive off the deep end of the pool, but is still holy writ. There are also many good introductions to Python at `http://www.python.org/Intros`.

Let's begin to explore Python.

2 Running Python

The first question you might ask is, "How do I run Python?" On almost all Unix computers, we can usually type `python` at our prompt. Here's what it looks like on my machine:

```
dyoo@einfall:~/work/python$ python
Python 2.1b1 (#1, Mar  4 2001, 22:57:21) [GCC 2.95.2 20000220 (Debian GNU/Linux)] on linux2
Type "copyright", "credits" or "license" for more information.
>>>
```

We're brought to an interactive interpreter prompt which lets us type in Python statements. For example:

```
>>> print "Hello World"
Hello World
```

For the majority of this introduction, I'll introduce new features by trying them out on the interactive interpreter. The interpreter itself is an invaluable tool for debugging and exploring our Python programs. Normally, however, we'd write a prepared file of Python statements into a file, say, for example, `helloworld.py`. If we do this, we can run our Python "script" by feeding it as an argument to Python:

```
dyoo@einfall:~/work/python$ cat helloworld.py
print "Hello World"
dyoo@einfall:~/work/python$ python helloworld.py
Hello World
```

and if we really want to make a python script look like an executable, we can insert in the magic line `#!/usr/bin/env python`, which will tell the system to treat this file as a Python program:

```
dyoo@einfall:~/work/python$ cat helloworld.py
#!/usr/bin/env python
```

```
print "Hello World"
dyoo@einfall:~/work/python$ chmod +x helloworld.py
dyoo@einfall:~/work/python$ ./helloworld.py
Hello World
```

That's how we start up Python, so let's look at the things we can do with it.

3 Functions

A high-level programming language should make it easy to write modular functions, and Python is no exception. In order to create functions, we use the keyword `def`, which stands for "define". When we're using it, we tell Python what arguments the function takes in.

```
>>> def sayHello():
...     print "Hello!"
...
>>> sayHello
<function sayHello at 0x810e244>
>>> sayHello()
Hello!
>>> def double(x):
...     return x + x
...
>>> double(5)
10
>>> double('cs')
'cscs'
>>> def factorial(x):
...     if x == 0: return 1
...     else: return x * factorial(x-1)
...
>>> factorial
<function factorial at 0x810d88c>
>>> factorial(5)
120
```

One thing to notice is that Python programs have a curious lack of semicolons or braces. In Python, statements are terminated by the end of a line, and we use indentation to group blocks of statements together. If we really need to, we can use the continuation character \ to write a long command:

```
>>> def squareSum(a, b):
...     return a**2\
...          + b**2
...
```

but this is usually very rare.

By the way, if we don't specify a return value of our function, we'll automatically get the `None` value back to us.

```
>>> type(x)
<type 'None'>
```

and by using the `return` keyword, we can return a specific value back to our caller. Otherwise, functions are fairly straightforward. There are quite a few "built-in" functions in Python that make our lives easier. We'll introduce them every so often; there's a complete list of them in the Python Library Reference at `http://python.org/doc/current/lib/lib.html`

Now that we've briefly introduced functions, let's talk about Python's variables.

4 Variables

Here are some examples of variables in Python:

```
>>> num = 42
>>> name = 'beowulf'
>>> hashtable = { 0 : 'zero', 1 : 'one', 2 : 'two', 3 : 'three' }
```

The first thing that we see is that we don't need to predefine what type of thing a name binds to; a name can stand for anything:

```
>>> x = 'macbeth'
>>> x = 3.1415926
```

although, by convention, we try to use a nice naming convention to avoid confusing ourselves.

If we really want to look at the type of a value, we can use the built-in `type()` function:

```
>>> type(num)
<type 'int'>
>>> type(hashtable)
<type 'dictionary'>
```

but in many cases, we can write programs without having to worry too much about testing for types.

If we want to create a variable name, but not give it an initial value, we can assign it the `None` value:

```
>>> result = None
>>> type(result)
<type 'None'>
```

In general, `None` is very much like `null` in other programming languages.

Let's talk about some of the kinds of variables Python provides us.

4.1 Numbers

We start off with numbers, since they're one of the most fundamental types. As we'd expect, we can use many familiar operations:

```
>>> 1 + 1
2
>>> 25 * 26
650
>>> 22/7.
3.1428571428571428
>>> 1 + 2 + (3 * 4 * (5 / 6))
3
```

but we still need to be careful about the behavior of integer division. To convert an integer to a float, we can either multiply by `1.0`, or use the `float()` conversion function instead:

```
>>> float(100)
100.0
```

For symmetry, Python also gives us a `int()` function that lets us coerce anything into an integer.

Python also provides a 'long integer' numeric type that's an integer of unbounded range; we append an `l` or `L` to the end of the number. So we can do this:

```
>>> 2**128L            ## Exponentiation
340282366920938463463374607431768211456L
```

without any qualms — useful for those combinatorics problems that deal with huge quantities.

Finally, for the engineers, Python even provides a complex number type:

```
>>> mynum = 0+1j
>>> mynum**2
(-1+0j)
```

which is pretty nice, if a tad specialized. I won't even touch on some of the extended numeric types provided outside of core Python, but if you're curious, there are matrix types within *Numeric Python*, which can by found at `http://numpy.sourceforge.net`.

4.2 Strings

This next section is a bit long, but that's because manipulating strings is very useful in many applications of Python.

First, let's show some examples of strings:

```
>>> msg = 'i am a string with a \ttab in it'
>>> msg
'i am a string with a \ttab in it'
>>> print msg
i am a string with a    tab in it
>>> print 'this is a string with a \\backslash'
this is a string with a \backslash
>>> print 'this string\
... spans multiple\
... lines'
this stringspans multiplelines
>>> '''I am a
... multi line
... string'''
'I am a\nmulti line\nstring'
```

Python strings are nice because we don't need to worry about allocating them; they handle their own space, so it's very easy to manipulate and pass them around between functions. Building literal strings usually requires us to surround the text with quotes, but we can also coerce anything into a string by using the `str()` function:

```
>>> n = 2**20
>>> str(n) + str(n)
'10485761048576'
```

In Python, there's no difference between `"` and `'`; they both delimit string literals. The reason this is useful is because we can use one kind of quote within another, and not have to worry too much about abrupt quote closing:

```
>>> "Hygelac's thane scouted by the wall in Grendel's wake."
"Hygelac's thane scouted by the wall in Grendel's wake."
>>> 'He shouted, "Run away!  Run away!"'
'He shouted, "Run away!  Run away!"'
```

The last example with triple quotes above is an example of a *multi-line* string; by using triple quotes `"""`, we can make strings that span across lines pretty easily.

Once we have strings, we can manipulate them with a rich set of operations. For example, we can “multiply” strings by a number:

```
>>> header = '*' * 40
>>> print header
****************************************
```

and "add" strings together:

```
>>> 'russians declare war' + 'rington vodka is delicious'
'russians declare warrington vodka is delicious'
>>> 'a' + 'b' + 'c'
'abc'
```

Unlike Perl, Python will not automatically coerce a string into a number, so:

```
>>> multiplier = '3'
>>> print 'hello' * multiplier
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: unsupported operand type(s) for *
```

doesn't quite work: we need to tell Python to explicitly convert that string into a number first:

```
>>> print 'hello' * int(multiplier)
hellohellohello
```

which might catch a few people off-guard the first time they see it. In general, Python's philosophy is to be explicit whenever there's a source of ambiguity, to improve the readability of our programs.

Another operation that's very useful is string interpolation:

```
>>> template = 'language: %s\tcreator: %s'
>>> template % ('python', 'guido')
'language: python\tcreator: guido'
>>> template % ('perl', 'wall')
'language: perl\tcreator: wall'
```

which works very much like the `sprintf` function in C, with a few nice extensions. String interpolation is like Mad Libs, and it's very useful when we want to write string templates. Because all types can be coerced into strings, it's very common to just use `%s` as our format type.

One thing that might surprise C programmers is that Python has the notion of immutable strings — once we create a string, we can't change its internal contents:

```
>>> greeting = 'hella world'
>>> greeting[4]
'a'
>>> greeting[4] = 'o'
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object doesn't support item assignment
```

However, it's pretty easy to take "slices" out of a string:

```
>>> greeting[:4]
'hell'
>>> greeting[5:]
' world'
```

and use string concatenation:

```
>>> greeting = greeting[:4] + 'o' + greeting[5:]
>>> greeting
'hello world'
```

for most common string tasks. If we really need to look at a string, character by character, we can convert a string into a list:

```
>>> list('abcdefg')
['a', 'b', 'c', 'd', 'e', 'f', 'g']
```

do our manipulations on the list, and stitch it back into a string. But there's usually a much easier way to do this kind of string manipulation without picking out characters: Python provides a rich range of string manipulation functions in its `string` and `re` (regular expression) library modules. (I'll see if I can squeeze a regular-expression example at the end of this introduction.)

4.3 Files: Opening, reading, and writing

Before we go on with the other types of variables, it would be nice to apply some of these ideas on something useful: how do we read input from the user? Let's show how to open files, read their contents, and spit out the string to standard output:

```
>>> file = open('helloworld.py')
>>> contents = file.read()
>>> print contents
#!/usr/bin/env python
print "Hello World"
```

We can open up files by using the `open()` built-in function; what we get back is an instance of a file object, something that supports file operations like reading, and writing. Let's see what we can do with files:

```
>>> dir(file)
['close', 'closed', 'fileno', 'flush', 'isatty', 'mode', 'name',
'read', 'readinto', 'readline', 'readlines', 'seek', 'softspace',
'tell', 'truncate', 'write', 'writelines', 'xreadlines']
```

Although there are many methods, we don't need to feel too overwhelmed. For the most part, we can work with `read()`, `readline()`, and `write()` functions for the majority of our file processing jobs.

`open()` takes in an optional "mode" argument that tells it how we want to use a file: While we're `open`ing files, we can tell Python to make the file writable by giving it the "`w`" option:

```
>>> f = open('testfile.txt', 'w')
>>> f.write('this is a test.\nHello world')
>>> f.close()
>>> print open('testfile.txt').read()   ## a common idiom to concisely read a file
this is a test.
Hello world
```

Whew! Let's talk a little bit about lists, tuples, and dictionaries; afterwards, we can do a simple example that ties these ideas together.

4.4 Lists and Tuples

Lists and tuples both represent sequences of data. Let's talk about lists first. We can make Python lists by putting braces, "`[`" and "`]`", around a collection of values. For example:

```
>>> names = ['john', 'paul', 'george', 'ringo']
```

assigns the variable `names` to the list that contains those four strings.

That wasn't too bad! Lists are heterogeneous and easily nestable:

```
>>> authors_books = [ ['knuth', 'taocp'], ['tolkien', 'lotr'],
...                   [['abelson', 'sussman'], 'sicp'] ]
>>> grabbag = [22/7., 'pi', ['another list']]
>>> grabbag
[3.1428571428571428, 'pi', ['another list']]
```

and lists even support the notion of addition and multiplication:

```
>>> caesar = ['pizza']
>>> caesar * 2
['pizza', 'pizza']
```

Once we have a list, how do we get at its elements? We can access any particular element of a list with the indexing operator "`[]`":

```
>>> names[0]
'john'
>>> authors_books[2]
[['abelson', 'sussman'], 'sicp']
>>> authors_books[2][0]
['abelson', 'sussman']
```

and we can even use negative indices, which start counting from the end of the list backwards:

```
>>> names[-1]
'ringo'
>>> names[-2]
'george'
```

But not only can we get at single elements of a list, but we can also take "slices", that is, subsequences of our lists:

```
>>> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
>>> numbers[0:5]                  ## numbers[a:b] --> [a, b)
[0, 1, 2, 3, 4]
>>> numbers[7:]
[7, 8, 9, 10]
```

where we tell Python the beginning and ending points of the slice. We saw this slicing behavior with strings, and it's comforting to note that their behavior is fairly consistent with lists.

In fact, many of the operations that work on lists work equally well on strings. The built-in `len()` function, for example, can give us the length of any "sequence-like" thing:

```
>>> mylist = ['g', 'k', 'p']
>>> len(mylist)
3
>>> mystr = 'supercalifragisliticexpialidocious'
>>> len(mystr)
34
```

Furthermore, in a few sections, we'll see that the looping construct `for` can work equally well along strings as well as lists.

A nice thing about lists is that they can change and expand. For example, we can append to a list:

```
>>> mylist = []
>>> mylist.append('guns')
>>> mylist.append(['germs'])
>>> mylist.append([['steel']])
>>> mylist
['guns', ['germs'], [['steel']]]
```

and start deleting elements...or even cut whole slices out!

```
>>> toppings = ['pepperoni', 'artichoke', 'tomato', 'anchovy']
>>> del toppings[0]
>>> toppings
['artichoke', 'tomato', 'anchovy']
>>> del toppings[1:3]
>>> toppings
['artichoke']
```

One very neat thing about lists is that they know how to sort themselves with their `sort()` method:

```
>>> mystr = 'qwertyuiopasdfghjklzxcvbnm'
>>> letters = list(mystr)
>>> letters
['q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p', 'a', 's', 'd',
'f', 'g', 'h', 'j', 'k', 'l', 'z', 'x', 'c', 'v', 'b', 'n', 'm']
>>> letters.sort()
>>> letters
['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h', 'i', 'j', 'k', 'l', 'm',
'n', 'o', 'p', 'q', 'r', 's', 't', 'u', 'v', 'w', 'x', 'y', 'z']
```

Whew! When we talked about files in the previous section, we saw that we can use the `dir()` function to query what things it can do. Likewise, we can apply `dir()` on a list:

```
>>> dir(toppings)
['append', 'count', 'extend', 'index', 'insert', 'pop', 'remove', 'reverse', 'sort']
```

but let's leave the other functions for another introduction.

You might wonder: is there an easy way of building lists fast? For example, let's say that I wanted to make a list of the numbers from zero to 9. Can we do this easily? Yes! Python provides a function called `range()` which generates the numbers from 0 to n, excluding the n.

```
>>> range(10)
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

or we can take every other element from 0 to 20:

```
>>> range(0, 20, 2)
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

`range()` might seem like an arbitrary function, but it does become very useful in conjunction with the `for` loop.

Finally, I should briefly mention tuples. We can construct tuples by using parentheses:

```
>>> children = ('hermione', 'ron', 'harry')
>>> teachers = ('lupin', 'quirrell') + ('snape',)
```

Tuples have many of the features of lists, but like strings, they're immutable and can only support addition and indicing. There are a few technical reasons why Python supports both lists and tuples: Tuples are a little more efficient to construct in Python, so they're internally used to pass parameters between functions. Also, because they're guaranteed to be immutable, they can be used as keys to a hashtable. Speaking of which, let's look at dictionaries now.

4.5 Dictionaries

In CS terms, a Python dictionary is a hashtable, a container that holds things like a list. The big difference between a list and a dictionary is that the "indices" of a dictionary don't have to be numbers! We construct dictionaries by using curly braces `{}`:

```
>>> myhash = {}
>>> myhash['name'] = 'boromir'
>>> myhash['pizza'] = "zachary's"
>>> myhash.keys()
['pizza', 'name']
>>> myhash.values()
["zachary's", 'boromir']
>>> myhash.items()
[('pizza', "zachary's"), ('name', 'boromir')]
>>> myhash['name']
'boromir'
>>> dict2 = { 'holt' : 'how children fail',
...           'polya' : 'how to solve it',
...         }
>>> dict2.items()
[('holt', 'how children fail'), ('polya', 'how to solve it')]
```

Because the indices can be any immutable type, dictionaries are great when we want to store key/value pairs. We can use strings, numbers, and tuples as our keys in our dictionary, and practically anything else as our values. Pythons uses dictionaries internally for a lot of stuff, including its implementation of OOP. Dictionaries are simple to use, but are one of the most powerful tools a Python programmer has; we'll see dictionaries in many of the examples below.

That's definitely enough about variables — my apologies if that went a little long. However, it's nice to know that, on the whole, Python's variable types have a a consistent interface.

5 Control Structures

Now that we know a bit about variables, let's talk about the control structures that let us make our programs a little smarter.

5.1 Truth and Falsehood

By the way, I'd better quickly mention that Python considers the number `0`, the empty string `""`, the empty list `[]`, the empty tuple `()`, and the value `None` to be false — everything else is a true value.

5.2 if/elif

In everyday life, we find ourselves needing to make a decision. For example,

```
if the weather is cloudy:
    stay inside
otherwise, if the weather is hot:
    go for a swim
else:
    walk outside
```

could be considered a series of decisions. It turns out that we have the same need to make decisions when we're programming. Let's see how we can write if statements in pseudocodeish Python:

```
if weather.cloudy():
    self.stayInside()
elif weather.hot():
    self.swim()
else:
    self.walk()
```

Unlike C's `if` statement, we don't need to put braces, nor do we need to put parentheses around our test expression. Let's see some real `if` statements:

```
>>> num = 42
>>> if num % 2 == 0:
...     print 'num is even'
... else:
...     print 'num is odd'
...
num is even
```

5.3 Emulating "switch"

Python doesn't quite support the `switch` statement of other languages, but with some creativity, we can cook something up that has similar functionality:

```
>>> def actionHi(): print "hi"
...
>>> def actionBye(): print "bye"
...
>>> table = { 'start' : actionHi,
...           'stop' : actionBye }
>>> cmd = 'start'
>>> if table.has_key(cmd): table[cmd]()
... else: print "Invalid command"
...
hi
```

Using dictionaries that have references to functions usually works very well for these kind of situations.

5.4 for/in

The `for` loop in Python will look familiar to shell scripters: instead of iterating a variable until a condition fails, we iterate through a list of things:

```
>>> for x in ['a', 'b', 'c', 'd']:
...     print 'this is the letter', x
...
this is the letter a
this is the letter b
this is the letter c
this is the letter d
>>> for name, value in { 0 : 'zero', 1 : 'one', 2 : 'two' }.items():
...     print 'the numeral of %s is %d' % (value, name)
...
the numeral of two is 2
the numeral of one is 1
the numeral of zero is 0
```

If our background in programming is in C or Java, we shouldn't despair about Python's `for` loop — here's where `range()` comes into play: For the traditional for-loop from $i = 0$ to n, we can use range()!

```
>>> sum = 0
>>> for i in range(10):
...     sum = sum + i
...
>>> i
9
>>> sum
45
>>> for i in range(10, 20, 2):
...     print i,
...
10 12 14 16 18
```

This is nice is because there's less of a chance of writing a buggy `if` statement, since we're working on elements of a list. Furthermore, in normal cases, there's no way to write an infinite loop with `for`, because we have to do some hustling to make an infinite sequence.

If we want to modify a list in-place, it's a Python idiom to iterate over the indices of a list. For example, let's square a list of numbers:

```
>>> nums
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
>>> for i in range(len(nums)):
...     nums[i] = nums[i] * nums[i]
...
>>> nums
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

In summary, `for` loops are straightforward: we give it the variable we want iterated, as well as the sequence we want to iterate over. By sequence, I mean a tuple, list, or even a string:

```
>>> for letter in 'aeiou':
...     print letter, 'is a vowel'
...
a is a vowel
e is a vowel
i is a vowel
o is a vowel
u is a vowel
```

Sometimes, though, the `for` loop might not be powerful enough for a particular task: we can't really use Python's `for` loop to loop until a certain condition is true or not. For this kind of conditional looping, we need the `while` loop instead, and that's our next topic.

5.5 while

Let's take a look at a simple `while` loop:

```
>>> i = 0
>>> while i < 5:
...     print i, i**2, i**3
...     i = i + 1
...
0 0 0
```

```
1 1 1
2 4 8
3 9 27
4 16 64
```

Like the `if` statement, we should tell Python what condition needs to hold true as we loop through the body. Since we have more control over the looping, we can do more devious things. For example, if we want to write an infinite loop, we can write this:

```
>>> while 1:
...     # do stuff here
...     pass         # let's do a no-op
...
                     # [Time passes.  I press Ctrl-C to interrupt.]
Traceback (innermost last):
  File "<stdin>", line 1, in ?
KeyboardInterrupt
```

Both the `for` and `while` loop construct support the use of the keywords `break` and `continue`, which let us "break" out of a loop, or go back to the loop's beginning. For example, by using `break`:

```
>>> while 1:
...     cmd = raw_input()
...     if cmd == 'quit': break
...
hello
let me out of here
quit
>>>
```

we can duplicate the behavior of a `do/while` construction from the C-style languages. I won't go too much into the details here because I'm very impatient to show the next section: how do we use all these things? How do we put it all together?

6 Putting it Together 1: countwords

Now that we've gone through a lot of material, it's a good idea to try writing a program that puts these ideas together. Let's try writing a program that takes words from standard input, counts how many times each word appears, and prints those results out.

In the two programs that follow, we'll need a little bit of functionality from two modules from the standard library: `sys` and `string`. We'll see the use of the `import` and `from` statements — think of them as a way of grabbing definitions outside of core Python. There's a complete list of modules in the Library Reference, and if you're interested, I'd recommend browsing through it.

There are several ways we can approach this problem. Here's one way: let's read all the words into a list, and sort those words. If we have a sorted list, it's very easy to count how many times we've see a word, because we can look at consecutive elements. I'll need to use `sys.stdin` to get at standard input, and `string.strip()` for its whitespace-eating abilities.

```
import sys
from string import strip

words = []
while 1:
    w = strip(sys.stdin.readline())
    if not w: break
    words.append(w)
words.sort()

i = 0
while i < len(words):
    current_word, count = words[i], 0
    while i < len(words) and words[i] == current_word:
        count = count + 1
        i = i + 1
    print current_word, count
```

Another approach we can take is to use a dictionary that maps words to their distribution counts. For every word, we can lookup our dictionary and gradually increment word distributions until we read all the words. Finally, we print out all the items—both keys and values—in our dictionary.

```
import sys
from string import strip

dict = {}
while 1:
    w = strip(sys.stdin.readline())
    if not w: break
    dict[w] = dict.get(w, 0) + 1       ## get(): loopup with default value

for word, count in dict.items():
    print word, count
```

One thing to note is that the dictionary version is a lot simpler, but both versions are still pretty easy to read. Let's look at another program:

7 Putting it Together 2: counting lines, words, and characters

Here's another example that uses a few more modules from the standard library. This word-counting program takes advantage of `string.split()`, which can take in a string, and split it along whitespace boundaries. It also uses `fileinput.input()`, which makes it easy to read input from either files in our argument list, or from standard input.

```
import fileinput                        # Library module
from string import split                # split() breaks a space-delimited line into a list
lcount, wcount, ccount = 0, 0, 0        # lcount = 0, wcount = 0, ...
for line in fileinput.input():          # Similar to Perl's magic while(<>)
    lcount = lcount + 1
    wcount = wcount + len(split(line))  # len() function works well on both lists
    ccount = ccount + len(line)         # and strings.
print lcount, wcount, ccount
```

8 Advanced Topics

Believe it or not, we've just gone through most of core Python; everything else is applications of these concepts. To make this introduction a little more practical, let's explore some of these applications. The first we'll cover is regular expressions!

8.1 Regular Expressions

Regular expression are a very concise way of getting a computer to recognize patterns within strings. What do we mean by patterns? If we look at a line like:

```
The link to python.org's documentation is http://python.org/doc.
```

we can easily pick out the `http` url out of that sentence. With regular expressions, we can tell Python how to recognize and pull out these kinds of strings for us.

We can access the machinery of regular expressions through its `re` module. Let's see what `re` provides:

```
>>> import re
>>> dir(re)
['DOTALL', 'I', 'IGNORECASE', 'L', 'LOCALE', 'M', 'MULTILINE',
'S', 'U', 'UNICODE', 'VERBOSE', 'X', '__all__', '__builtins__',
'__doc__', '__file__', '__name__', 'compile', 'engine', 'error',
'escape', 'findall', 'match', 'purge', 'search', 'split', 'sub',
'subn', 'template']
```

There's quite a few functions in there! The function that we'll concentrate for this introduction is `search()`. `search()` takes in a regular expression and a target string that we're searching through, and returns something interesting if it finds the pattern we're looking for.

Let's begin by talking about how we can describe a certain pattern. A regular expression pattern itself is a string. For example, the string

```
"alpha beta"
```

is a pattern that, when given any string that contains "alpha beta" anywhere in it, will match successfully. Nothing too exciting: normal letters match with letters. But how about:

```
"a+"
```

? Here, the `+` doesn't literally stand for the plus character: instead, it attaches a meaning to the character right before it. This particular example, as a regular expression, means "One or more `a`'s". So if we're looking for repetition, using `+` works very well. In regular expression terms, `+` is a metacharacter, and there are quite a few of them. We can list out a few others:

- `.` matches any single character
- `*` matches zero or more of the last character
- `?` matches zero or one of the last character

Now that we know a little bit about specifying regular expressions, let's try it out:

```
>>> re.search('a+', 'aaaa')
<SRE_Match object at 0x812e028>
>>> re.search('a+', 'ba')
<SRE_Match object at 0x810c020>
>>> re.search('a+', 'bcde')
>>>                             ## No output means no match
>>> re.search('sp.m', 'spam')
<SRE_Match object at 0x810c020>
>>> re.search('sp.m', 'spim')
<SRE_Match object at 0x812e028>
>>> re.search('sp.m', 'spin')
```

If our regular expression engine can recognize a regular expression within our string, it returns back to us a "Match object" — something that lets us ask it for more information, and otherwise gives us `None`.

Let's take a flying leap into the unknown — let's show a regular expression that can recognize most `http` urls!

```
"http://[\w\.-/]+\.?(?![\w.-/])"
```

Admittedly, trying to figure out this regular expression takes a lot of time; however, because they're just strings, we can break them down, comment them heavily, and then use them.

```
LETTERS_OR_SYMBOLS = r'[\w\.-/]+'
OPTIONAL_LITERAL_PERIOD = r'\.?'
LOOKAHEAD_NOT_LETTER_OR_SYMBOL = r'(?![\w.-/])'
whole_re_pattern = 'http://' + LETTERS_OR_SYMBOLS + OPTIONAL_LITERAL_PERIOD \
                             + LOOKAHEAD_NOT_LETTER_OR_SYMBOL
```

There's something somewhat new here that we haven't seen yet — what's that "`r`" in front of the quotes? In Python, it's called a "raw" string. Basically, it turns off the special meaning of the backslash, so that the \ literally means backslash. We'll see raw strings a lot, especially with regular expressions that use a lot of backslashes.

Let's see if it works:

```
>>> re.search(whole_re_pattern, 'this is a test without an url')
>>> re.search(whole_re_pattern, 'this is a pattern http://with_an_url.')
<SRE_Match object at 0x810c020>
```

When we find ourselves using a certain regular expression over and over, we can cache a regular expression by `compile()`ing it:

```
>>> myre = re.compile(r"http://[\w\.-]+\.?(?![\w.-])")
```

What we get back is a regular expression object that remembers the pattern it uses to search for things. Let's see it in action.

```
>>> myre.search('all your http are belong to us')     ## no match
>>> myre.search("let's try another url: http://perl.com.")
<SRE_Match object at 0x812eb68>
```

What makes compiling a regular expression especially neat is that we can make lists of regular expressions, put them in dictionaries, and pass them around as objects, just like any other Python type.

If we want to get more information from the search, we can put parentheses around a region of our regular expression; in technical terms, we're defining a *group*. Later on, if our `search()` is successful, we can later query our Match object about what matched using its `group()` method:

```
>>> myre = re.compile(r"(http://[\w\.-]+)\.?(?![\w.-])")
>>> myre.search("The website http://python.org is neat").group(1)
'http://python.org'
```

I certainly can't do justice to regular expressions; they're very powerful and magical, and invaluable to anyone working with text manipulation. If you want to know more about them, I'd recommend taking a look at the regular expression documentation within the Library Module. Because Python's regular expressions are similar to Perl's, it's also very useful to read Perl documentation on regular expressions, especially Tom Christensen's "Far More than Everything You Wanted To Know about Regexps".

8.2 CGI

[write this later]

8.3 Tkinter

[write this later]

9 Afterwords

As with any living language, Python has its vocal speakers. Most of them chatter in `comp.lang.python`, which is one of the friendlier forums on Usenet. Also, there are several mailing lists on python.org that concentrate on topics like databases, XML parsing, and other esoteric stuff.

Personally, I participate in `tutor@python.org`, the mailing list for people who're learning Python; if you're interested, come join in! Subscribing involves going to the web site:

`http://mail.python.org/mailman/listinfo/tutor`

and we'll be happy to answer any questions about learning Python.

There are a lot of things I haven't even touched yet in this introduction, like classes, functional programming, scoping, or list comprehension. I'm already pushing past ten pages, so I'd better stop for now. (Plus I'm running out of time typing this!) I'll be happy to expand this material later, if people are interested.

Thanks for reading this!