Brief Summary of Python

Python is an interpreted language. Statements in Python are terminated by a newline; a continuation character `\` can be used to continue a statement on a second line.

Python is sensitive not only to newlines but also to how many spaces a line is indented. A block of code has all the lines indented the same number of spaces, and that is the only way that a block is indicated. It is illegal to change the indenting within a block of code, unless you are beginning a new inner block, such as the body of an if statement. There is no defined amount of spaces that the inner block must be indented, as long as it is at least one more than the outer block. A block of code ends when the indent level reverts back to the level of an outer block.

Variable and function names are case-sensitive. Variables do not need to be declared as an explicit type, in fact they are not declared at all. Python figures out which type a variable should be the first time it is used. Thus writing

```
i = 5
```

will make `i` an integer, and

```
i = "Hello"
```

will make `i` a string. However, once the type of a variable is declared, it keeps that type; if you declare:

```
i = "5"
```

you cannot then use `i` as an integer without converting it (in this case, using the built-in `int()` function). Python takes care of issues such as garbage collecting storage used by strings that are reassigned.

We won't use them here, but Python also supports floating point numbers and complex numbers (using the syntax `2.0+0.5j`, with the `j` or `J` being a required element: `2.0` is the real part and `0.5` is the imaginary part, so `1j * 1j` equals `-1+0j`).

Multiple variables can be assigned at once, as in

```
two, three = 2, 3
```

and all the expressions on the right-hand side are evaluated before any assignments are done, so two variables can be easily swapped:

```
x, y, = y, x
```

Arithmetic in Python uses the standard symbols, with `%` used for modulo:

```
total = (total * 3) - 4
counter = (counter + 1) % 100
```

Strings are surrounded by single or double quotes, which must match. Multi-line strings are surrounded by triples quotes, that is either `"""` or `'''`.

```
state = "Maryland"
longstring = """This is a multi-
   line string"""
```

Multi-line strings are often included in Python code without being assigned to anything (which means they have no semantic relevance): it's a way to include multi-line comments in the code.

Strings can be indexed into with integers (starting at zero). In addition, indices can be negative; the index –1 refers to the last character in the string, -2 the second-to-last, and so on:

```
testStr = "Ostrich"
testStr[0]   # "O"
testStr[6]   # "h"
testStr[-1]  # "h"
testStr[5]   # "c"
testStr[-2]  # "c"
testStr[-7]  # "O"
testStr[-8]  # ERROR!
testStr[7]   # ERROR!
```

Note that there is no distinction between a single character and a string; indexing into a string as shown above produces a string of length one.

The slice character, `:` (the colon), is used to obtain substrings, using the syntax `x:y` to mean "all characters from index `x` up to but not including index `y`." The default for `x` is `0`, the default for `y` is the length of the string. Finally, strings are concatenated using the `+` symbol. Thus,

```
name1 = "Tom"
name2 = "Jones"
newname = name1[0] + name2[:3] + "123"
```

Sets `newname` to `"Tjon123"`. You cannot assign directly into a substring, so

```
newname[1] = "X"  # WRONG!!
```

is illegal; instead you could write

```
newname = newname[:1] + "X" + newname[2:]
```

The function len() returns the length of a string:

```
byteCount = len(buffer)
```

Python has two types, lists and tuples, which function like arrays in many other languages. Both are sets of values. The main difference is that tuples cannot be modified once declared. The elements in a list or tuple can be of any type, mixed together.

A list is declared with square brackets

```
mylist = [ "hello", 7, "green" ]
```

and a tuple is declared with parenthesis

```
mytuple = ( "hello", 7, ( "red", 31 ) )
```

Note that mytuple has three elements, the third of which is another tuple, which itself has two elements, `"red"` and `31`. I'll mention that the middle of such a declaration list (or any place in Python that is inside an expression surrounded with parenthesis, curly braces, or square brackets) is one occasion where a line can be broken without adding the line joining character, the backslash (\).

Accessing lists and tuples is done the same way as strings, with zero-based integer indexing, negative indexing going backwards from the end, and the slice operator:

```
firstelement = mylist[0]
lastelement = mylist[-1]
subtuple = mytuple[1:3]
```

You can assign a value to an element in a list (but not an element in a tuple, since they cannot be modified):

```
mylist[1] = "newlistvalue"
```

Or assign a list to a slice; the list doesn't have to be the same length as the slice:

```
mylist [0:3] = [ "Just one value now" ]
```

`append()` adds an element to a list and `extend()` combines two lists

```
mylist.append( 9 )
mylist..extend( [ "new", "entries", 50 ] )
```

Note that `append()` and `extend()` are methods on the list object, as opposed to built-in functions; in Python all variables (and constants) are objects, but in most cases we won't care about this fact.

A new list can be created from an existing list using what are called "list comprehensions". An example is the easiest way to show this

```
oldlist = [ 1, 2, 3, 4 ]
newlist = [ elem* 2 for elem in oldlist if elem != 3 ]
```

This sets `newlist` to be `[ 2, 4, 8 ]`. The `if` part of the list comprehension is optional.

The length of a list is returned using the built-in `len()` function:

```
print "List has", len(mylist), "elements"
```

Entries are deleted using `del`, which can take a single index or a whole slice:

```
del mylist[2]
del mylist[0:1]
```

None of the functions that modify lists are allowed on tuples.

Python has a more sophisticated storage type called a dictionary. A dictionary is an unordered set of keys and values, in which values can be looked up by key. The keys are usually an integer or a string. The values can be any Python value, including another dictionary. Keys do not need to be in any order or of the same type.

For example, you can write:

```
mydict[0] = "First"
mydict[1] = "Second"
```

and from then on indexing into `mydict` will look similar to that used to access a single entry in a list or tuple:

```
newvar = mydict[1]
```

However, you don't have to use integers as keys, so you could instead write:

```
mydict["zero"] = "First"
mydict["one"] = "Second"
```

and index using the strings `"zero"` and `"one"`, or you can write

```
mydict[0] = "Zero"
mydict["Name"] = "Joe"
```

or you could combine all six statements above. If you did so, the second assignment to `mydict[0]` would replace the value `"First"` with the value `"Zero"`.

Dictionaries are defined with curly braces, `{` and `}`, using a `key:value` syntax, separated by commas:

```
mydict = { 0 : "Zero", "Name" : "Joe" }
```

You can check the number of entries in a dictionary, delete entries from a dictionary, and check if a value exists for a given key

```
length = len(mydict)
del mydict["Name"]
if 1 in mydict:
```

the `in` syntax is new in Python 2.2; previously the notation `mydict.has_key(1)` was used (`has_key()` being a method on the dictionary object).

The dictionary function `keys()` will return a list whose elements are the keys currently in the dictionary.

`if` statements end with a `:`, and the body of the `if` is then indented. The block ends when the indent ends. The keywords `else` and `elif` (else-if) can also be used:

```
if c == 12:
```

```
    c = 1
    print "starting over"
elif c < 12:
    c = c + 2
    if c in mylist
        del mylist[c]
else:
    print c
```

Note that == (two equal signs) is used for comparisons.

Loops are done by walking through the elements in a list or a string:

```
for element in mylist:
    print element
```

The `range` statement can be used to easily declare a list of increasing numbers, to simulate a typical `for` loop

```
for i in range(0, 12):
    print mystring[:i]
```

This will loop 12 times, with `i` having values from `0` to `11`. The beginning point is optional, so `range(10)` has values from `0` to `9` inclusive. A third argument to `range()` can specifying the increment to use each time:

```
for j in range (5, 105, 5):
    print j
```

will count to 100 by fives (the top of the range could have been 101 instead of 105; the range ends when it reaches or passes the end point).

A couple of things should be noted about for loops with `range`. First, after the loop is done, the loop counter will contain the value it had on the last iteration, so in the example above with `range(0, 12)`, after the loop is done `i` will be equal to `11`. Second, if the range turns out to be empty (the end point is less than or equal to the beginning point) the loop counter will not be modified at all, so if it was not initialized before the loop, it will remain uninitialized after the loop.

There are also `while` loops which iterate as long as the specific condition is true:

```
while len(mystring) > 0:
    print mystring[0]
```

```
    mystring = mystring[1:]
```

`break` and `continue` can be used to leave a loop and continue with the next iteration, respectively. Loops can also have an `else` clause, which executes if the loop reaches its end naturally (that is, without hitting a `break` statement).

```
while (k < length):
    if (something):
        break;
else:
    # if we get here, did not hit the break statement
```

Functions are defined with the `def` keyword. Only the names of the arguments are specified; the types are whatever is passed to the function at runtime:

```
def lookup_value(dict, keyvalue, defaultvalue):
    if keyvalue in dict:
        return dict[keyvalue]
    else:
        return defaultvalue
```

Functions that do not exit via a `return` statement, or that execute `return` with no arguments, actually return a built-in value called `None`.

Functions can have default parameters, as in the following definition:

```
def show_text (text, intensity=100):
```

Comments are denoted with #; everything after that on a line is ignored. Comments can also be written as strings that are not assigned to any variable.

```
""" Reinitialize the translation dictionary
"""
translations.clear()
```

Code in other modules must be imported before it is used. The statement

```
import random
```

brings in the code in the standard `random` module, which has a useful function, `random()`, that returns a floating-point number between 0 and 1 (but not equal to 1). Imported functions are called with the syntax `module-name.function-name()`, so in this case `random.random()`.

Output can be displayed with the print statement:

```
print j
```

`print` can display more complicated types, such as lists and dictionaries, with a single command.

Python supports exceptions. Code can be put inside a `try` block, followed by one or more except clauses to handle various exceptions:

```
try:
    x = int(input_buffer)
except(ValueError):
    print "Invalid input!"
```

Exceptions will propagate upwards if not handled; unhandled exceptions cause a program to terminate. Exceptions can be raised with the `raise` statement:

```
if (j > 100):
    raise ValueError, j
```

The statement `raise` with no arguments can be used within an exception handled to re-raise the exception.

The `except` statement can list multiple exceptions, and the last except clause can list no exceptions, to serve as a wildcard (this is risky since it will catch all exceptions). Exception handlers are one area where you may see the Python statement `pass`, which is defined to do nothing and exists for places where the language syntax requires a statement.

```
try:
    # some code
except (TypeError, NameError):
    pass
except:
    print "Unknown exception"
    raise
```

Python also supports user-defined exceptions, which we won't discuss or use in this book.

Python has classes, as in C++ and other languages. However, in this book we won't use classes except in their most basic form, as a way to associate named data items, the same as a `struct` (structures) in C. Such Python classes are declared with an empty declaration, with no member functions:

```
class Point:
    pass
```

A new instance of this class is created with the syntax:

```
p = Point()
```

As with local variables, class member variables don't need to be declared ahead of time, they can simply be assigned to in order to create them:

```
Point.x = 12
Point.y = 15
```

Methods can be written to allow the class instantiation operator to take parameters, for example `Point(12, 15)`, but we won't cover that, or other features of Python classes.