Python Programing

M. M. Badham Leading Edge Business Solutions (Pty) Ltd

Derived From
http://en.wikibooks.org/wiki/Python_Programming

Table of Contents

Overview

Python is high-level

Python is a high-level, structured, open-source programming language that can be used for a wide variety of programming tasks. It is good for simple quick-and-dirty scripts, as well as complex and intricate applications.

It is also a dynamically typed language that includes (but does not require one to use) object oriented features and constructs. Python supports many other programming paradigms including functional programming and aspect oriented programming.

Implementations

Python refers to programming language specification rather than an implementation. The reference implementation for Python is called CPython. CPython (written in C), behaves like an interpreter but in fact automatically compiles the Python code to byte-code before execution (the byte-code then may be saved to disk, just as automatically, so that compilation need not happen again until and unless the source gets changed).

Other popular implementations of Python include IronPython and Jython. IronPython creates CIL byte code for the dot net runtime environment, and thus can use dot net classes. Jython compiles to Java Byte code and similarly can use Java Classes.

Versions

There are two current versions of Python - 2.7 and 3.3. This is because Python 3x is not compatible with Python 2x. Most software projects are still based on Python 2x. The most obvious difference you will notice between Python 2x and Python 3x is that **print** is a statement in 2x and a function in 3x.

Easy to learn

Python supports reflection and introspection. The **`dir()`** function returns the list of the names of objects in the current scope. However, **`dir(object)`** will return the names of the attributes of the specified object. The **`locals()`** routine returns a dictionary in which the names in the local namespace are the keys and their values are the objects to which the names refer. Combined with the interactive interpreter, this provides a useful environment for exploration and prototyping.

The **`help()`** command also provides information on how to use objects.

The most unusual aspect of Python is that whitespace is significant; instead of block delimiters (braces → "{}" in the C family of languages). Indentation is used to indicate where blocks begin and end.

For example, the following Python code can be interactively typed at an interpreter prompt, to display the beginning values in the Fibonacci series:

```
>>> a,b = 0,1
>>> print(b)
1
>>> while b < 100:
...   a,b = b,(a+b)
...   print(b, end=" ")
...
1 2 3 5 8 13 21 34 55 89 144
```

Batteries included!

Python provides a powerful assortment of built-in types (e.g., lists, dictionaries and strings), a number of built-in functions, and a few constructs, mostly statements. For example, loop constructs that can iterate over items in a collection instead of being limited to a simple range of integer values. Python also comes with a powerful standard library, which includes hundreds of modules to provide routines for a wide variety of services including regular expressions and TCP/IP sessions.

Open Source

Python is used and supported by a large Python Community that exists on the Internet. The mailing lists and news groups like the tutor list actively support and help new python programmers. While they discourage doing homework for you, they are quite helpful and are populated by the authors of many of the Python textbooks currently available on the market. Python is named after Monty Python's Flying Circus comedy program and was created by Guido Van Rossum.

Getting Python

In order to program in Python you need the Python interpreter. If it is not already installed or if the version you are using is obsolete, you will need to obtain and install Python using the methods below:

Installing Python in Windows

Go to the Python Homepage or the ActiveState website and get the proper version for your platform. Download it, read the instructions and get it installed.

In order to run Python from the command line, you will need to have the python directory in your PATH. Alternatively, you could use an Integrated Development Environment (IDE) for Python like PyDEV(Eclipse), Microsoft Development Studio, DrPython, eric, PyScripter, or Python's own IDLE (which ships with every version of Python since 2.3).

The PATH variable can be modified from the Window's System control panel. The advanced tab will contain the button labeled Environment *Variables*, where you can append the newly created folder to the search path.

If you prefer having a temporary environment, you can create a new command prompt shortcut that automatically executes the following statement:

```
PATH %PATH%;c:\python26
```

Changing the "26" for the version of Python you have (26 is 2.6.x, the current version of Python 2)

Cygwin

By default, the Cygwin installer for Windows does not include Python in the downloads. However, it can be selected from the list of packages.

Installing Python on Mac

Users on Apple Mac OS X will find that it already ships with Python 2.3 (OS X 10.4 Tiger), but if you want the more recent version head to Python Download Page follow the instruction on the page and in the installers. As a bonus you will also install the Python IDE.

Installing Python on Unix environments

Python is available as a package for some Linux distributions. In some cases, the distribution CD will contain the python package for installation, while other distributions require downloading the source code and using the compilation scripts.

Gentoo GNU/Linux

Gentoo is an example of a distribution that installs Python by default - the package system *Portage* depends on Python.

Ubuntu GNU/Linux

Users of Ubuntu 6.04 (Dapper Drake) and earlier will notice that Python comes installed by default, only it sometimes is not the latest version. If you would like to update it, just open a terminal and type at the prompt:

```
$ sudo apt-get update  # This will update the software repository
$ sudo apt-get install python   # This one will actually install python
```

Source code installations

Some platforms do not have a version of Python installed, and do not have pre-compiled binaries. In these cases, you will need to download the source code from the official site. Once the download is complete, you will need to unpack the compressed archive into a folder.

To build Python, simply run the configure script (requires the Bash shell) and compile using make.

Interactive mode

Python has two basic modes: The normal "mode" is the mode where the scripted and finished `.py` files are run in the python interpreter. Interactive mode is a command line shell which gives immediate feedback for each statement, while running previously fed statements in active memory. As new lines are fed into the interpreter, the fed program is evaluated both in part and in whole.

To get into interactive mode, simply type "python" without any arguments. This is a good way to play around and try variations on syntax. Python should print something like this:

```
$ python
Python 3.0b3 (r30b3:66303, Sep  8 2008, 14:01:02) [MSC v.1500 32 bit (Intel)]
on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python 3

(If Python wouldn't run, make sure your path is set correctly. See Getting Python.)

```
$ python
Python 2.7.4 (default, Apr 19 2013, 18:28:01)
[GCC 4.7.3] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

Python 2

The >>> prompt, is Python's way of telling you that you are in interactive mode. In interactive mode what you type is immediately run. Try typing `1+1` in. Python will respond with 2. Interactive mode allows you to test out and see what Python will do. If you ever feel the need to play with new Python statements, go into interactive mode and try them out.

A sample interactive session:

```
>>> 5
5
>>> print(5*7)
35
>>> "hello" * 4
'hellohellohellohello'
>>> "hello".__class__
<type 'str'>
```

(The above code will work in both Python 2 and 3)

However, you need to be careful in the interactive environment to avoid any confusion. For example, the following is a valid Python script:

```
if 1:
  print("True")
print("Done")
```

Python 3

```
if 1:
  print "True"
print "Done"
```

Python 2

If you try to enter this as written in the interactive environment, you might be surprised by the result:

```
>>> if 1:
...   print("True")
... print("Done")
  File "<stdin>", line 3
    print("Done")
        ^
SyntaxError: invalid syntax
```

What the interpreter is saying is that the indentation of the second print was unexpected. What you should have entered was a blank line to end the first (i.e., "if") statement, before you started writing the next print statement. For example, you should have entered the statements as though they were written:

```
if 1:
  print("True")

print("Done")
```

Which would have resulted in the following:

```
>>> if 1:
...   print("True")
...
True
>>> print("Done")
Done
>>>
```

Interactive mode

Instead of Python exiting when the program is finished, you can use the -i flag to start an interactive session. This can be **very** useful for debugging and prototyping.

```
python -i hello.py
```

Creating Python programs

Welcome to Python! This tutorial will show you how to start writing programs.

Python programs are nothing more than text files, and they may be edited with a standard text editor program. What text editor you use will probably depend on your operating system; any text editor can create Python programs. It is easier to use a text editor that includes Python syntax highlighting.

Hello, World!

The first program that every programmer writes is called the "Hello, World!" program. This program simply outputs the phrase "Hello, World!" and then quits. Let's write "Hello, World!" in Python!

Open up your text editor and create a new file called `hello.py` containing just this line (you can copy-paste if you want):

For Python 3x

```
print("Hello, world!")
```

For Python 2x

```
print "Hello, world!"
```

This program uses the `print` function, which simply outputs its parameters to the terminal. `print` ends with a `newline` character, which simply moves the cursor to the next line.

Now that you've written your first program, let's run it in Python! This process differs slightly depending on your operating system.

NOTE:
In Python 2.x, print is a statement rather than a function. As such, it printed everything until the end of the line, it did not utilize parenthesis and required using a stand alone comma after the final printed item to identify that the current line was not yet complete.

Windows

- Create a folder on your computer to use for your Python programs, such as `C:\pythonpractice`, and save your `hello.py` program in that folder.
- In the Start menu, select "Run...", and type in `cmd`. This is cause the Windows terminal to open.
- Type `cd \pythonpractice` to **c**hange **d**irectory to your `pythonpractice`

folder, and hit Enter.
- Type `python hello.py` to run your program!

If it didn't work, make sure your PATH contains the python directory. See Getting Python.

Mac

- Create a folder on your computer to use for your Python programs. A good suggestion would be to name it `pythonpractice` and place it in your Home folder (the one that contains folders for Documents, Movies, Music, Pictures, etc). Save your `hello.py` program into this folder.
- Open the Applications folder, go into the Utilities folder, and open the Terminal program.
- Type `cd pythonpractice` to **c**hange **d**irectory to your `pythonpractice` folder, and hit Enter.
- Type `python hello.py` to run your program!

Linux

- Create a folder on your computer to use for your Python programs, such as `~/pythonpractice`, and save your `hello.py` program in that folder.
- Open up the terminal program. In KDE, open the main menu and select "Run Command..." to open Konsole. In GNOME, open the main menu, open the Applications folder, open the Accessories folder, and select Terminal.
- Type `cd ~/pythonpractice` to **c**hange **d**irectory to your `pythonpractice` folder, and hit Enter.
- Type `python hello.py` to run your program!

Result

The program should print:

```
Hello, world!
```

Congratulations! You're well on your way to becoming a Python programmer.

Exercises

1. Modify the `hello.py` program to say hello to a historical political leader (or to Ada Lovelace).
2. Change the program so that after the greeting, it asks, "How did you get here?".
3. Re-write the original program to use two `print` statements: one for "Hello" and one for "world". The program should still only print out on one line.

Basic syntax

There are four fundamental concepts affecting Python syntax.

1. Case Sensitivity
All variables are case-sensitive. Python treats 'number' and 'Number' as separate, unrelated entities.

2. Logical lines

Statements in Python occupy one logical line. That is a newline normally terminates a Python statement. A logical line can be extended by:

- a backslash (\) at the end of the line
- opening a bracket, any of **([{** will extend the logical line until the corresponding closing bracket.
- Opening a triple quoted string. Either ''' or """ will open as string literal and the logical line does not end until the string literal is closed.

3. Indentation

A block of code starts with a colon (:) and ends with an un-indent. Because whitespace is significant, remember that spaces and tabs don't mix, so use only one or the other when indenting your programs. A common error is to mix them. While they may look the same in editor, the interpreter will read them differently and it will result in either an error or unexpected behavior. Most decent text editors can be configured to let tab key emit spaces instead.

Python's Style Guideline describes that the preferred way is using 4 spaces.

Tips: If you invoked python from the command-line, you can give -t or -tt argument to python to make python issue a warning or error on inconsistent tab usage.

```
pythonprogrammer@wikibook:~$ python -tt myscript.py
```

This will issue an error if you mixed spaces and tabs.

4. Objects

In Python, like all object oriented languages, there are aggregations of code and data called Objects. These typically represent the pieces in a conceptual model of a system.

Objects in Python are created (i.e., instantiated) from templates called Classes (which are covered later, as much of the language can be used without understanding classes). They have "attributes", which represent the various pieces of code and data which comprise the object. To access attributes, one writes the name of the object followed by a period (henceforth called a dot), followed by the name of the attribute.

An example is the 'upper' attribute of strings, which refers to the code that returns a copy of the string in which all the letters are uppercase. To get to this, it is necessary to have a way to refer to the object (in the following example, the way is the literal string that constructs the object).

```
'bob'.upper
```

Code attributes are called "methods". So in this example, upper is a method of 'bob' (as it is of all strings). To execute the code in a method, use a matched pair of parentheses surrounding a comma separated list of whatever arguments the method accepts (upper doesn't accept any arguments). So to find an uppercase version of the string 'bob', one could use the following:

```
'bob'.upper()
```

Scope

In a large system, it is important that one piece of code does not affect another in difficult to predict ways. One of the simplest ways to further this goal is to prevent one programmer's choice of names from preventing another from choosing that name. Because of this, the concept of scope was invented. A scope is a "region" of code in which a name can be used and outside of which the name cannot be easily accessed. There are two ways of delimiting regions in Python: with functions or with modules. They each have different ways of accessing the useful data that was produced within the scope from outside the scope. With functions, that way is to return the data. The way to access names from other modules lead us to another concept.

Namespaces

It would be possible to teach Python without the concept of namespaces because they are so similar to attributes, which we have already mentioned, but the concept of namespaces is one that transcends any particular programming language, and so it is important to teach. To begin with, there is a built-in function **`dir()`** that can be used to help one understand the concept of namespaces. When you first start the Python interpreter (i.e., in interactive mode), you can list the objects in the current (or default) namespace using this function.

```
Python 2.3.4 (#53, Oct 18 2004, 20:35:07) [MSC v.1200 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> dir()
['__builtins__', '__doc__', '__name__']
```

```
Python 3.3.1 (default, Sep 25 2013, 19:29:01)
```

```
[GCC 4.7.3] on linux
Type "help", "copyright", "credits" or "license" for more information.
>>> dir()
['__builtins__', '__doc__', '__loader__', '__name__', '__package__']
```

This function can also be used to show the names available within a module namespace. To demonstrate this, first we can use the **`type()`** function to show what __builtins__ is:

```
>>> type(__builtins__)
<type 'module'>
```

Since it is a module, we can list the names within the __builtins__ namespace, again using the **`dir()`** function (note the complete list of names has been abbreviated):

```
>>> dir(__builtins__)
['ArithmeticError', ... 'copyright', 'credits', ... 'help', ... 'license', ...
'zip']
>>>
```

Namespaces are a simple concept. A namespace is a place in which a name resides. Each name within a namespace is distinct from names outside of the namespace. This layering of namespaces is called scope. A name is placed within a namespace when that name is given a value. For example:

```
>>> dir()
['__builtins__', '__doc__', '__name__']
>>> name = "Bob"
>>> import math
>>> dir()
['__builtins__', '__doc__', '__name__', 'math', 'name']
```

Note that I was able to add the "name" variable to the namespace using a simple assignment statement. The import statement was used to add the "math" name to the current namespace. To see what math is, we can simply:

```
>>> math
<module 'math' (built-in)>
```

Since it is a module, it also has a namespace. To display the names within this namespace, we:

```
>>> dir(math)
['__doc__', '__name__', 'acos', 'asin', 'atan', 'atan2', 'ceil', 'cos', 'cosh',
'degrees', 'e',
'exp', 'fabs', 'floor', 'fmod', 'frexp', 'hypot', 'ldexp', 'log', 'log10',
'modf', 'pi', 'pow',
'radians', 'sin', 'sinh', 'sqrt', 'tan', 'tanh']
>>>
```

If you look closely, you will notice that both the default namespace, and the math module namespace have a '__name__' object. The fact that each layer can contain an object with the same name is what scope is all about. To access objects inside a namespace, simply use the name of the module, followed by a dot, followed by the name of the object. This allow us to differentiate between the `__name__` object within the current namespace, and that of the object with the same name within the `math` module. For example:

```
>>> print __name__
__main__
>>> print math.__name__
math
```

```
>>> print math.__doc__
This module is always available.  It provides access to the
mathematical functions defined by the C standard.
>>> math.pi
3.1415926535897931
```

Data types

Data types determine whether an object can do something, or whether it just would not make sense. Other programming languages often determine whether an operation makes sense for an object by making sure the object can never be stored somewhere where the operation will be performed on the object (this type system is called static typing). Python does not do that. Instead it stores the type of an object with the object, and checks when the operation is performed whether that operation makes sense for that object (this is called dynamic typing).

Python's built-in datatypes are:

- **Numbers**
    - Integers, equivalent to C longs
    - Floating-Point numbers, equivalent to C doubles
    - Long integers of non-limited length (Python 2x only)
    - Complex Numbers.

- **Interables**
  Objects that support iteration
    - **Sequences**
      Objects that are subscriptable with a zero based index. Sequences also support slicing.
        - Strings
        - lists
        - tuples
    - Sets (frozen sets)
    - dictionaries, also called hashmaps, or associative arrays
    - files
    - generator objects
- functions, methods (callables)
- classes

Numbers

Python supports 4 types of Numbers, the int, the long, the float and the complex. You don’t have to specify what type of variable you want; Python does that automatically.

- *Int:* This is the basic integer type in python, it is equivalent to the hardware 'c long' for the platform you are using.
- *Long:* This is a integer number that's length is non-limited. In Python 2.2 and later, Ints are automatically turned into long ints when they overflow. In Python 3 longs have been unified
- *Float:* This is a binary floating point number. Longs and Ints are automatically converted to floats when a float is used in an expression, and with the true-division `//` operator.
- *Complex:* This is a complex number consisting of two floats. Complex literals are written as a + bj where a and b are floating-point numbers denoting the real and imaginary parts respectively.

In general, the number types are automatically 'up cast' in this order:

Int → Long → Float → Complex. The farther to the right you go, the higher the precedence.

Number literals

The type of the number is most often determined by assigning it to a literal.

Python 2

```
>>> x = 5
>>> type(x)
<type 'int'>
>>> x = 187687654564658970978909869576453
>>> type(x)
<type 'long'>
>>> x = 1.34763
>>> type(x)
<type 'float'>
>>> x=5E3
>>> type(x)
<type 'float'>
>>> x = 5 + 2j
>>> type(x)
<type 'complex'>
```

Python 3

```
>>> x=5
>>> type(x)
<class 'int'>
>>> x = 187687654564658970978909869576453
>>> type(x)
<class 'int'>
>>> x = 1.34763
>>> type(x)
<class 'float'>
>>> x=5E3
>>> type(x)
<class 'float'>
>>> x = 5 + 2j
>>> type(x)
<class 'complex'>
```

Int literals may include Octal and Hexadecimals.

```
>>> x = 010  #octal 8 decimal
>>> x = 0x10 #hex 15 decimal
```

Numerical Operators

Basics

Python math works like you would expect.

```
>>> x = 2
>>> y = 3
>>> z = 5
>>> x * y
6
>>> x + y
5
>>> x * y + z
11
>>> (x + y) * z
25
```

Note that Python's arithmetic adhere to the PEMDAS. A full list of operators an their order of precedence is shown later.

Powers (**)

There is a builtin exponentiation operator '**', which can take either integers, floating point or complex numbers. This occupies its proper place in the order of operations.

Division and Type Conversion (/ //)

For Python 2.x, dividing two integers or longs uses integer division, also known as "floor division" (applying the floor function after division. So, for example, 5 / 2 is 2. Using "/" to do division this way is deprecated; if you want floor division, use "//" (available in Python 2.2 and later).

"/" does "true division" for floats and complex numbers; for example, 5.0/2.0 is 2.5.

For Python 3.x, "/" does "true division" for all types.

Dividing by or into a floating point number (there are no fractional types in Python) will cause Python to use true division. To coerce an integer to become a float, 'float()' with the integer as a parameter

```
>>> x = 5
>>> float(x)
5.0
```

This can be generalized for other numeric types: int(), complex(), long().

Beware that due to the limitations of floating point arithmetic, rounding errors can cause unexpected results. For example:

```
>>> print 0.6/0.2
3.0
>>> print 0.6//0.2
2.0
```

Using:

```
from __future__ import division
```

Causes Python 2 to behave like Python 3 with respect to division.

```
>>> 5/2
2
>>>5/2.
2.5
>>>5./2
2.5
>>> from __future__ import division
>>> 5/2
2.5
>>> 5//2
2
```

Modulo (%)

The modulus (remainder of the division of the two operands, rather than the quotient) can be found using the % operator, or by the divmod builtin function. The divmod function returns a tuple containing the quotient and remainder.

Negation

Unlike some other languages, variables can be negated directly:

```
>>> x = 5
>>> -x
-5
```

Augmented Assignment

There is shorthand for assigning the output of an operation to one of the inputs:

```
>>> x = 2
>>> x # 2
2
>>> x *= 3
>>> x # 2 * 3
6
>>> x += 4
>>> x # 2 * 3 + 4
10
>>> x /= 5
>>> x # (2 * 3 + 4) / 5
2
>>> x **= 2
>>> x # ((2 * 3 + 4) / 5) ** 2
4
>>> x %= 3
>>> x # ((2 * 3 + 4) / 5) ** 2 % 3
1
```

Boolean

Python supports dedicated boolean literals True and False, however any non zero number can be used to represent True.

Operators

Operator	Description
x[index], x.attribute	Subscription, slicing, call, attribute reference
**	Exponentiation
+x, -x, ~x	Positive, negative, bitwise NOT
*, @, /, //, %	Multiplication, matrix multiplication, division, floor division, remainder
+, -	Addition and subtraction
<<, >>	Shifts
& ^ |	Bitwise AND XOR OR
in, not in, is, is not, <, <=, >, >=, !=, ==	Comparisons, including membership tests and identity tests
not x	Boolean NOT
and	Boolean AND
or	Boolean OR
if – else	Conditional expression
lambda	Lambda expression

Strings

String Literals

Firstly lets look at string literals, that is how we create strings.

We have single quoted strings:

```
str1 = 'hello world'
```

Double quoted strings

```
str1 = “hello world”
```

Triple quoted strings

```
str1 = '''
           hello
           world
        '''
str2 = “””
              Hello
              World
          “””
```

Triple quoted strings allow any characters between the sets of quotes including new lines. They are particularly useful if the output is HTML/XML or SQL, where newlines and quotes may be useful inside the string.

Escaping and Raw Strings

Python strings support a number of ecsape codes including:

\n	Newline
\r	Carriage return
\t	Tab
\\	A backslash

```
>>> print "hello\nworld"
hello
world
>>> print "hello\tworld"
hello          world
```

A raw string, which starts with an 'r' ingores escape sequences.

```
>>> print r"hello\tworld"
hello\tworld
```

String literals support a form of variable interpolation. That is we can place the value of a variable in the middle of a string as follows:

```
>>> change = 5
>>> print 'the change is %d cents' %  change
the change is 5 cents
```

Strings in Python are part of a group of object types calles Se

String manipulation

String operations

Equality

Two strings are equal if and only if they have *exactly* the same contents, meaning that they are both the same length and each character has a one-to-one positional correspondence. Many other languages test strings only for identity; that is, they only test whether two strings occupy the same space in memory. This latter operation is possible in Python using the operator `is`.

Example:

```
>>> a = 'hello'; b = 'hello' # Assign 'hello' to a and b.
>>> print a == b             # True
True
>>> print a == 'hello'       #
True
>>> print a == "hello"       # (choice of delimiter is unimportant)
True
>>> print a == 'hello '      # (extra space)
False
>>> print a == 'Hello'       # (wrong case)
False
```

Numerical

There are two quasi-numerical operations which can be done on strings -- addition and multiplication. String addition is just another name for concatenation. String multiplication is repetitive addition, or concatenation. So:

```
>>> c = 'a'
>>> c + 'b'
'ab'
>>> c * 5
'aaaaa'
```

Variable interpolation

String literals support a form of variable interpolation. That is we can place the value of a variable in the middle of a string this can ve done it two ways, using the **%** operator or the

format method:

The % Operator – The Old way

```
>>> change = 5
>>> print 'the change is %d cents' %  change
the change is 5 cents
```

The format method – The New Way

```
>>> change = 5
>>> print 'the change is {} cents'.format(change)
the change is 5 cents
```

By default values are formatted to take up only as many characters as needed to represent the content. It is however also possible to define that a value should be padded to a specific length.

Unfortunately the default alignment differs between old and new style formatting. The old style defaults to right aligned while for new style it's left.

Align right:

Old

```
'%10s' % ('test',)
```

New

```
'{:>10}'.format('test')
```

Output

```
      test
```

Align left:

Old

```
'%-10s' % ('test',)
```

New

```
'{:10}'.format('test')
```

Output

```
test      
```

Similar to strings numbers can also be constrained to a specific width.

Old

```
'%4d' % (42,)
```

New

```
'{:4d}'.format(42)
```

Output

```
  42
```

Again similar to truncating strings the precision for floating point numbers limits the number of positions after the decimal point.

For floating points the padding value represents the length of the complete output. In the example below we want our output to have at least 6 characters with 2 after the decimal point.

Old

```
'%06.2f' % (3.141592653589793,)
```

New

```
'{:06.2f}'.format(3.141592653589793)
```

Output

```
003.14
```

For integer values providing a precision doesn't make much sense and is actually forbidden in the new style (it will result in a ValueError).

Old

```
'%04d' % (42,)
```

New

```
'{:04d}'.format(42)
```

Output

```
0042
```

Sequence Operators

Strings, along with lists and tuples are part of a group of types called *sequences*. All sequences support the in orator, indexing (ordinal subscripts) and slicing.

Containment

There is a simple operator 'in' that returns True if the first operand is contained in the second. This also works on substrings

```
>>> x = 'hello'
>>> y = 'll'
>>> x in y
False
>>> y in x
True
```

Note that 'print x in y' would have also returned the same value.

Indexing and Slicing

Much like arrays in other languages, the individual characters in a string can be accessed by an integer representing its position in the string. The first character in string s would be s[0] and the nth character would be at s[n-1].

```
>>> s = "Xanadu"
>>> s[1]
'a'
```

Unlike arrays in other languages, Python also indexes the arrays backwards, using negative numbers. The last character has index -1, the second to last character has index -2, and so on.

```
>>> s[-4]
'n'
```

We can also use "slices" to access a substring of s. s[a:b] will give us a string starting with s[a] and ending with s[b-1].

```
>>> s[1:4]
'ana'
```

Neither of these is assignable.

```
>>> print s
>>> s[0] = 'J'
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> s[1:3] = "up"
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: object does not support slice assignment
>>> print s
```

Outputs (assuming the errors were suppressed):

```
Xanadu
Xanadu
```

Another feature of slices is that if the beginning or end is left empty, it will default to the first or last index, depending on context:

```
>>> s[2:]
'nadu'
>>> s[:3]
'Xan'
>>> s[:]
'Xanadu'
```

You can also use negative numbers in slices:

```
>>> print s[-2:]
'du'
```

To understand slices, it's easiest not to count the elements themselves. It is a bit like counting not on your fingers, but in the spaces between them. The list is indexed like this:

```
Element:     1     2     3     4
Index:       0     1     2     3
            -4    -3    -2    -1
```

So, when we ask for the [1:3] slice, that means we start at index 1, and end at index 3, and take everything in between them. If you are used to indexes in C or Java, this can be a bit disconcerting until you get used to it.

String methods

There are a number of methods of built-in string functions:

`capitalize`( )

Return a copy of the string with only its first character capitalized.

`center`(*width*[*, fillchar*])

Return centered in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). Changed in version 2.4: Support for the *fillchar* argument.

`count`(*sub*[*, start*[*, end*]])

Return the number of occurrences of substring *sub* in string `S[`*start*`:`*end*`]`. Optional arguments *start* and *end* are interpreted as in slice notation.

Example:

```
>>> s = 'Hello, world'
>>> s.count('l') # print the number of 'l's in 'Hello, World' (3)
3
```

`decode`([*encoding*[*, errors*]])

Decodes the string using the codec registered for *encoding*. *encoding* defaults to the default string encoding. *errors* may be given to set a different error handling scheme. The default is `'strict'`, meaning that encoding errors raise `UnicodeError`. Other possible values are `'ignore'`, `'replace'` and any other name registered via `codecs.register_error`, see section 4.8.1. New in version 2.2. Changed in version 2.3: Support for other error handling schemes added.

`encode`([*encoding*[*,errors*]])

Return an encoded version of the string. Default encoding is the current default string encoding. *errors* may be given to set a different error handling scheme. The default for *errors* is `'strict'`, meaning that encoding errors raise a `UnicodeError`. Other possible values are `'ignore'`, `'replace'`, `'xmlcharrefreplace'`, `'backslashreplace'` and any other name registered via `codecs.register_error`, see section 4.8.1. For a list of possible encodings, see section 4.8.3. New in version 2.0. Changed in version 2.3: Support for `'xmlcharrefreplace'` and `'backslashreplace'` and other error handling schemes added.

`endswith`(*suffix*[*, start*[*, end*]])

Return `True` if the string ends with the specified *suffix*, otherwise return `False`. *suffix* can also be a tuple of suffixes to look for. With optional *start*, test beginning at that position. With optional *end*, stop comparing at that position.

`expandtabs`([*tabsize*])

Return a copy of the string where all tab characters are expanded using spaces. If *tabsize* is not given, a tab size of 8 characters is assumed.

`find`(*sub*[*, start*[*, end*]])

Return the lowest index in the string where substring *sub* is found, such that *sub* is contained in the range [*start*, *end*]. Optional arguments *start* and *end* are interpreted as in slice notation. Return `-1` if *sub* is not found.

index(*sub*[*, start*[*, end*]])

Like `find()`, but raise `ValueError` when the substring is not found.

isalnum( )

Return true if all characters in the string are alphanumeric and there is at least one character, false otherwise.

isalpha( )

Return true if all characters in the string are alphabetic and there is at least one character, false otherwise.

isdigit( )

Return true if all characters in the string are digits and there is at least one character, false otherwise.

islower( )

Return true if all cased characters in the string are lowercase and there is at least one cased character, false otherwise.

isspace( )

Return true if there are only whitespace characters in the string and there is at least one character, false otherwise.

istitle( )

Return true if the string is a titlecased string and there is at least one character, for example uppercase characters may only follow uncased characters and lowercase characters only cased ones. Return false otherwise.

Example:

```
>>> '2YK'.istitle()
False
>>> '2Yk'.istitle()
True
>>> '2Y K'.istitle()
True
```

isupper( )

Return true if all cased characters in the string are uppercase and there is at least one cased character, false otherwise.

join(*seq*)

Return a string which is the concatenation of the strings in the sequence *seq*. The separator between elements is the string providing this method.

ljust(*width*[*, fillchar*])

Return the string left justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if *width* is less than `len(s)`. Changed in version 2.4: Support for the *fillchar* argument.

`lower`( )

Return a copy of the string converted to lowercase.

For 8-bit strings, this method is locale-dependent.

`lstrip`([*chars*])

Return a copy of the string with leading characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.lstrip()
'spacious   '
>>> 'www.example.com'.lstrip('cmowz.')
'example.com'
```

`partition`(*sep*)

Split the string at the first occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing the string itself, followed by two empty strings. .

`replace`(*old, new*[*, count*])

Return a copy of the string with all occurrences of substring *old* replaced by *new*. If the optional argument *count* is given, only the first *count* occurrences are replaced.

`rfind`(*sub* [*,start* [*,end*]])

Return the highest index in the string 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.

`rindex`(*sub*[*, start*[*, end*]])

Like `rfind()` but raises `ValueError` when the substring *sub* is not found.

`rjust`(*width*[*, fillchar*])

Return the string right justified in a string of length *width*. Padding is done using the specified *fillchar* (default is a space). The original string is returned if *width* is less than `len(s)`. Changed in version 2.4: Support for the *fillchar* argument.

`rpartition`(*sep*)

Split the string at the last occurrence of *sep*, and return a 3-tuple containing the part before the separator, the separator itself, and the part after the separator. If the separator is not found, return a 3-tuple containing two empty strings, followed by the string itself. New in version 2.5.

`rsplit`([*sep* [*,maxsplit*]])

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done, the *rightmost* ones. If *sep* is not specified or `None`, any whitespace string is a separator. Except for splitting from the right, `rsplit()` behaves like `split()` which is described in detail below. New in version 2.4.

`rstrip`([*chars*])

Return a copy of the string with trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a suffix; rather, all combinations of its values are stripped:

```
>>> '   spacious   '.rstrip()
'   spacious'
>>> 'mississippi'.rstrip('ipz')
'mississ'
```

`split`([*sep* [,*maxsplit*]])

Return a list of the words in the string, using *sep* as the delimiter string. If *maxsplit* is given, at most *maxsplit* splits are done. (thus, the list will have at most *maxsplit*`+1` elements). If *maxsplit* is not specified, then there is no limit on the number of splits (all possible splits are made). Consecutive delimiters are not grouped together and are deemed to delimit empty strings (for example, "`'1„2'.split(',')`"returns "`['1', '', '2']`"). The *sep* argument may consist of multiple characters (for example, "`'1, 2, 3'.split(', ')`" returns "`['1', '2', '3']`"). Splitting an empty string with a specified separator returns "`['']`".

If *sep* is not specified or is `None`, a different splitting algorithm is applied. First, whitespace characters (spaces, tabs, newlines, returns, and formfeeds) are stripped from both ends. Then, words are separated by arbitrary length strings of whitespace characters. Consecutive whitespace delimiters are treated as a single delimiter ("`'1 2 3'.split()`" returns "`['1', '2', '3']`"). Splitting an empty string or a string consisting of just whitespace returns an empty list.

`splitlines`([*keepends*])

Return a list of the lines in the string, breaking at line boundaries. Line breaks are not included in the resulting list unless *keepends* is given and true.

`startswith`(*prefix*[, *start*[, *end*]])

Return `True` if string starts with the *prefix*, otherwise return `False`. *prefix* can also be a tuple of prefixes to look for. With optional *start*, test string beginning at that position. With optional *end*, stop comparing string at that position.

`strip`([*chars*])

Return a copy of the string with the leading and trailing characters removed. The *chars* argument is a string specifying the set of characters to be removed. If omitted or `None`, the *chars* argument defaults to removing whitespace. The *chars* argument is not a prefix or suffix; rather, all combinations of its values are stripped:

```
    >>> '   spacious   '.strip()
    'spacious'
    >>> 'www.example.com'.strip('cmowz.')
    'example'
```

`swapcase`( )

Return a copy of the string with uppercase characters converted to lowercase and vice versa.

`title`( )

Return a titlecased version of the string: words start with uppercase characters, all remaining cased characters are lowercase.

Example:

```
>>> s = 'Hello, wOrLD'
>>> s
'Hello, wOrLD'
>>> s.title()
'Hello, World'
>>> s.swapcase()
'hELLO, WoRld'
>>> s.upper()
'HELLO, WORLD'
>>> s.lower()
'hello, world'
>>> s.capitalize()
'Hello, world'
```

`translate`(*table*[, *deletechars*])

Return a copy of the string where all characters occurring in the optional argument *deletechars* are removed, and the remaining characters have been mapped through the given translation table, which must be a string of length 256.

You can use the `maketrans()` helper function in the `string` module to create a translation table.

`upper`( )

Return a copy of the string converted to uppercase.

`zfill`(*width*)

Return the numeric string left filled with zeros in a string of length *width*. The original string is returned if *width* is less than `len(`*`s`*`)`.

`Format`(*args, **kwargs)

Return a formatted version of the string, using substitutions from args and kwargs. The substitutions are identified by braces ('{' and '}').

Lists

About lists in Python

A list in Python is an ordered group of items (or *elements*). It is a very general structure, and list elements don't have to be of the same type. For instance, you could put numbers, letters, strings and donkeys all on the same list.

Like strings lists are sequences, however unlike strings lists are mutable that means you can actually change a list after creating it unlike a strings.

```
>>> s = "hello"
>>> z = s
>>> s += " there"
>>> print(s)
hello there
>>> print(z)
hello
>>> l = [1,2,3,4]
>>> m = l
>>> l += [5,6,7]
>>> print l
[1, 2, 3, 4, 5, 6, 7]
>>> print m
[1, 2, 3, 4, 5, 6, 7]
```

List notation

There are two different ways to make a list in python. The first is through assignment ("statically"), the second is using list comprehensions("actively").

To make a static list of items, write them between square brackets. For example:

```
[ 1,2,3,"This is a list",'c',Donkey("kong") ]
```

A couple of things to look at.

1. There are different data types here. Lists in python may contain more than one data type.
2. Objects can be created 'on the fly' and added to lists. The last item is a new kind of Donkey.

Writing lists this way is very quick (and obvious). However, it does not take into account the current state of anything else. The other way to make a list is to form it using list comprehension. That means you actually describe the process. To do that, the list is broken into two pieces. The first is a picture of what each element will look like, and the second is what you do to get it.

For instance, lets say we have a list of words:

```
listOfWords = ["this","is","a","list","of","words"]
```

We will take the first letter of each word and make a list out of it.

```
>>> listOfWords = ["this","is","a","list","of","words"]
>>> items = [ word[0] for word in listOfWords ]
>>> print(items)
['t', 'i', 'a', 'l', 'o', 'w']
```

List comprehension allows you to use more than one for statement. It will evaluate the items in all of the objects sequentially and will loop over the shorter objects if one object is longer than the rest.

```
>>> item = [x+y for x in 'flower' for y in 'pot']
>>> print item
['fp', 'fo', 'ft', 'lp', 'lo', 'lt', 'op', 'oo', 'ot', 'wp', 'wo', 'wt', 'ep',
'eo', 'et', 'rp', 'ro', 'rt']
```

Python's list comprehension does not define a scope. Any variables that are bound in an evaluation remain bound to whatever they were last bound to when the evaluation was completed:

```
>>> print x, y
r t
```

This is exactly the same as if the comprehension had been expanded into an explicitly-nested group of one or more 'for' statements and 0 or more 'if' statements.

List creation shortcuts

Python provides a shortcut to initialize a list to a particular size and with an initial value for each element:

```
>>> zeros=[0]*5
>>> print zeros
[0, 0, 0, 0, 0]
```

This works for any data type:

```
>>> foos=['foo']*8
>>> print foos
['foo', 'foo', 'foo', 'foo', 'foo', 'foo', 'foo', 'foo']
```

with a caveat. When building a new list by multiplying, Python copies each item by reference. This poses a problem for mutable items, for instance in a multidimensional array where each element is itself a list. You'd guess that the easy way to generate a two dimensional array would be:

```
listoflists=[ [0]*4 ] *5
```

and this works, but probably doesn't do what you expect:

```
>>> listoflists=[ [0]*4 ] *5
>>> print listoflists
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> listoflists[0][2]=1
>>> print listoflists
[[0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0]]
```

What's happening here is that Python is using the same reference to the inner list as the elements of the outer list. Another way of looking at this issue is to examine how Python sees the above definition:

```
>>> innerlist=[0]*4
>>> listoflists=[innerlist]*5
>>> print listoflists
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> innerlist[2]=1
>>> print listoflists
[[0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0], [0, 0, 1, 0]]
```

Assuming the above effect is not what you intend, one way around this issue is to use list comprehensions:

```
>>> listoflists=[[0]*4 for i in range(5)]
>>> print listoflists
[[0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
>>> listoflists[0][2]=1
>>> print listoflists
[[0, 0, 1, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0], [0, 0, 0, 0]]
```

range()

The **range()** function can be used to create lists in Python2, and indirectly on Python3. The **range** function in Python 3 is like the **xrange()** function of python 2, that is rather than returning a real list it returns an iterable object that in-turn can be iterated over to create a list.

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

Python 2

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

Python 3

range can accept up to three arguments, **range(**start, stop, step**)**.

```
>>> list(range(5,10))
[5, 6, 7, 8, 9]
>>> list(range(2,100,5))
[2, 7, 12, 17, 22, 27, 32, 37, 42, 47, 52, 57, 62, 67, 72, 77, 82, 87, 92, 97]
>>> list(range(10,1,-1))
[10, 9, 8, 7, 6, 5, 4, 3, 2]
```

Just as in with slices the start is inclusive and the stop is exclusive.

Operations on lists

List Attributes

To find the length of a list use the built in len() method.

```
>>> len([1,2,3])
3
>>> a = [1,2,3,4]
>>> len( a )
4
```

Combining lists

Lists can be combined in several ways. The easiest is just to 'add' them. For instance:

```
>>> [1,2] + [3,4]
[1, 2, 3, 4]
```

The other way to append a value to a list is to use **append**. For example:

```

>>> p=[1,2]
>>> p.append([3,4])
>>> p
[1, 2, [3, 4]]
>>> # or
>>> print p
[1, 2, [3, 4]]
```

Getting pieces of lists (slices)

Like strings, lists can be indexed and sliced.

```
>>> list = [2, 4, "usurp", 9.0,"n"]
>>> list[2]
'usurp'
>>> list[3:]
[9.0, 'n']
```

Much like the slice of a string is a substring, the slice of a list is a list. However, lists differ from strings in that we can assign new values to the items in a list.

```
>>> list[1] = 17
>>> list
```

```
[2, 17, 'usurp', 9.0,'n']
```

We can even assign new values to slices of the lists, which don't even have to be the same length

```
>>> list[1:4] = ["opportunistic", "elk"]
>>> list
[2, 'opportunistic', 'elk', 'n']
```

It's even possible to append things onto the end of lists by assigning to an empty slice:

```
>>> list[:0] = [3.14,2.71]
>>> list
[3.14, 2.71, 2, 'opportunistic', 'elk', 'n']
```

You can also completely change contents of a list:

```
>>> list[:] = ['new', 'list', 'contents']
>>> list
['new', 'list', 'contents']
```

On the right site of assign statement can be any iterable type:

```
>>> list[:2] = ('element',('t',),[])
>>> list
['element', ('t',), [], 'contents']
```

With slicing you can create copy of list because slice returns a new list:

```
>>> original = [1, 'element', []]
>>> list_copy = original[:]
>>> list_copy
[1, 'element', []]
>>> list_copy.append('new element')
>>> list_copy
[1, 'element', [], 'new element']
>>> original
[1, 'element', []]
```

but this is shallow copy and contains references to elements from original list, so be careful with mutable types:

```
>>> list_copy[2].append('something')
>>> original
[1, 'element', ['something']]
```

Comparing lists

Lists can be compared for equality.

```
>>> [1,2] == [1,2]
True
```

```
>>> [1,2] == [3,4]
False
```

List methods

count(value)

Return number of occurrences of value in the list

extend(iterable)

Extend list by appending elements from the iterable object

```
>>> a = [1,2,3]
>>> b = [4,5,6]
>>> a.extend(b)
>>> print a
[1, 2, 3, 4, 5, 6]
```

index(value, [start, [stop]])

Return first index of value. Raises ValueError if the value is not present.

insert(index, object)

Insert object before index

pop([index])

Remove and return item at index (default last). Raises IndexError if list is empty or index is out of range.

remove(value)

remove first occurrence of value.

reverse()

Reverse does not return a value but reverses the order of the elements in the list.

sort()

Sort does not return a value but sorts the list that it is called from.

```
>>> list = [2, 3, 1, 'a', 'b']
>>> list.sort()
>>> list
[1, 2, 3, 'a', 'b']
```

Note that the list is sorted in place, and the sort() method returns **None** to emphasize this side effect.

If you use Python 2.4 or higher there are some more sort parameters:

sort(cmp,key,reverse)

cmp : method to be used for sorting

key : function to be executed with key element. List is sorted by return-value of the function

reverse : sort ascending y/n

Like other sequences we can test membership using the **in** operator

```
>>> "three" in my_list
True
```

List method examples:

```
>>> my_list.append(2)
>>> my_list
['one', 'two', 'three', 4, 2]
>>> my_list.count(2)
1
>>> my_list.append(2)
>>> my_list.count(2)
2
>>> my_list.extend([3,4])
>>> my_list
['one', 'two', 'three', 4, 2, 2, 3, 4]
>>> my_list.index(2)
4
>>> my_list.insert(1,"new")
>>> my_list
['one', 'new', 'two', 'three', 4, 2, 2, 3, 4]
>>> print my_list.pop()
4
>>> my_list
['one', 'new', 'two', 'three', 4, 2, 2, 3]
>>> my_list.remove("one")
>>> my_list
['new', 'two', 'three', 4, 2, 2, 3]
>>> my_list.sort()
>>> my_list
[2, 2, 3, 4, 'new', 'three', 'two']
>>> my_list.reverse()
>>> my_list
['two', 'three', 'new', 4, 3, 2, 2]
```

Tuples

Tuples like lists are ordered collections of objects of any types. The difference between lists and tuples is fundamentally about mutability. Lists are mutable and tuples are immutable.

Tuple Notation

Tuples are created in a similar manner to lits. Instead of surrounding a tuple literal with square brackets [], use parentheses ().

```
t1 = (1,2,3,4)
```

Other sequences can be cast as tuples using the **tuple()** function.

```
t2 = tuple([1,2,3,4])
```

When defining a tuple with of one element: Use a trailing comma to avoid the confusion between the use of parentheses as an order of precedence control and a tuple. .

```
i1 = (1)   # int
t3 = (1,)  # tuple
```

Tuple Operations

Tuples support all the operations that apply to sequences including:

- subscripts with a zero based index
- slicing
- concatenation with +
- repetition with *

Tuples can be assigned to lists:

```
(a,b) = 'one two'.split(' ') # a='one' and b = 'two'
```

If the context does not create ambiguity the parentheses of the tuple can be omitted, as below:

```
a,b = 'one two'.split(' ') # a='one' and b = 'two'
```

Tuple Methods

Tuples are immutable, and therefore do not support methods that would change the object. Methods provided by tuples include: **count** and **index**, which beghave identically to the equivalent list methods.

Dictionaries

About dictionaries in Python

A dictionary in python is a collection of unordered values which are accessed by key.

Dictionary notation

Dictionaries may be created directly or converted from sequences. Dictionaries are enclosed in curly braces, {}

```
>>> d = {'city':'Paris', 'age':38, (102,1650,1601):'A matrix coordinate'}
>>> seq = [('city','Paris'), ('age', 38), ((102,1650,1601),'A matrix
coordinate')]
>>> d
{'city': 'Paris', 'age': 38, (102, 1650, 1601): 'A matrix coordinate'}
>>> dict(seq)
{'city': 'Paris', 'age': 38, (102, 1650, 1601): 'A matrix coordinate'}
>>> d == dict(seq)
True
```

Also, dictionaries can be easily created by zipping two sequences.

```
>>> seq1 = ('a','b','c','d')
>>> seq2 = [1,2,3,4]
>>> d = dict(zip(seq1,seq2))
>>> d
{'a': 1, 'c': 3, 'b': 2, 'd': 4}
```

Operations on Dictionaries

The operations on dictionaries are somewhat unique. Slicing is not supported, since the items have no intrinsic order.

```
>>> d = {'a':1,'b':2, 'cat':'Fluffers'}
>>> d.keys()
['a', 'b', 'cat']
>>> d.values()
[1, 2, 'Fluffers']
>>> d['a']
1
>>> d['cat'] = 'Mr. Whiskers'
>>> d['cat']
'Mr. Whiskers'
>>> d.has_key('cat')
True
>>> d.has_key('dog')
False
>>> 'cat' in d
True
```

Combining two Dictionaries

You can combine two dictionaries by using the update method of the primary dictionary.

Note that the update method will merge existing elements if they conflict.

```
>>> d = {'apples': 1, 'oranges': 3, 'pears': 2}
>>> ud = {'pears': 4, 'grapes': 5, 'lemons': 6}
>>> d.update(ud)
>>> d
{'grapes': 5, 'pears': 4, 'lemons': 6, 'apples': 1, 'oranges': 3}
>>>
```

Deleting from dictionary

```
del dictionaryName[membername]
```

Dictionary Methods

Dictionaries also support a number of methods.

clear( )

Remove all items from dictionary.

copy( )

A shallow copy of the dictionary, that is the dictionary itself is copied but its members still reference the same objects.

get(key[,default])

Returns the value referenced by key, unless key does not exist in which case: either the default value is returned if supplied or None. Almost the same as:

d[key] **if** key in d **else** default

```
>>> d['name']
'fred'
>>> d.get('name')
'fred'
>>> d['foo']
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
KeyError: 'foo'
>>> d.get('foo')
>>> d.get('foo','default')
'default'
```

has_key(key)

Returns True if the dictionary has key in argument, else False. The **in** operator will have a similar result to the has_key method.

```
>>> d
{'age': '10', 'surname': 'basset', 'name': 'fred'}
>>> 'name' in d
True
>>> d.has_key('name')
True
```

items( )

A list of the dictionary's key, value pairs, as tuples. [(k1,v1),(k2,v2),...]

In python 3 **items** returns an iterable object.

keys( )

A list of the dictionary's keys. In python 3 **keys** returns an iterable object.

pop(key)

Remove specified key and return the corresponding value.

popitem( )

remove and return a key, value pair as a tuple.

values( )

A list of the dictionary's values.

```
>>> d.values()
['10', 'basset', 'fred']
```

In Python 3 **values** retuns an iterable objetc, to get the same result as above use the **list** function.

```
>>> list(d.values())
['10', 'basset', 'fred']
```

The in operator cannot directly test the membership of values:

```
>>> 'fred' in d
False
>>> 'fred' in d.values()
True
```

Sets

Python also has an implementation of the mathematical set. Unlike sequence objects such as lists and tuples, in which each element is indexed, a set is an unordered collection of objects. Sets also cannot have duplicate members - a given object appears in a set 0 or 1 times. For more information on sets, see the Set Theory wikibook. Sets also require that all members of the set be hashable. Any object that can be used as a dictionary key can be a set member. Integers, floating point numbers, tuples, and strings are hashable; dictionaries, lists, and other sets (except frozensets) are not.

Constructing Sets

One way to construct sets is by passing any sequential object to the "set" constructor.

```
>>> set([0, 1, 2, 3])
set([0, 1, 2, 3])
>>> set("obtuse")
set(['b', 'e', 'o', 's', 'u', 't'])
```

We can also add elements to sets one by one, using the "add" function.

```
>>> s = set([12, 26, 54])
>>> s.add(32)
>>> s
set([32, 26, 12, 54])
```

Note that since a set does not contain duplicate elements, if we add one of the members of s to s again, the add function will have no effect. This same behavior occurs in the "update" function, which adds a group of elements to a set.

```
>>> s.update([26, 12, 9, 14])
>>> s
set([32, 9, 12, 14, 54, 26])
```

Note that you can give any type of sequential structure, or even another set, to the update function, regardless of what structure was used to initialize the set.

The set function also provides a copy constructor. However, remember that the copy constructor will copy the set, but not the individual elements.

```
>>> s2 = s.copy()
>>> s2
set([32, 9, 12, 14, 54, 26])
```

Membership Testing

We can check if an object is in the set using the same "in" operator as with sequential data types.

```
>>> 32 in s
True
>>> 6 in s
False
```

```
>>> 6 not in s
True
```

We can also test the membership of entire sets. Given two sets S_1 and S_2, we check if S_1 is a subset or a superset of S_2.

```
>>> s.issubset(set([32, 8, 9, 12, 14, -4, 54, 26, 19]))
True
>>> s.issuperset(set([9, 12]))
True
```

Note that "issubset" and "issuperset" can also accept sequential data types as arguments

```
>>> s.issuperset([32, 9])
True
```

Note that the <= and >= operators also express the issubset and issuperset functions respectively.

```
>>> set([4, 5, 7]) <= set([4, 5, 7, 9])
True
>>> set([9, 12, 15]) >= set([9, 12])
True
```

Like lists, tuples, and string, we can use the "len" function to find the number of items in a set.

Removing Items

There are three functions which remove individual items from a set, called pop, remove, and discard. The first, pop, simply removes an item from the set. Note that there is no defined behavior as to which element it chooses to remove.

```
>>> s = set([1,2,3,4,5,6])
>>> s.pop()
1
>>> s
set([2,3,4,5,6])
```

We also have the "remove" function to remove a specified element.

```
>>> s.remove(3)
>>> s
set([2,4,5,6])
```

However, removing a item which isn't in the set causes an error.

```
>>> s.remove(9)
```

```
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
KeyError: 9
```

If you wish to avoid this error, use "discard." It has the same functionality as remove, but will simply do nothing if the element isn't in the set

We also have another operation for removing elements from a set, clear, which simply removes all elements from the set.

```
>>> s.clear()
>>> s
set([])
```

Iteration Over Sets

We can also have a loop move over each of the items in a set. However, since sets are unordered, it is undefined which order the iteration will follow.

```
>>> s = set("blerg")
>>> for n in s:
...     print n,
...
r b e l g
```

Set Operations

Python allows us to perform all the standard mathematical set operations, using members of set. Note that each of these set operations has several forms. One of these forms, s1.function(s2) will return another set which is created by "function" applied to S_1 and S_2. The other form, s1.function_update(s2), will change S_1 to be the set created by "function" of S_1 and S_2. Finally, some functions have equivalent special operators. For example, s1 & s2 is equivalent to s1.intersection(s2)

Union

The union is the merger of two sets. Any element in S_1 or S_2 will appear in their union.

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.union(s2)
set([1, 4, 6, 8, 9])
>>> s1 | s2
set([1, 4, 6, 8, 9])
```

Note that union's update function is simply "update" above.

Intersection

Any element which is in both S_1 and S_2 will appear in their intersection.

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.intersection(s2)
set([6])
```

```
>>> s1 & s2
set([6])
>>> s1.intersection_update(s2)
>>> s1
set([6])
```

Symmetric Difference

The symmetric difference of two sets is the set of elements which are in one of either set, but not in both.

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.symmetric_difference(s2)
set([8, 1, 4, 9])
>>> s1 ^ s2
set([8, 1, 4, 9])
>>> s1.symmetric_difference_update(s2)
>>> s1
set([8, 1, 4, 9])
```

Set Difference

Python can also find the set difference of S_1 and S_2, which is the elements that are in S_1 but not in S_2.

```
>>> s1 = set([4, 6, 9])
>>> s2 = set([1, 6, 8])
>>> s1.difference(s2)
set([9, 4])
>>> s1 - s2
set([9, 4])
>>> s1.difference_update(s2)
>>> s1
set([9, 4])
```

frozenset

A frozenset is basically the same as a set, except that it is immutable - once it is created, its members cannot be changed. Since they are immutable, they are also hashable, which means that frozensets can be used as members in other sets and as dictionary keys. frozensets have the same functions as normal sets, except none of the functions that change the contents (update, remove, pop, etc.) are available.

```
>>> fs = frozenset([2, 3, 4])
>>> s1 = set([fs, 4, 5, 6])
>>> s1
set([4, frozenset([2, 3, 4]), 6, 5])
>>> fs.intersection(s1)
frozenset([4])
>>> fs.add(6)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'frozenset' object has no attribute 'add'
```

Flow control

As with most imperative languages, there are three main categories of program flow control:

- loops
- branches
- function calls

Function calls are covered in a later section.

Generators and list comprehensions are advanced forms of program flow control, but they are not covered here.

Branches

If ... elif ... else

There is basically only one kind of branch in Python, the 'if' statement. The simplest form of the if statement simple executes a block of code only if a given predicate is true, and skips over it if the predicate is false

For instance,

```
>>> x = 10
>>> if x > 0:
...    print("Positive")
...
Positive
>>> if x < 0:
...    print("Negative")
...
```

You can also add "elif" (short for "else if") branches onto the if statement. If the predicate on the first “if” is false, it will test the predicate on the first elif, and run that branch if it’s true. If the first elif is false, it tries the second one, and so on. Note, however, that it will stop checking branches as soon as it finds a true predicate, and skip the rest of the if statement. You can also end your if statements with an "else" branch. If none of the other branches are executed, then python will run this branch.

```
>>> x = -6
>>> if x > 0:
...    print "Positive"
... elif x == 0:
...    print "Zero"
... else:
...    print "Negative"
...
'Negative'
```

Boolean expressions

In Python, any non zero number, and non empty string or any object that is not None; is evaluated as True.

The following operators are used in dedicated boolean expressions:

< <= > >= == != in

The key words **and**, **or** and **not** can be used to create compound with boolean expressions.

Loops

In Python, there are two kinds of loops, 'for' loops and 'while' loops.

While loops

A while loop repeats a sequence of statements until some condition becomes false. For example:

```
x = 5
while x > 0:
    print(x)
    x -= 1
```

will output

```
5
4
3
2
1
```

For loops

A for loop iterates over elements of an iterable object. Which includes sequences (string, tuple or list). A variable is created in the loop to represent each object in the sequence.

For example,

```
l = [100,200,300,400]
for i in l:
    print(i)
```

This will output

```
100
200
300
400
```

The range function is often used to loop a fixed number of times.

```
l = range(1, 6)
for i in l:
    print(i)
```

or

```
for i in range(10, 0, -1):
    print(i)
```

This will output

```
10
9
8
7
6
5
4
3
2
1
```

or

```
for i in range(10, 0, -2):
    print(i)
```

This will output

```
10
8
6
4
2
```

or

```
for i in range(10, 0, -1):
    print i,
```

Python2

```
for i in range(10, 0, -1):
    print(i, end='')
```

Python3

This will output

```
10 9 8 7 6 5 4 3 2 1
```

In for loops as in other contexts we can assign a tuple directly to another tuple, if it loops over a sequence of tuples. For instance

```
l = [(1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
for x, xsquared in l:
    print x, ':', xsquared
```

will output

```
1 : 1
2 : 4
3 : 9
```

```
4 : 16
5 : 25
```

Breaking, continuing and the else clause of loops

Python includes statements to exit a loop (either a for loop or a while loop) prematurely. To exit a loop, use the break statement

```
x = 5
while x > 0:
    print x
    break
    x -= 1
    print x
```

this will output

```
5
```

The statement to begin the next iteration of the loop without waiting for the end of the current loop is 'continue'.

```
l = [5,6,7]
for x in l:
    continue
    print x
```

This will not produce any output.

The else clause of loops will be executed if no break statements are met in the loop.

```
l = range(1,100)
for x in l:
    if x == 100:
        print x
        break
    else:
        print x," is not 100"
else:
    print "100 not found in range"
```

Functions

Function calls

A *callable object* is an object that can accept some arguments (also called parameters) and possibly return an object (often a tuple containing multiple objects).

A function is the simplest callable object in Python, but there are others, such as classes or certain class instances.

Defining functions

A function is defined in Python by the following format:

```
def functionname(arg1, arg2, ...):
    statement1
    statement2
    ...
>>> def functionname(arg1,arg2):
...     return arg1+arg2
...
>>> t = functionname(24,24) # Result: 48
```

If a function takes no arguments, it must still include the parentheses, but without anything in them:

```
def functionname():
    statement1
    statement2
    ...
```

The arguments in the function definition bind the arguments passed at function invocation (i.e. when the function is called), which are called actual parameters, to the names given when the function is defined, which are called formal parameters. The interior of the function has no knowledge of the names given to the actual parameters; the names of the actual parameters may not even be accessible (they could be inside another function).

A function can 'return' a value, like so

```
def square(x):
    return x*x
```

A function can define variables within the function body, which are considered 'local' to the function. The locals together with the arguments comprise all the variables within the scope of the function. Any names within the function are unbound when the function returns or reaches the end of the function body.

Declaring Arguments

Default Argument Values

If any of the formal parameters in the function definition are declared with the format "arg = value," then you will have the option of not specifying a value for those arguments when calling the function. If you do not specify a value, then that parameter will have the default value given when the function executes.

```
>>> def display_message(message, truncate_after = 4):
...     print message[:truncate_after]
...
>>> display_message("message")
mess
>>> display_message("message", 6)
messag
```

Variable-Length Argument Lists

Python allows you to declare two special arguments which allow you to create arbitrary-length argument lists. This means that each time you call the function, you can specify any number of arguments above a certain number.

```
def function(first,second,*remaining):
    statement1
    statement2
    ...
```

When calling the above function, you must provide value for each of the first two arguments. However, since the third parameter is marked with an asterisk, any actual parameters after the first two will be packed into a tuple and bound to "remaining."

```
>>> def print_tail(first,*tail):
...        print(tail)
...
>>> print_tail(1, 5, 2, "omega")
(5, 2, 'omega')
```

If we declare a formal parameter prefixed with *two* asterisks, then it will be bound to a dictionary containing any keyword arguments in the actual parameters which do not correspond to any formal parameters. For example, consider the function:

```
def make_dictionary(max_length = 10, **entries):
    return dict([(key, entries[key])
```

I

f we call this function with any keyword arguments other than max_length, they will be placed in the dictionary "entries." If we include the keyword argument of max_length, it will be bound to the formal parameter max_length, as usual.

```
>>> make_dictionary(max_length = 2, key1 = 5, key2 = 7, key3 = 9)
{'key3': 9, 'key2': 7}
```

Calling functions

A function can be called by appending the arguments in parentheses to the function name, or an empty matched set of parentheses if the function takes no arguments.

```
foo()
square(3)
bar(5, x)
```

A function's return value can be used by assigning it to a variable, like so:

```
x = foo()
y = bar(5,x)
```

As shown above, when calling a function you can specify the parameters by name and you can do so in any order

```
def display_message(message, start=0, end=4):
   print(message[start:end])

display_message("message", end=3)
```

This above is valid and start will be the default value of 0. A restriction placed on this is after the first named argument then all arguments after it must also be named. The following is not valid

```
display_message(end=5, start=1, "my message")
```

because the third argument ("my message") is an unnamed argument.

Closure

Closure, also known as nested function definition, is a function defined inside another function. Perhaps best described with an example:

```
>>> def outer(outer_argument):
...     def inner(inner_argument):
...         return outer_argument + inner_argument
...     return inner
...
>>> f = outer(5)
>>> f(3)
8
>>> f(4)
9
```

Closure is possible in python because function is a first-class object, that means a function is merely an object of type function. Being an object means it is possible to pass function object (an uncalled function) around as argument or as return value or to assign another name to the function object. A unique feature that makes closure useful is that the enclosed function may use the names defined in the parent function's scope.

lambda

lambda is an anonymous (unnamed) function, it is used primarily to write very short functions that is a hassle to define in the normal way. A function like this:

```
>>> def add(a, b):
...    return a + b
...
>>> add(4, 3)
7
```

may also be defined using lambda

```
>>> print (lambda a, b: a + b)(4, 3)
7
```

Lambda is often used as an argument to other functions that expects a function object, such as sorted()'s 'key' argument.

```
>>> sorted([[3, 4], [3, 5], [1, 2], [7, 3]], key=lambda x: x[1])
[[1, 2], [7, 3], [3, 4], [3, 5]]
```

The lambda form is often useful to be used as closure, such as illustrated in the following example:

```
>>> def attribution(name):
...     return lambda x: x + ' -- ' + name
...
>>> pp = attribution('John')
>>> pp('Dinner is in the fridge')
'Dinner is in the fridge -- John'
```

note that the lambda function can use the values of variables from the scope in which it was created (like pre and post). This is the essence of closure.

Scoping

Variables

Variables in Python are automatically declared by assignment. Variables are always references to objects, and are never typed. Variables exist only in the current scope or global scope. When they go out of scope, the variables are destroyed, but the objects to which they refer are not (unless the number of references to the object drops to zero).

Scope is delineated by function and class blocks. Both functions and their scopes can be nested. So therefore

```
def foo():
    def bar():
        x = 5 # x is now in scope
        return x + y # y is defined in the enclosing scope later
    y = 10
    return bar() # now that y is defined, bar's scope includes y
```

Now when this code is tested,

```
>>> foo()
15
>>> bar()
Traceback (most recent call last):
  File "<pyshell#26>", line 1, in -toplevel-
    bar()
NameError: name 'bar' is not defined
```

The name 'bar' is not found because a higher scope does not have access to the names lower in the hierarchy.

It is a common pitfall to fail to lookup an attribute (such as a method) of an object (such as a container) referenced by a variable before the variable is assigned the object. In its most common form:

```
>>> for x in range(10):
         y.append(x) # append is an attribute of lists

Traceback (most recent call last):
  File "<pyshell#46>", line 2, in -toplevel-
    y.append(x)
NameError: name 'y' is not defined
#add y=[] before the loop
```

Functional Programming

List Comprehensions

A list comprehensions creates a list based on an iteration as follows:

```
>>> [x**2 for x in range(5)]
[0, 1, 4, 9, 16]
```

Any variable can be used in place of x. The output of the comprehensions is a list whose elements are the first term of the comprehensions (in this case x^2). The variable x is assigned in the for loop.

Generator expressions

A generators expression is similar to a list comprehensions and is created in exactly the same manner, except that the brackets - [] are replaced with parentheses - (). The output of the generator expression is no a list, but an iterable object that will iteratively return the same elements that the corresponding list comprehension would have returned as a list.

```
>>> (x**2 for x in range(5))
<generator object <genexpr> at 0x7f85af030dc0>
>>> for i in (x**2 for x in range(5)):
...     print(i)
...
0
1
4
9
16
```

The generator expression has the advantage that the entire list is never stored in memory, and is therefore more efficient for a large number of elements.

map()

The **map()** function accepts two arguments, a function and a sequence. In Python2 a list is returned whose members are the result of the function applied to each member of the list supplied. In Python3 an iterable map object is returned instead, which could easily be transformed into a sequence

```
map(lambda x:x**2, range(5))
[0, 1, 4, 9, 16].
```

Python2

```
list(map(lambda x:x**2, range(5)))
[0, 1, 4, 9, 16]
```

Python3

In the following example **map** is used to convert each member of a list to a string using the **str** function. A list of **int** cannot be joined, we need a list of **str**.

```
a>>> l = [1,2,3,4,5]
>>> ','.join(l)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: sequence item 0: expected string, int found
>>> ','.join(map(str,l))
'1,2,3,4,5
```

Generator Functions

A generator function returns a generator object, similar to that returned by generator expressions, **map**, and **dict keys** / **items** / **values**. Generator functions always make use of the key word yield.

The object returned by the generator function has a **__next__** method which make iteration possible. Each time the __next__ method is called, the function executes up until a yield statement and returns the yield value.

Generator functions are best understood by referring to an example.

```
def updown(n):
    for i in range(n): yield i
    for i in range(n,0,-1): yield i

for x in updown(5) : print(x, end='')
0123454321
```

Modules

Modules are a simple way to structure a program. Mostly, there are modules in the standard library and there are other Python files, or directories containing Python files, in the current directory (each of which constitute a module). You can also instruct Python to search other directories for modules by placing their paths in the PYTHONPATH environment variable.

Modules in Python are used by importing them. For example,

```
import math
```

This imports the math standard module. All of the functions in that module are namespaced by the module name, i.e.

```
import math
print(math.sqrt(10))
```

This is often a nuisance, so other syntaxes are available to simplify this,

```
from string import whitespace
from math import *
from math import sin as SIN
from math import cos as COS
from ftplib import FTP as ftp_connection
print sqrt(10)
```

The first statement means whitespace is added to the current scope (but nothing else is). The second statement means that all the elements in the math namespace is added to the current scope.

Modules can be three different kinds of things:

- Python files
- Shared Objects (under Unix and Linux) with the .so suffix
- DLL's (under Windows) with the .pyd suffix
- directories

Modules are loaded in the order they're found, which is controlled by sys.path. The current directory is always on the path.

Directories should include a file in them called __init__.py, which should probably include the other files in the directory.

Creating a DLL that interfaces with Python is covered in another section.

Classes

Classes are a way of aggregating similar data and functions. A class is basically a scope inside which various code (especially function definitions) is executed, and the locals to this scope become *attributes* of the class, and of any objects constructed by this class. An object constructed by a class is called an *instance* of that class.

Defining a Class

To define a class, use the following format:

```
class ClassName:
    ...
    ...
```

The capitalization in this class definition is the convention, but is not required by the language.

Instance Construction

The class is a callable object that constructs an instance of the class when called. To construct an instance of a class, "call" the class object:

```
f = Foo()
```

This constructs an instance of class Foo and creates a reference to it in f.

Class Members

In order to access the member of an instance of a class, use the syntax <class instance>.<member>. It is also possible to access the members of the class definition with <class name>.<member>.

Methods

A method is a function within a class. The first argument (methods must always take at least one argument) is always the instance of the class on which the function is invoked. For example

```
>>> class Foo:
...     def setx(self, x):
...         self.x = x
...     def bar(self):
...         print(self.x)
```

If this code were executed, nothing would happen, at least until an instance of Foo were constructed, and then bar were called on that instance.

Invoking Methods

Calling a method is much like calling a function, but instead of passing the instance as the first parameter like the list of formal parameters suggests, use the function as an attribute of the instance.

```
>>> f.setx(5)
>>> f.bar()
```

This will output

```
5
```

It is possible to call the method on an arbitrary object, by using it as an attribute of the defining class instead of an instance of that class, like so:

```
>>> Foo.setx(f,5)
>>> Foo.bar(f)
```

This will have the same output.

Dynamic Class Structure

As shown by the method setx above, the members of a Python class can change during runtime, not just their values, unlike classes in languages like C or Java. We can even delete f.x after running the code above.

```
>>> del f.x
>>> f.bar()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
  File "<stdin>", line 5, in bar
AttributeError: Foo instance has no attribute 'x'
```

Another effect of this is that we can change the definition of the Foo class during program execution. In the code below, we create a member of the Foo class definition named y. If we then create a new instance of Foo, it will now have this new member.

```
>>> Foo.y = 10
>>> g = Foo()
>>> g.y
10
```

Viewing Class Dictionaries

At the heart of all this is a dictionary that can be accessed by "vars(ClassName)"

```
>>> vars(g)
{}
```

At first, this output makes no sense. We just saw that g had the member y, so why isn't it in the member dictionary? If you remember, though, we put y in the class definition, Foo, not g.

```
>>> vars(Foo)
{'y': 10, 'bar': <function bar at 0x4d6a3c>, '__module__': '__main__',
 'setx': <function setx at 0x4d6a04>, '__doc__': None}
```

And there we have all the members of the Foo class definition. When Python checks for g.member, it first checks g's vars dictionary for "member," then Foo. If we create a new member of g, it will be added to g's dictionary, but not Foo's.

```
>>> g.setx(5)
>>> vars(g)
{'x': 5}
```

Note that if we now assign a value to g.y, we are not assigning that value to Foo.y. Foo.y will still be 10, but g.y will now override Foo.y

```
>>> g.y = 9
>>> vars(g)
{'y': 9, 'x': 5}
>>> vars(Foo)
{'y': 10, 'bar': <function bar at 0x4d6a3c>, '__module__': '__main__',
 'setx': <function setx at 0x4d6a04>, '__doc__': None}
```

Sure enough, if we check the values:

```
>>> g.y
9
>>> Foo.y
10
```

Note that f.y will also be 10, as Python won't find 'y' in vars(f), so it will get the value of 'y' from vars(Foo).

Some may have also noticed that the methods in Foo appear in the class dictionary along with the x and y. If you remember from the section on lambda forms, we can treat functions just like variables. This means that we can assign methods to a class during runtime in the same way we assigned variables. If you do this, though, remember that if we call a method of a class instance, the first parameter passed to the method will always be the class instance itself.

Changing Class Dictionaries

We can also access the members dictionary of a class using the __dict__ member of the class.

```
>>> g.__dict__
{'y': 9, 'x': 5}
```

If we add, remove, or change key-value pairs from g.__dict__, this has the same effect as if we had made those changes to the members of g.

```
>>> g.__dict__['z'] = -4
>>> g.z
-4
```

New Style Classes

New style classes were introduced in python 2.2. A new-style class is a class that has a built-in as its base, most commonly object. At a low level, a major difference between old and new classes is their type. Old class instances were all of type `instance`. New style class instances will return the same thing as x.__class__ for their instance. This puts user defined classes on a level playing field with built-ins. Old/Classic classes are slated to disappear in Python 3000. With this in mind all development should use new style classes. New Style classes also add constructs like properties and static methods familiar to Java programmers.

Old/Classic Class

```
>>> class ClassicFoo:
...     def __init__(self):
...         pass
```

New Style Class

```
>>> class NewStyleFoo(object):
...     def __init__(self):
...         pass
```

Properties

Properties are attributes with getter and setter methods.

```
>>> class SpamWithProperties(object):
...     def __init__(self):
...         self.__egg = "MyEgg"
...     def getEgg(self):
...         return self.__egg
...     def setEgg(self,egg):
...         self.__egg = egg
...     egg = property(getEgg,setEgg)

>>> sp = SpamWithProperties()
>>> sp.egg
'MyEgg'
>>> sp.egg = "Eggs With Spam"
>>> sp.egg
'Eggs With Spam'
>>>
```

Static Methods

Static methods in Python are just like their counterparts in C++ or Java. Static methods have no "self" argument and don't require you to instantiate the class before using them. They can be defined using staticmethod()

```
>>> class StaticSpam(object):
...     def StaticNoSpam():
...         print "You can't have have the spam, spam, eggs and spam without 
any spam... that's disgusting"
...     NoSpam = staticmethod(StaticNoSpam)

>>> StaticSpam.NoSpam()
'You can't have have the spam, spam, eggs and spam without any spam... that's 
disgusting'
```

They can also be defined using the function decorator @staticmethod.

```
>>> class StaticSpam(object):
...     @staticmethod
...     def StaticNoSpam():
...         print "You can't have have the spam, spam, eggs and spam without 
any spam... that's disgusting"
```

Similarly we have class methods, which have their class implicitly passed to them when called. Class methods can also be defined using the function decorator @classmethod.

```
>>> class StaticSpam(object):
...     @classmethod
...     def ClassNoSpam(cls):
...         print "No spam in %s class either" % cls
```

Inheritance

Like all object oriented languages, Python provides for inheritance. Inheritance is a simple concept by which a class can extend the facilities of another class, or in Python's case, multiple other classes. Use the following format for this:

```
class ClassName(superclass1,superclass2,superclass3,...):
    ...
```

The subclass will then have all the members of its superclasses. If a method is defined in the subclass and in the superclass, the member in the subclass will override the one in the superclass. In order to use the method defined in the superclass, it is necessary to call the method as an attribute on the defining class, as in Foo.setx(f,5) above:

```
>>> class Foo:
...     def bar(self):
...         print "I'm doing Foo.bar()"
...     x = 10
...
>>> class Bar(Foo):
...     def bar(self):
...         print "I'm doing Bar.bar()"
...         Foo.bar(self)
...     y = 9
...
>>> g = Bar()
>>> Bar.bar(g)
```

```
I'm doing Bar.bar()
I'm doing Foo.bar()
>>> g.y
9
>>> g.x
10
```

Once again, we can see what's going on under the hood by looking at the class dictionaries.

```
>>> vars(g)
{}
>>> vars(Bar)
{'y': 9, '__module__': '__main__', 'bar': <function bar at 0x4d6a04>,
 '__doc__': None}
>>> vars(Foo)
{'x': 10, '__module__': '__main__', 'bar': <function bar at 0x4d6994>,
 '__doc__': None}
```

When we call g.x, it first looks in the vars(g) dictionary, as usual. Also as above, it checks vars(Bar) next, since g is an instance of Bar. However, thanks to inheritance, Python will check vars(Foo) if it doesn't find x in vars(Bar).

Special Methods

There are a number of methods which have reserved names which are used for special purposes like mimicking numerical or container operations, among other things. All of these names begin and end with two underscores. It is convention that methods beginning with a single underscore are 'private' to the scope they are introduced within.

Initialization

__init__

One of these purposes is constructing an instance, and the special name for this is '__init__'. __init__() is called before an instance is returned (it is not necessary to return the instance manually). As an example,

```
class A:
    def __init__(self):
        print 'A.__init__()'
a = A()
```

outputs

```
A.__init__()
```

__init__() can take arguments, in which case it is necessary to pass arguments to the class in order to create an instance. For example,

```
class Foo:
    def __init__ (self, printme):
        print printme
foo = Foo('Hi!')
```

outputs

```
Hi!
```

Here is an example showing the difference between using __init__() and not using __init__():

```
class Foo:
    def __init__ (self, x):
         print x
foo = Foo('Hi!')
class Foo2:
    def setx(self, x):
        print(x)
f = Foo2()
Foo2.setx(f,'Hi!')
```

outputs

```
Hi!
Hi!
```

Representation

__str__

Converting an object to a string, as with the print statement or with the str() conversion function, can be overridden by overriding __str__. Usually, __str__ returns a formatted version of the objects content. This will NOT usually be something that can be executed.

For example:

```
class Bar:
    def __init__ (self, iamthis):
        self.iamthis = iamthis
    def __str__ (self):
        return self.iamthis
bar = Bar('apple')
print(bar)
```

outputs

```
apple
```

__repr__

This function is much like __str__(). If __str__ is not present but this one is, this function's output is used instead for printing. __repr__ is used to return a representation of the object in string form. In general, it can be executed to get back the original object.

For example:

```
class Bar:
    def __init__ (self, iamthis):
        self.iamthis = iamthis
    def __repr__(self):
        return "Bar('%s')" % self.iamthis
bar = Bar('apple')
bar
```

outputs (note the difference: now is not necessary to put it inside a print)

```
Bar('apple')
```

Attributes

__setattr__

This is the function which is in charge of setting attributes of a class. It is provided with the name and value of the variables being assigned. Each class, of course, comes with a default __setattr__ which simply sets the value of the variable, but we can override it.

```
>>> class Unchangable:
...    def __setattr__(self, name, value):
...        print "Nice try"
...
>>> u = Unchangable()
>>> u.x = 9
Nice try
>>> u.x
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: Unchangable instance has no attribute 'x'
```

__getattribute__

This method intercepts any attempt ro access a class member.

__getattr___

Similar to __getattribute__, except this function is called when we try to access a **non-existant** class member, and the default simply returns the value.

```
>>> class HiddenMembers:
...     def __getattr__(self, name):
...         return "You don't get to see " + name
...
>>> h = HiddenMembers()
>>> h.anything
"You don't get to see anything"
```

__delattr__

This function is called to delete an attribute.

```
>>> class Permanent:
...     def __delattr__(self, name):
...         print name, "cannot be deleted"
...
>>> p = Permanent()
>>> p.x = 9
>>> del p.x
x cannot be deleted
>>> p.x
9
```

Programming Practices

The flexibility of python classes means that classes can adopt a varied set of behaviors. For the sake of understandability, however, it's best to use many of Python's tools sparingly. Try to declare all methods in the class definition, and always use the <class>.<member> syntax instead of __dict__ whenever possible. Look at classes in C++ and Java to see what most programmers will expect from a class.

Encapsulation

Since all python members of a python class are accessible by functions/methods outside the class, there is no way to enforce encapsulation short of overriding __getattr__, __setattr__ and __delattr__. General practice, however, is for the creator of a class or module to simply trust that users will use only the intended interface and avoid limiting access to the workings of the module for the sake of users who do need to access it. When using parts of a class or module other than the intended interface, keep in mind that the those parts may change in later versions of the module, and you may even cause errors or undefined behaviors in the module.

Doc Strings

When defining a class, it is convention to document the class using a string literal at the start of the class definition. This string will then be placed in the __doc__ attribute of the class definition.

```
>>> class Documented:
...     """This is a docstring"""
...     def explode(self):
...         """
...         This method is documented, too! The coder is really serious about
...         making this class usable by others who don't know the code as well
...         as he does.
...
...         """
...         print "boom"
>>> d = Documented()
>>> d.__doc__
'This is a docstring'
```

Docstrings are a very useful way to document your code. Even if you never write a single piece of separate documentation (and let's admit it, doing so is the lowest priority for many coders), including informative docstrings in your classes will go a long way toward making them usable.

Several tools exist for turning the docstrings in Python code into readable API documentation, *e.g.*, EpyDoc.

Don't just stop at documenting the class definition, either. Each method in the class should have its own docstring as well. Note that the docstring for the method *explode* in the example class *Documented* above has a fairly lengthy docstring that spans several lines. Its formatting is in accordance with the style suggestions of Python's creator, Guido van Rossum.

Exceptions

Python handles all errors with exceptions.

An *exception* is a signal that an error or other unusual condition has occurred. There are a number of built-in exceptions, which indicate conditions like reading past the end of a file, or dividing by zero. You can also define your own exceptions.

Raising exceptions

Whenever your program attempts to do something erroneous or meaningless, Python raises exception to such conduct:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
ZeroDivisionError: integer division or modulo by zero
```

This *traceback* indicates that the `ZeroDivisionError` exception is being raised. This is a built-in exception -- see below for a list of all the other ones.

Catching exceptions

In order to handle errors, you can set up *exception handling blocks* in your code. The keywords `try` and `except` are used to catch exceptions. When an error occurs within the `try` block, Python looks for a matching `except` block to handle it. If there is one, execution jumps there.

If you execute this code:

```
try:
    print(1/0)
except ZeroDivisionError:
    print("You can't divide by zero, you silly.")
```

Then Python will print this:

```
You can't divide by zero, you silly.
```

If you don't specify an exception type on the `except` line, it will cheerfully catch all exceptions. This is generally a bad idea in production code, since it means your program will blissfully ignore *unexpected* errors as well as ones which the `except` block is actually prepared to handle.

Exceptions can propagate up the call stack:

```
def f(x):
    return g(x) + 1

def g(x):
    if x < 0: raise ValueError, "I can't cope with a negative number here."
    else: return 5

try:
    print f(-6)
except ValueError:
    print "That value was invalid."
```

In this code, the `print` statement calls the function `f`. That function calls the function `g`, which will raise an exception of type ValueError. Neither `f` nor `g` has a `try`/`except` block to handle ValueError. So the exception raised propagates out to the main code, where there *is* an exception-handling block waiting for it. This code prints:

```
That value was invalid.
```

Sometimes it is useful to find out exactly what went wrong, or to print the python error text yourself. For example:

```
 try:
    the_file = open("the_parrot")
 except IOError, (ErrorNumber, ErrorMessage):
     if ErrorNumber == 2: # file not found
        print "Sorry, 'the_parrot' has apparently joined the choir invisible."
     else:
        print "Congratulation! you have managed to trip a #%d error" %
ErrorNumber  # String concatenation is slow, use % formatting whenever possible
        print ErrorMessage
```

Which of course will print:

```
Sorry, 'the_parrot' has apparently joined the choir invisible.
```

Custom Exceptions

Code similar to that seen above can be used to create custom exceptions and pass information along with them. This can be extremely useful when trying to debug complicated projects. Here is how that code would look; first creating the custom exception class:

```
   class CustomException(Exception):
       pass
```

And then using that exception:

```
try:
    raise CustomException("My Useful Error Message")
except (CustomException, instance):
    print("Caught: " + instance.parameter)
```

Trying over and over again

Recovering and continuing with `finally`

Exceptions could lead to a situation where, after raising an exception, the code block where the exception occurred might not be revisited. In some cases this might leave external resources used by the program in an unknown state.

`finally` clause allows programmers to close such resources in case of an exception. Between 2.4 and 2.5 version of python there is change of syntax for `finally` clause.

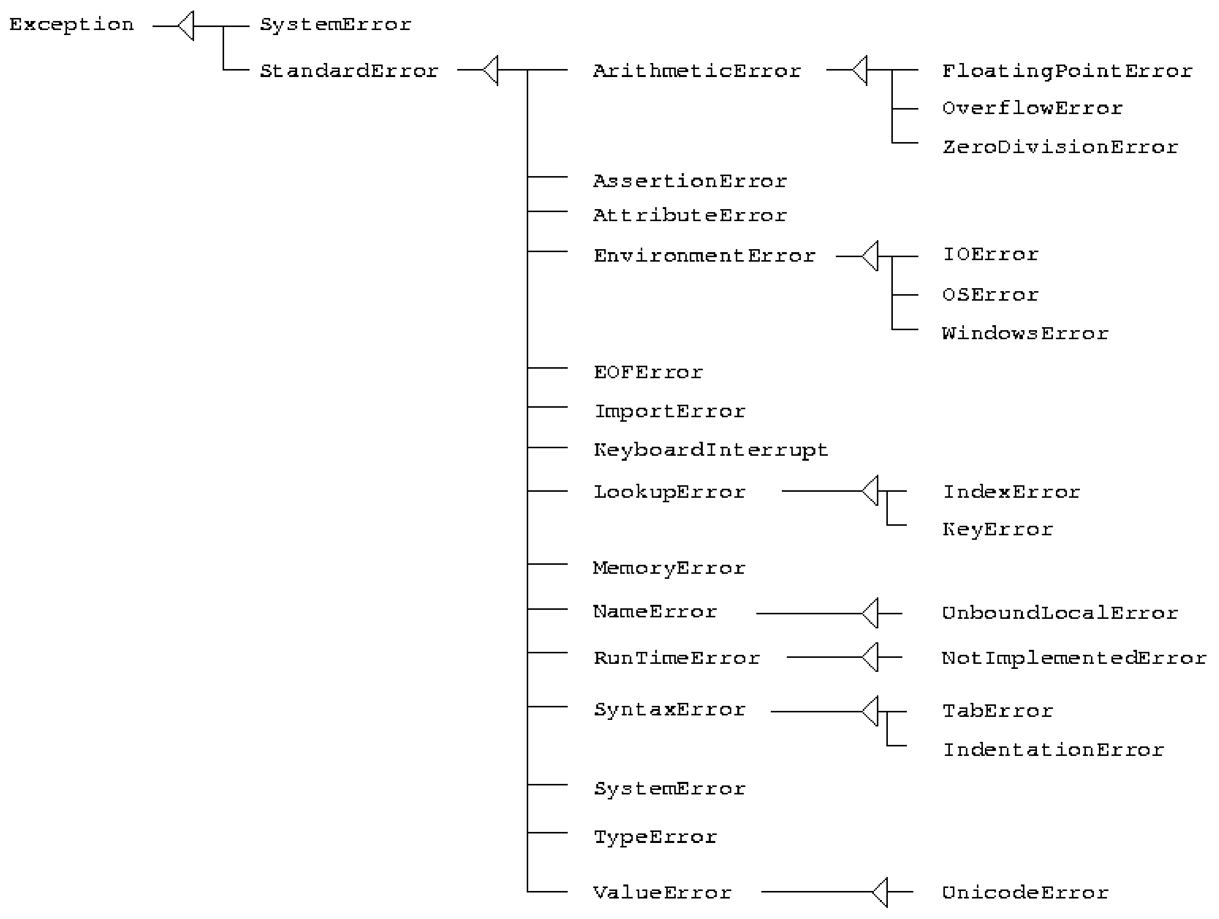
- Python 2.4

```
try:
    result = None
    try:
       result = x/y
    except ZeroDivisionError:
       print "division by zero!"
    print "result is ", result
finally:
     print "executing finally clause"
```

- Python 2.5

```
try:
     result = x / y
except ZeroDivisionError:
     print "division by zero!"
else:
     print "result is", result
finally:
     print "executing finally clause"
```

Built-in exception classes


Exception
SystemError
StandardError
ArithmeticError
FloatingPointError
OverflowError
ZeroDivisionError
AssertionError
AttributeError
EnvironmentError
IOError
OSError
WindowsError
EOFError
ImportError
KeyboardInterrupt
LookupError
IndexError
KeyError
MemoryError
NameError
UnboundLocalError
RunTimeError
NotImplementedError
SyntaxError
TabError
IndentationError
SystemError
TypeError
ValueError
UnicodeError

Input and output

Input

Python2 has twRo functions designed for accepting data directly from the user:

- `input()`
- `raw_input()`

In Python 3 **raw_input** has been renamed **input** and the original **input** function has been deprecated.

There are also very simple ways of reading a file, and for stricter control over input, reading from stdin is necessary.

raw_input() - input() Python3

raw_input() asks the user for a string of data (ended with a newline), and simply returns the string. It can also take an argument, which is displayed as a prompt before the user enters the data. E.g.

```
print raw_input('What is your name?')
```

prints out

```
What is your name? <user input data here>
```

input() (Not available in Python3)

input() uses raw_input to read a string of data, and then attempts to evaluate it as if it were a Python program, and then returns the value that results. So entering

```
[1,2,3]
```

would return a list containing those numbers, just as if it were assigned directly in the Python script.

More complicated expressions are possible. For example, if a script says:

```
x = input('What are the first 10 perfect squares? ')
```

it is possible for a user to input:

```
map(lambda x: x*x, range(10))
```

which yields the correct answer in list form. Note that no inputted statement can span more than one line.

input() should not be used for anything but the most trivial program, turning the strings returned from raw_input() into python types using an idiom such as:

```
x = None
while not x:
    try:
        x = int(raw_input())
    except ValueError:
        print 'Invalid Number'
```

is preferable, as input() uses eval() to turn a literal into a python type. This will allow a malicious person to run arbitrary code from inside your program trivially.

File Input

File Objects

Python includes a built-in file type. Files can be opened by using the file type's constructor:

```
f = file('test.txt', 'r')
```

This means f is open for reading. The first argument is the filename and the second parameter is the mode, which can be 'r', 'w', or 'rw', among some others.

The most common way to read from a file is simply to iterate over the lines of the file:

```
f = open('test.txt', 'r')
for line in f:
    print(line[0])
f.close()
```

This will print the first character of each line. Note that a newline is attached to the end of each line read this way.

Because files are automatically closed when the file object goes out of scope, there is no real need to close them explicitly. So, the loop in the previous code can also be written as:

```
for line in open('test.txt', 'r'):
    print(line[0])
```

It is also possible to read limited numbers of characters at a time, like so:

```
c = f.read(1)
while len(c) > 0:
    if len(c.strip()) > 0: print c,
    c = f.read(1)
```

This will read the characters from f one at a time, and then print them if they're not whitespace.

A file object implicitly contains a marker to represent the current position. If the file marker should be moved back to the beginning, one can either close the file object and reopen it or just move the marker back to the beginning with:

```
f.seek(0)
```

Standard File Objects

Like many other languages, there are built-in file objects representing standard input, output, and error. These are in the sys module and are called stdin, stdout, and stderr. There are also immutable copies of these in __stdin__, __stdout__, and __stderr__. This is for IDLE and other tools in which the standard files have been changed.

You must import the sys module to use the special stdin, stdout, stderr I/O handles.

```
import sys
```

For finer control over input, use sys.stdin.read(). In order to implement the UNIX 'cat' program in Python, you could do something like this:

```
import sys
for line in sys.stdin:
    print(line, end='')
```

Also important is the sys.argv array. sys.argv is an array that contains the command-line arguments passed to the program.

```
python program.py hello there programmer!
```

This array can be indexed,and the arguments evaluated. In the above example, sys.argv[2] would contain the string "there", because the name of the program ("program.py") is stored in argv[0]. For more complicated command-line argument processing, see also(getopt module)

Output

The basic way to do output is the print statement.

```
print 'Hello, world'
```

```
print('Hello, world')
```

This code ought to be obvious.

In order to print multiple things on the same line, use commas between them, like so:

```
print 'Hello,', 'World'
```

This will print out the following:

```
Hello, World
```

Note that although neither string contained a space, a space was added by the print statement because of the comma between the two objects. Arbitrary data types can be printed this way:

```
print 1,2,0xff,0777,(10+5j),-0.999,map,sys
```

This will print out:

```
1 2 255 511 (10+5j) -0.999 <built-in function map> <module 'sys' (built-in)>
```

Objects can be printed on the same line without needing to be on the same line if one puts a comma at the end of a print statement:

```
for i in range(10):
    print i,
```

Or in Python2

```
for i in range(10):
    print(i, end='')
```

will output:

```
0 1 2 3 4 5 6 7 8 9
```

printing without commas or newlines

If it is not desirable to add spaces between objects, but you want to run them all together on one line, there are several techniques for doing that.

concatenation

Concatenate the string representations of each object, then later print the whole thing at once.

```
print(str(1)+str(2)+str(0xff)+str(0777)+str(10+5j)+str(-0.999)+str(map)
+str(sys))
```

will output:

```
12255511(10+5j)-0.999<built-in function map><module 'sys' (built-in)>
```

write

you can make a shorthand for *sys.stdout.write* and use that for output.

```
import sys
write = sys.stdout.write
write('20')
write('05\n')
```

will output:

```
2005
```

You may need sys.stdout.flush() to get that text on the screen quickly.

Regular Expressions

Python includes a module for working with regular expressions on strings. For more information about writing regular expressions and syntax not specific to Python, see the regular expressions wikibook. Python's regular expression syntax is similar to Perl's

To start using regular expressions in your Python scripts, just import the "re" module:

```
import re
```

Pattern objects

If you're going to be using the same regexp more than once in a program, or if you just want to keep the regexps separated somehow, you should create a pattern object, and refer to it later when searching/replacing.

To create a pattern object, use the compile function.

```
import re
foo = re.compile(r'foo(.{,5})bar', re.I+re.S)
```

The first argument is the pattern, which matches the string "foo", followed by up to 5 of any character, then the string "bar", storing the middle characters to a group, which will be discussed later. The second, optional, argument is the flag or flags to modify the regexp's behavior. The flags themselves are simply variables referring to an integer used by the regular expression engine. In other languages, these would be constants, but Python does not have constants. Some of the regular expression functions do not support adding flags as a parameter when defining the pattern directly in the function, if you need any of the flags, it is best to use the compile function to create a pattern object.

The `r` preceding the expression string indicates that it should be treated as a raw string. This should normally be used when writing regexps, so that backslashes are interpreted literally rather than having to be escaped.

Regular expressions consist of: literal characters that match themselves, metacharacters that act as “wild cards”, and quantifiers specify how many times to match the preceding atom (literal, metacharacter or group)

Metacharacters

Metacharacter	Meaning	Regex	Matches
.	Match any character	foo.bar	foodbar

Metacharacter	Meaning	Regex	Matches
^	Match the beginning of line	^hi	lines starting “hi”
$	Match the end of line	hi$	lines ending “hi”
[ABC]	Match A or B or C	h[ae]ll	hall, hell
[^DEF]	Any character except D or E or F	h[^ae]ll	h9ll but not hall
[A-Z]	A character in the range A-Z	A[A-Z]T	ANT, but not AnT
Special Character Sets			
\w	Word characters, same as [A-Za-z0-9_]		
\W	Not word characters, same as [^A-Za-z0-9_]		
\d	Digit same as [0-9]		
\D	Not Digit, same as [^0-9]		
\s	White space, that is space, tab, newline, vspace, hspace etc.		
\S	Not whitespace		
\b	Word break. Matches a word character to non word character transition or vice versa		

Quantifiers

Quantifier	Meaning
*	any no. of times (including O)
+	1 or more times
?	0 or 1 times
{N}	N times
{N,}	N or more times
{N,M}	At least N times at most M times

Grouping and Alternation

For normal regular expressions, the following methods of grouping are available.

Metacharacter	Meaning	Regex	Matches
(whatever)	Treat *whatever* as a single entity		
|	alternation	car|bike	car or bike

Flags

The different flags are:

Flag	Full name	Description
`re.I`	`re.IGNORECASE`	Makes the regexp case-insensitive
`re.L`	`re.LOCALE`	Makes the behavior of some special sequences (`\w, \W, \b, \B, \s, \S`) dependent on the current locale
`re.M`	`re.MULTILINE`	Makes the `^` and `$` characters match at the beginning and end of each line, rather than just the beginning and end of the string
`re.S`	`re.DOTALL`	Makes the `.` character match every character *including* newlines.
`re.U`	`re.UNICODE`	Makes `\w, \W, \b, \B, \d, \D, \s, \S` dependent on Unicode character properties
`re.X`	`re.VERBOSE`	Ignores whitespace except when in a character class or preceded by an non-escaped backslash, and ignores `#` (except when in a character class or preceded by an non-escaped backslash) and everything after it to the end of a line, so it can be used as a comment. This allows for cleaner-looking regexps.

Matching and searching

One of the most common uses for regular expressions is extracting a part of a string or testing for the existence of a pattern in a string. Python offers several functions to do this.

The match and search functions do mostly the same thing, except that the match function will only return a result if the pattern matches at the beginning of the string being searched, while search will find a match anywhere in the string.

```
>>> import re
>>> foo = re.compile(r'foo(.{,5})bar', re.I+re.S)
>>> st1 = 'Foo, Bar, Baz'
>>> st2 = '2. foo is bar'
>>> search1 = foo.search(st1)
>>> search2 = foo.search(st2)
>>> match1 = foo.match(st1)
>>> match2 = foo.match(st2)
```

In this example, match2 will be `None`, because the string `st2` does not start with the given pattern. The other 3 results will be Match objects (see below).

You can also match and search without compiling a regexp:

```
>>> search3 = re.search('oo.*ba', st1, re.I)
```

Here we use the search function of the re module, rather than of the pattern object. For most cases, its best to compile the expression first. Not all of the re module functions support the flags argument and if the expression is used more than once, compiling first is more efficient and leads to cleaner looking code.

The compiled pattern object functions also have parameters for starting and ending the search, to search in a substring of the given string. In the first example in this section, `match2` returns no result because the pattern does not start at the beginning of the string, but if we do:

```
>>> match3 = foo.match(st2, 3)
```

it works, because we tell it to start searching at character number 3 in the string.

What if we want to search for multiple instances of the pattern? Then we have two options. We can use the start and end position parameters of the search and match function in a loop, getting the position to start at from the previous match object (see below) or we can use the findall and finditer functions. The findall function returns a list of matching strings, useful for simple searching. For anything slightly complex, the finditer function should be used. This returns an iterator object, that when used in a loop, yields Match objects. For example:

```
>>> str3 = 'foo, Bar Foo. BAR FoO: bar'
>>> foo.findall(str3)
[', ', '. ', ': ']
>>> for match in foo.finditer(str3):
...     match.group(1)
...
', '
'. '
': '
```

If you're going to be iterating over the results of the search, using the finditer function is almost always a better choice.

Match objects

Match objects are returned by the search and match functions, and include information about the pattern match.

The group function returns a string corresponding to a capture group (part of a regexp wrapped in `()`) of the expression, or if no group number is given, the entire match. Using the `search1` variable we defined above:

```
>>> search1.group()
'Foo, Bar'
>>> search1.group(1)
', '
```

Capture groups can also be given string names using a special syntax and referred to by `matchobj.group('name')`. For simple expressions this is unnecessary, but for more complex expressions it can be very useful.

You can also get the position of a match or a group in a string, using the start and end functions:

```
>>> search1.start()
0
>>> search1.end()
8
>>> search1.start(1)
3
>>> search1.end(1)
5
```

This returns the start and end locations of the entire match, and the start and end of the first (and in this case only) capture group, respectively.

Replacing

Another use for regular expressions is replacing text in a string. To do this in Python, use the sub function.

sub takes up to 3 arguments: The text to replace with, the text to replace in, and, optionally, the maximum number of substitutions to make. Unlike the matching and searching functions, sub returns a string, consisting of the given text with the substitution(s) made.

```
>>> import re
>>> mystring = 'This string has a q in it'
>>> pattern = re.compile(r'(a[n]? )(\w) ')
>>> newstring = pattern.sub(r"\1'\2' ", mystring)
>>> newstring
"This string has a 'q' in it"
```

This takes any single alphanumeric character (\w in regular expression syntax) preceded by "a" or "an" and wraps in in single quotes. The `\1` and `\2` in the replacement string are backreferences to the 2 capture groups in the expression; these would be group(1) and group(2) on a Match object from a search.

The subn function is similar to sub, except it returns a tuple, consisting of the result string and the number of replacements made. Using the string and expression from before:

```
>>> subresult = pattern.subn(r"\1'\2' ", mystring)
>>> subresult
("This string has a 'q' in it", 1)
```

Other functions

The re module has a few other functions in addition to those discussed above.

The split function splits a string based on a given regular expression:

```
>>> import re
>>> mystring = '1. First part 2. Second part 3. Third part'
>>> re.split(r'\d\.', mystring)
['', ' First part ', ' Second part ', ' Third part']
```

The escape function escapes all non-alphanumeric characters in a string. This is useful if you need to take an unknown string that may contain regexp metacharacters like `(` and `.` and create a regular expression from it.

```
>>> re.escape(r'This text (and this) must be escaped with a "\" to use in a
regexp.')
'This\\ text\\ \\(and\\ this\\)\\ must\\ be\\ escaped\\ with\\ a\\ \\"\\\\\\"\\
to\\ use\\ in\\ a\\ regexp\\.'
```

External links

- Python `re` documentation - Full documentation for the re module, including pattern objects and match objects

URLLib, XML and JSON

Accessing Remote Data

The Python **urllib** package has the following modules:

- **urllib.request** for opening and reading URLs
- **urllib.error** containing the exceptions raised by urllib.request
- **urllib.parse** for parsing URLs
- **urllib.robotparser** for parsing robots.txt files

The **urllib.request** module has functions that can open remote files (using HTTP of FTP), that resemble the built in **open** function used to create file objects.

The first step is to import **urllib.request**, and read the help.

```
>>> import urllib.request
>>> help(urllib.request)
```

Opening a JSON file:

```
>>> urllib.request.urlopen("http://leadingtraining.co.za/album.json");
<http.client.HTTPResponse object at 0x7f9f6d761610>
>>> for line in
urllib.request.urlopen("http://leadingtraining.co.za/album.json"): print(line)
...
b'[\r\n'
b'\t{"artist": "Pink"},\r\n'
b'\t{"album": {\r\n'
b'        "2000":"Can\'t Take Me Home",\r\n'
b'\t    "2001":"Misundaztood",\r\n'
b'\t    "2003":"Try This",\r\n'
b'\t    "2006":"I\'m Not Dead"\r\n'
b']\r\n'
b'\r\n'
```

Opening an XML file:

```
>>> for line in
urllib.request.urlopen("http://leadingtraining.co.za/album.xml"): print(line)
...
b'<?xml version="1.0" encoding="UTF-8" ?>\r\n'
b'\r\n'
b'<!-- XML in easy steps - Page 54. -->\r\n'
b'\r\n'
```

```
b'<discography \r\n'
b'xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="album.xsd" >\r\n'
b'\r\n'
b'\t<artist>Pink</artist>\r\n'
b'\t<album id="2000">Can\'t Take Me Home</album>\r\n'
b'\t<album id="2001">Misundaztood</album>\r\n'
b'\t<album id="2003">Try This</album>\r\n'
b'\t<album id="2006">I\'m Not Dead</album>\r\n'
b'\t<album>(...in production)</album>\r\n'
b'\r\n'
b'</discography>\r\n'
b'\r\n'
```

The openurl method returns an **http.client.HTTPResponse** object, which supports the same methods as a file object. Including: read, readline, readlines and can supports iteration in a for loop.

Parsing JSON

```
>>> import json
>>> req = urllib.request.urlopen("http://leadingtraining.co.za/album.json")
>>> json.loads(req.read().decode('utf-8'))
{'album': {'2001': 'Misundaztood', '2000': "Can't Take Me Home", '2003': 'Try
This', '2006': "I'm Not Dead"}, 'artist': 'Pink'}
```

Parsing XML (SAX)

There is a package called **xml** that ships with Python. In this package is a module called **sax** that contains a parser that makes use of the SAX library. When using the sax parser the user defined call back methods are triggered for every XML element that the parser encounters.

Here is an example of a sax handler class for the album.xml file.

```
import xml.sax,re

class AlbumHandler(xml.sax.ContentHandler):
    def startDocument(self):
        print "starting document \n"
        self.artists = {}
        self.lastartist=''
        self.currentTag=''
        self.found = False
    def startElement(self,tag,attributes):
        print "found element %s \n" % tag
        self.currentTag = tag
        if tag == 'album':
            self.artists[self.lastartist].append(attributes.get('id'))
    def characters(self,data):
        if not re.search('^\s*$',data):
            print "found data : %s \n" % data
            if self.currentTag == 'artist':
                self.artists[data] = []
                self.lastartist = data
    def endDocument(self):
        print(self.artists.__repr__())

xml.sax.parseString(req.read().decode('utf-8'),AlbumHandler())
```

XML Element Tree

The xml.etree.ElementTree module parses an XML file and creates a Python object tree structure representing the elements of the XML file.

```
>>> import xml.etree.ElementTree as ET
>>> req = urllib.request.urlopen("http://leadingtraining.co.za/album.xml")
>>> tree=ET.parse(req)
>>> root=tree.getroot()
>>> for child in root:
...    print(child.tag, child.attrib)
...
artist {}
album {'id': '2000'}
album {'id': '2001'}
album {'id': '2003'}
album {'id': '2006'}
album {}
```

GUI Programming

There are various GUI toolkits to start with.

Tkinter

Tkinter, a Python wrapper for Tcl/Tk, comes bundled with Python (at least on Win32 platform though it can be installed on Unix/Linux and Mac machines) and provides a cross-platform GUI. It is a relatively simple to learn yet powerful toolkit that provides what appears to be a modest set of widgets. However, because the Tkinter widgets are extensible, many compound widgets can be created rather easily (i.e. combo-box, scrolled panes). Because of its maturity and extensive documentation Tkinter has been designated as the de facto GUI for Python.

A minimal application

Here is a trivial Tkinter program containing only a Quit button:

```
1.  #!/usr/local/bin/python
2.  from Tkinter import *
3.  class Application(Frame):
4.      def __init__(self, master=None):
5.             Frame.__init__(self, master)
6.             self.createWidgets()
7.      def createWidgets(self):
8.             self.quitButton = Button ( self, text='Quit',
9.                                        command=self.quit )
10.            self.quitButton.grid()
11. app = Application()
12. app.master.title("Sample application")
13. app.mainloop()
```

Explanation

1. Shebang
2. This line imports the entire Tkinter package into your program's namespace.
3. Your application class must inherit from Tkinter's Frame class.
4. App Constructor.
5. Calls the constructor for the parent class, Frame, which is necessary to make the application actually appear on the screen.
6. Calls createWidgets method
7. Defines createWidgets method

8. Creates a button labeled “Quit”.
9. Places the button on the application.
10. The main program starts here by instantiating the Application class.
11. This method call sets the title of the window to “Sample application”.
12. Starts the application's main loop, waiting for mouse and keyboard events.

To learn more about Tkinter visit the following links:

- http://www.astro.washington.edu/owen/TkinterSummary.html <- A summary
- http://infohost.nmt.edu/tcc/help/lang/python/tkinter.html <- A tutorial
- http://www.pythonware.com/library/tkinter/introduction/ <- A reference

PyGTK

PyGTK provides a convenient wrapper for the GTK+ library for use in Python programs, taking care of many of the boring details such as managing memory and type casting. The bare GTK+ toolkit runs on Linux, Windows, and Mac OS X (port in progress), but the more extensive features — when combined with PyORBit and gnome-python — require a GNOME install, and can be used to write full featured GNOME applications.

Home Page

PyQt

PyQt is a wrapper around the cross-platform Qt C++ toolkit. It has many widgets and support classes supporting SQL, OpenGL, SVG, XML, and advanced graphics capabilities. A PyQt hello world example:

```
from PyQt4.QtCore import *
from PyQt4.QtGui import *

class App(QApplication):
    def __init__(self, argv):
        super(App, self).__init__(argv)
        self.msg = QLabel("Hello, World!")
        self.msg.show()

if __name__ == "__main__":
    import sys
    app = App(sys.argv)
    sys.exit(app.exec_)
```

PyQt is a set of bindings for the cross-platform Qt application framework. PyQt v4 supports Qt4 and PyQt v3 supports Qt3 and earlier.

wxPython

Bindings for the cross platform toolkit wxWidgets. WxWidgets is available on Windows, Macintosh, and Unix/Linux.

```
import wx

class test(wx.App):
    def __init__(self):
        wx.App.__init__(self, redirect=False)

    def OnInit(self):
        frame = wx.Frame(None, -1,
                         "Test",
                         pos=(50,50), size=(100,40),
                         style=wx.DEFAULT_FRAME_STYLE)
        button = wx.Button(frame, -1, "Hello World!", (20, 20))
        self.frame = frame
        self.frame.Show()
        return True

if __name__ == '__main__':
        app = test()
        app.MainLoop()
```

- wxPython

Dabo

Dabo is a full 3-tier application framework. Its UI layer wraps wxPython, and greatly simplifies the syntax.

```
import dabo
dabo.ui.loadUI("wx")

class TestForm(dabo.ui.dForm):
        def afterInit(self):
                self.Caption = "Test"
                self.Position = (50, 50)
                self.Size = (100, 40)
                self.btn = dabo.ui.dButton(self, Caption="Hello World",
                      OnHit=self.onButtonClick)
                self.Sizer.append(self.btn, halign="center", border=20)

        def onButtonClick(self, evt):
                dabo.ui.info("Hello World!")

if __name__ == '__main__':
        app = dabo.ui.dApp()
        app.MainFormClass = TestForm
        app.start()
```

- Dabo

pyFltk

pyFltk is a Python wrapper for the FLTK, a lightweight cross-platform GUI toolkit. It is very simple to learn and allows for compact user interfaces.

The "Hello World" example in pyFltk looks like:

```
from fltk import *

window = Fl_Window(100, 100, 200, 90)
button = Fl_Button(9,20,180,50)
button.label("Hello World")
window.end()
window.show()
Fl.run()
```

Other Toolkits

- PyKDE - Part of the kdebindings package, it provides a python wrapper for the KDE libraries.
- PyXPCOM provides a wrapper around the Mozilla XPCOM component architecture, thereby enabling the use of standalone XUL applications in Python. The XUL toolkit has traditionally been wrapped up in various other parts of XPCOM, but with the advent of libxul and XULRunner this should become more feasible.

Sockets

HTTP Client

Make a very simple HTTP client

```
import socket
s = socket.socket()
s.connect(('localhost', 80))
s.send('GET / HTTP/1.1\nHost:localhost\n\n')
s.recv(40000) # receive 40000 bytes
```

NTP/Sockets

Connecting to and reading an NTP time server, returning the time as follows

```
ntpps       picoseconds portion of time
ntps        seconds portion of time
ntpms       milliseconds portion of time
ntpt        64-bit ntp time, seconds in upper 32-bits, picoseconds in lower 32-bits


import socket

BLOCKING = 1            # 0 = non blocking, 1 = blocking
NONBLOCKING = 0         # 0 = non blocking, 1 = blocking
TIME1970            = 2208988800L      # Thanks to F.Lundh
NTPPORT             = 123
MAXLEN              = 1024
NTPSERVER           = ('time.apple.com')
SKTRDRETRYCOUNT         = 2
SKTRDRETRYDLY           = 0.01

#***************************************************
## opensocket(servername, port, blocking) \n
# opens a socket at ip address "servername"

# \arg servername = ip address to open a socket to
# \arg port = port number to use
# ntp uses dgram sockets instead of stream
def opensocket(ipaddr, port, mode):
    # create the socket
    skt = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)

    # open the socket
    try:
        skt.connect((ipaddr, port))
    except socket.error, e:
        print "Failed to connect to server %s %d %d" % (ipaddr, port, mode)
        print "Error %s" % (e.args[0])
        print "Goodbye..."
        sys.exit()

    # set the blocking mode (0=nonblocking, 1=blocking)
    try:
        skt.setblocking(mode)
    except socket.error, e:
        print "Failed to set socket blocking mode for %s %d %d" %(ipaddr, port, mode)
        print "Error %s" % (e.args[0])
        print "Goodbye..."
        sys.exit()
    return(skt)

#***************************************************
##
# we should get 12 long words back in network order \n
# the 10th word is the transmit time (seconds since UT 1900-Jan-01 \n
# I = unsigned long integer \n
# ! = network (big endian) ordering
# \arg \c \b ntpsocket, the socket handle to connect to
# \arg \c \b msg, the message to send to the ntp server
def getntptime(ntpsocket, msg, servername):
    ntpsocket.send(msg)

    rtrycnt = 0
    data = 0
    while (data == 0) & (rtrycnt < SKTRDRETRYCOUNT):
        try:
            data = ntpsocket.recv(MAXLEN)
        except socket.error, e:
            rtrycnt += 1
            print "Error reading non-blocking socket, retries = %s, server = %s" %(rtrycnt,
servername)
            time.sleep(SKTRDRETRYDLY)       # don't retry too often

    # check and see if we got valid data back
    if data:
        ntps = unpack('!12I', data)[10]
        ntpps = unpack('!12I', data)[11]
        if ntps == 0:
            print "Error: NTP, invalid response, goodbye..."
            sys.exit()
    else:
        print "Error: NTP, no data returned, goodbye..."
        sys.exit()

    ntpms = ntpps/5000000L                  # 1ms/200ps, we want ms
    ntpt = (ntps << 32) + ntpps
    return (ntpsocket, ntps, ntpps, ntpms, ntpt)
```

Files

File I/O

Read entire file:

```
inputFileText = open("testit.txt", "r").read()
print inputFileText
```

In this case the "r" parameter means the file will be opened in read-only mode.

Read certain amount of bytes from a file:

```
inputFileText = open("testit.txt", "r").read(123)
print inputFileText
```

When opening a file, one starts reading at the beginning of the file, if one would want more random access to the file, it is possible to use `seek()` to change the current position in a file and `tell()` to get to know the current position in the file. This is illustrated in the following example:

```
>>> f=open("/proc/cpuinfo","r")
>>> f.tell()
0L
>>> f.read(10)
'processor\t'
>>> f.read(10)
': 0\nvendor'
>>> f.tell()
20L
>>> f.seek(10)
>>> f.tell()
10L
>>> f.read(10)
': 0\nvendor'
>>> f.close()
>>> f
<closed file '/proc/cpuinfo', mode 'r' at 0xb7d79770>
```

Here a file is opened, twice ten bytes are read, `tell()` shows that the current offset is at position 20, now `seek()` is used to go back to position 10 (the same position where the second read was started) and ten bytes are read and printed again. And when no more operations on a file are needed the `close()` function is used to close the file we opened.

Read one line at a time:

```
for line in open("testit.txt", "r").readlines():
    print line
```

In this case `readlines()` will return an array containing the individual lines of the file as array entries. Reading a single line can be done using the `readline()` function which returns the current line as a string. This example will output an additional newline between the individual lines of the file, this is because one is read from the file and print introduces another newline.

Write to a file requires the second parameter of `open()` to be "w", this will overwrite the existing contents of the file if it already exists when opening the file:

```
outputFileText = "Here's some text to save in a file"
open("testit.txt", "w").write(outputFileText)
```

Append to a file requires the second parameter of `open()` to be "a" (from append):

```
outputFileText = "Here's some text to add to the existing file."
open("testit.txt", "a").write(outputFileText)
```

Note that this does not add a line break between the existing file content and the string to be added.

Testing Files

Determine whether path exists:

```
import os
os.path.exists('<path string>')
```

When working on systems such as Microsoft Windows(tm), the directory separators will conflict with the path string. To get around this, do the following:

```
import os
os.path.exists('C:\\windows\\example\\path')
```

A better way however is to use "raw", or `r`:

```
import os
os.path.exists(r'C:\windows\example\path')
```

But there are some other convenient functions in `os.path`, where `path.code.exists()` only confirms whether or not path exists, there are functions which let you know if the path is a file, a directory, a mount point or a symlink. There is even a function `os.path.realpath()` which reveals the true destination of a symlink:

```
>>> import os
>>> os.path.isfile("/")
False
>>> os.path.isfile("/proc/cpuinfo")
True
>>> os.path.isdir("/")
True
>>> os.path.isdir("/proc/cpuinfo")
False
>>> os.path.ismount("/")
True
>>> os.path.islink("/")
False
>>> os.path.islink("/vmlinuz")
True
>>> os.path.realpath("/vmlinuz")
'/boot/vmlinuz-2.6.24-21-generic'
```

Common File Operations

To copy or move a file, use the shutil library.

```
import shutil
shutil.move("originallocation.txt","newlocation.txt")
shutil.copy("original.txt","copy.txt")
```

To perform a recursive copy it is possible to use `copytree()`, to perform a recursive remove it is possible to use `rmtree()`

```
import shutil
shutil.copytree("dir1","dir2")
shutil.rmtree("dir1")
```

To remove an individual file there exists the `remove()` function in the os module:

```
import os
os.remove("file.txt")
```

Database Programming

SQLite

SQLite is a C library that provides a lightweight disk-based database that doesn't require a separate server process and allows accessing the database using a nonstandard variant of the SQL query language. Some applications can use SQLite for internal data storage. It's also possible to prototype an application using SQLite and then port the code to a larger database such as PostgreSQL or Oracle.

The sqlite3 module was written by Gerhard Häring. It provides a SQL interface compliant with the DB-API 2.0 specification described by **PEP 249**.

To use the module, you must first create a `Connection` object that represents the database. Here the data will be stored in the `example.db` file:

```
import sqlite3
con = sqlite3.connect('test.db')
```

You can also supply the special name `:memory:` to create a database in RAM.

Once you have a `Connection`, you can create a `Cursor` object and call its `execute()` method to perform SQL commands:

```
import sqlite3 as db
con = db.connect('test.db')
cur = con.cursor()
#create a string with sql create table code
sql = '''CREATE TABLE users(
            id INT PRIMARY KEY,
           username VARCHAR(64),
            password VARCHAR(64))'''
#execute the sql code
cur.execute(sql)

#an insert rows into the table using a parameterised tables
sql = '''INSERT INTO users
         VALUES (?,?,?)'''
#execute the query with one row
cur.execute(sql,[44,'Billy','1234'])
#execute many rows as a list of tuples
cur.executemany(sql,[(55,'AA','1'),(56,'BB','2')])
#sqlite does not auto commit
con.commit()
sql = "SELECT * FROM users"
cur.execute(sql)
print cur.fetchall()
```

Postgres connection in Python

```
import psycopg2
conn = psycopg2.connect("dbname=test")
cursor = conn.cursor()
cursor.execute("select * from test");
for i in cursor.next():
    print i
conn.close()
```

SQLAlchemy in Action

SQLAlchemy has become the favorite choice for many large Python projects that use databases. A long, updated list of such projects is listed on the SQLAlchemy site. Additionally, a pretty good tutorial can be found there, as well. Along with a thin database wrapper, Elixir, it behaves very similarly to the ORM in Rails, ActiveRecord.

Generic Database Connectivity using ODBC

The Open Database Connectivity (ODBC) API standard allows transparent connections with any database that supports the interface. This includes most popular databases, such as PostgreSQL or Microsoft Access. The strengths of using this interface is that a Python script or module can be used on different databases by only modifying the connection string.

There are three ODBC modules for Python:

1. PythonWin ODBC Module: provided by Mark Hammond with the PythonWin package for Microsoft Windows (only). This is a minimal implementation of ODBC, and conforms to Version 1.0 of the Python Database API. Although it is stable, it will likely not be developed any further.[3]
2. mxODBC: a commercial Python package (http://www.egenix.com/products/python/mxODBC/), which features handling of DateTime objects and prepared statements (using parameters).
3. pyodbc: an open-source Python package (http://code.google.com/p/pyodbc), which uses only native Python data-types and uses prepared statements for increased performance. The present version supports the Python Database API Specification v2.0.[4]

pyodbc

An example using the `pyodbc` Python package with a Microsoft Access file (although this database connection could just as easily be a MySQL database):

```
import pyodbc

DBfile = '/data/MSAccess/Music_Library.mdb'
conn = pyodbc.connect('DRIVER={Microsoft Access Driver (*.mdb)};DBQ='+DBfile,
autocommit=True)
cursor = conn.cursor()

SQL = 'SELECT Artist, AlbumName FROM RecordCollection ORDER BY Year'
cursor.execute(SQL)
for row in cursor: # cursors are iterable
    print row.Artist, row.AlbumName

cursor.close()
conn.close()
```

Many more features and examples are provided on the pyodbc website.

Appendix

Licence

"Acknowledgements", "Dedications", "Endorsements", or "History".) To "Preserve the Title" of such a section when you modify the Document means that it remains a section "Entitled XYZ" according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING
You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY
If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document's license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before

their titles. Section numbers or the equivalent are not considered part of the section titles.
M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.
N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.
O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties--for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS
You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled "History" in the various original documents, forming one section Entitled "History"; likewise combine any sections Entitled "Acknowledgements", and any sections Entitled "Dedications". You must delete all