

## Python

Copyright (c) 2006, Simon Yuill and Harry Halpin 2006

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

*The programmer, like the poet, works only slightly removed from pure thought-stuff. He builds castles in the air, from air, creating by exertion of the imagination. Few media of creation are so flexible, so easy to polish and rework, so readily capable of realizing grand conceptual structures. Yet the program construct, unlike the poet's words, is real in the sense that it moves and works, producing visible outputs separate from the construct itself. It prints results, draws pictures, produces sounds, moves arms. The magic of myth and legend has come true in our time.*

Fred Brooks, author of The Mythical Man-Month

Guide to text formatting in examples:

plain fixed-width text

*italic fixed-width text*

<**BOLD**>

plan text

code, or commands to enter

items that can be changed

keyboard keys to press, or mouse actions to perform

output from python shell

### Why Python?

**Python** is just one programming language of many. Just like human languages, there are many different computer languages, such as **Java**, **LISP**, **PHP**, and **Perl** and...let's not forget C or others, as well as useful things like UNIX scripting. Most languages are good at least one thing – for example, writing easily portable programs is a strong-point for Java and accessing databases and splicing them into web-pages is the specialism for PHP. But underneath, all these languages are very similar at the core concept level – most have data in variables and functions (procedures, methods) to do stuff to that data. Some languages even combine data and functions into bundles called objects, and others like LISP let you treat functions like variables, and vice versa. **Python** is a powerful, elegant programming language that is easy to read and to understand. It demonstrates most of these features common to lots of other languages and is useful for real-world applications, to boot! It's also free software, has one standard implementation, and a large and friendly community of hackers around it. Once you learn Python, every other language you want to learn should seem pretty familiar...

### General Programming Language Concepts:

Programming languages consist of **code**, that is a bunch of sentences written for the computer that tell it what to do. Code's not magic – in fact, it looks like some bizarre bastard child of English and logic often enough. However, unlike **natural languages** such as English or Chinese, **formal languages** like **Python** strive to eliminate ambiguity. They have to eliminate ambiguity, because your code is supposed to tell *the computer exactly what to do*. You don't want your computer making any off-the-cuff decisions for you, do you? Like any foreign language, once you read code enough, any piece of code you find will start making sense, and if it doesn't, with a little bit of Internet research and a good reference book or two, the code will make sense. Still, the computer **does not directly read your code**. It has to have it translated into something even more indecipherable to mere mortals - **machine code** of zeros and ones, register commands, and so on. To do this you need an **interpreter**, a program that lets you type in and run code (or, a program that makes programs!). Interpreters let you type and run code immediately. **Compilers** let you turn your code directly into machine code to run later.

### Starting Python

*A programming language is for thinking of programs, not for expressing programs you've already thought of. It should be a pencil, not a pen.*

Paul Graham, author of Hackers and Painters

Starting the Python interactive shell that lets you start the interpreter consists of finding a command prompt and typing:

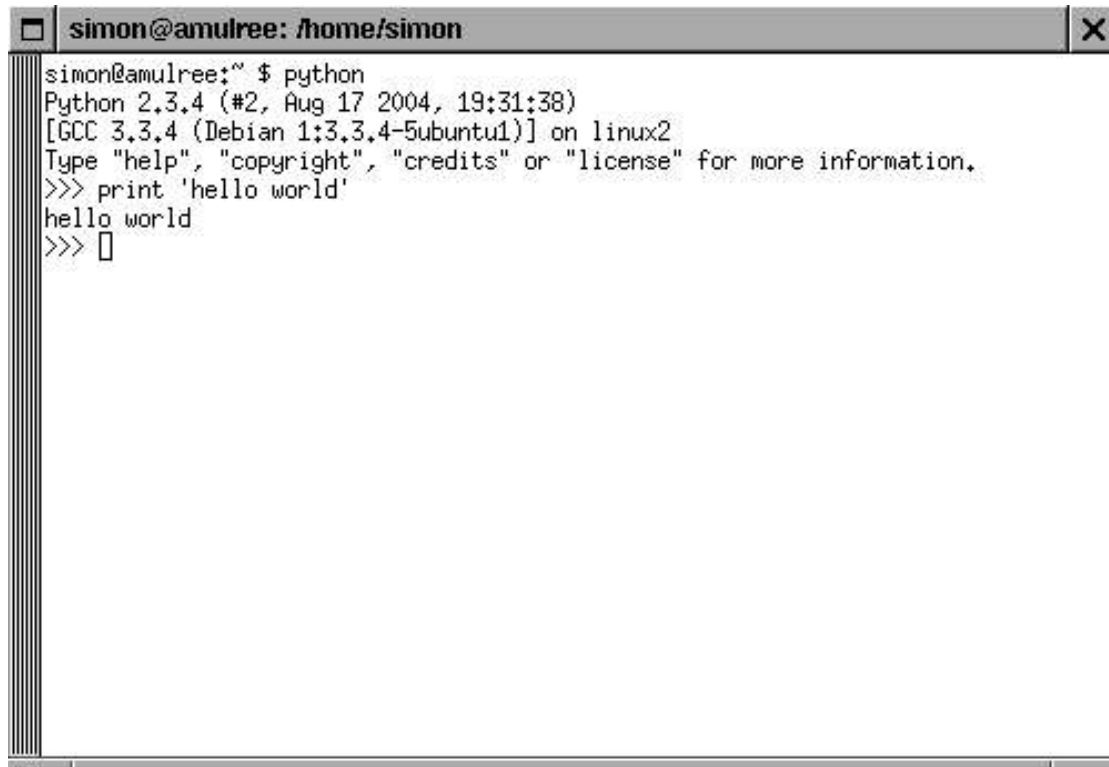
python <RETURN>

Running uncompiled Python code in the Python shell (usually ends with **.py**):

python *scriptname* <RETURN>

Running an executable (compiled) Python script:

*./scriptname* <RETURN>

A terminal window titled "simon@amulree: /home/simon" with a close button in the top right corner. The terminal shows the following text:

```
simon@amulree:~ $ python
Python 2.3.4 (#2, Aug 17 2004, 19:31:38)
[GCC 3.3.4 (Debian 1:3.3.4-5ubuntu1)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> print 'hello world'
hello world
>>> █
```

## **Variables**

*Everyone will need computer programming. It is the way we will speak to the servants.*

John McCarthy, rather hierarchical quote by the inventor of LISP programming language

A variable is a placeholder for a value of any kind that you are working with in your code. The important thing about variables is that the value they have **varies** over the time of the running of a program, therefore the name “variable.” However, the great thing of about variables is that they keep their value until changed. What most people think of as “data” is generally stored in variables or collections of variables. Most of what you will be doing programming is moving value around variables and then doing things with them, like printing them to the screen.

Setting a variable's value:

```
variableName = value
```

Examples:

```
height = 400  
locX = 0.56  
name = 'harry'
```

Using a variable:

```
locX = 0.56  
velocityX = 0.2  
newLocX = locX + velocityX
```

## **Data types**

*The purpose of computing is insight, not numbers.*

Richard Hamming, inventor of the Hamming error-correcting codes

### **Integers and Floats**

Integers and floats are just numbers. Integers are whole numbers, ie: 0,1,2,3, etc. Floats are decimal numbers, ie: 0.5, 34.666, etc.

The main commands you can use with integers and floats are: add, subtract, multiply, divide, and modulus (the remainder of the first number divided however many times by the second number). Modulus is great for discovering if things are even – so where any number modulus 2 is zero, that number is even. There are a few tricks – watch out for dividing integers! The result of dividing two integers has to be an integer, so you lose any fraction bits.

Adding numbers:

```
3 + 7  
10  
  
0.5 + 6  
6.5
```

Subtracting numbers:

```
7 - 3  
4  
  
0.5 - 6  
-5.5
```

Multiplying numbers:

```
4 * 3
12
```

Dividing numbers:

```
12 / 4
3
```

```
12 / 5
2
```

```
12 / 5.0
2.399999999999
```

Modulus:

```
6 % 10
6
```

```
12 % 10
2
```

```
10 % 10
0
```

## Tuple

A tuple is a simple list of values. Once it has been created, its values cannot be changed. Another word for this that you sometimes see is “**immutable.**” This means that there are things that can be changed, like the **atomic** (non-list!) variables we created earlier, that are **mutable**.

Creating a tuple:

```
myTuple = ('a','b','c', 'd', 'e')
```

To get values from a tuple you have to type in an “index” after the name surrounded by brackets. Computers, always trying to save space, start their indices with **zero**, not one. Unusually, Python also lets you count indices backwards using negative numbers, and grab whole chunks using the semi-colon in between the beginning index and end index of the sub-tuple you want.

```
myTuple[0]
'a'
```

```
myTuple[-1]
'e'
```

```
myTuple[1:3]
('b', 'c', 'd')
```

Tuples can contain various types of data:

```
myTuple = ('a','33.5','harry', myObject, None)
```

## Lists

A list is like a tuple but allows you to change the values of its contents and add new values to it. It is **mutable**. The big difference in distinguishing lists from tuples is that when you make a list, you've got to use **square brackets**, not parentheses like tuples..

Creating a list:

```
myList = []  
myList = ['a','b','c']
```

Adding items to a list. Note that there's this period in between the name of your list and the command you're using to "append" items to the list. This shows that this command is attached not to just any list, but to your list! Also, you can use commands like "print" to view the contents of a list, or any other variable to the screen. The object of a command like "print myList" is to print exactly the contents of that list, and nothing else. (like "Run Jack" tells not anyone, but just Jack, to run!)

```
myList.append('d')  
print myList  
['a','b','c', 'd']
```

You can also subtract items from a list by using the "remove" command after the name of the list (remember to put in the period!) and then giving the command the value you wish to remove from the list:

```
myList.append('e')  
['a','b','c','d'],'e'  
print myList.remove('e')  
['a','b','c','d']
```

Changing values in a list is just assigning another value to the variable using the index:

```
myList[2] = 'x'  
print myList  
['a', 'b', 'x', 'd']
```

There's tons of neat stuff you can do with lists built right in. You can reverse a list, or sort a list by alphabetical or numerical order. See what these commands do!

```
myList.reverse()  
myList.sort()
```

## String

A string is a piece of text. Strings can be treated a bit like lists, since they are basically lists of characters (members of the alphabets and numbers mostly).

Creating a string:

```
myString = 'hello harry'
```

Getting letters from a string:

```
myString[7]  
'a'  
  
myString[6:8]  
'ha'
```

Strings can be added together (known as 'concatenating a string'):

```
myString += ' and sally'  
print myString  
'hello harry and sally'
```

## Dictionary

A dictionary is a list of items that can be located by a name, or 'key' value. Just like looking up definitions in a dictionary! Sort of like an index, but not a number.

Creating a dictionary:

```
myDictionary = {}
```

Setting a value in a dictionary:

```
myDictionary['key'] = value
```

Getting a value from a dictionary:

```
myDictionary['key']
```

```
myDictionary.get('key', None)
```

## **None**

None is a special value that represents nothing, it can be used to show that a variable (or object...but we're not there quite yet!) exists but has no value:

```
myVariable = None
```

## **Built-in commands**

*The Analytical Engine weaves Algebraical patterns just as the Jacquard loom weaves flowers and leaves.*

Ada Augusta, daughter of Byron and first computer programmer!

Python comes with tons of commands built in you can use. These can save you lots of time. Now many of these you could program yourself, but why bother?

### **Displaying information in the console:**

```
print 'hello'
```

Formatting print statements. The weird percent sign tells the string to “stick the variable value” here, and the letter right after the percent sign tells it how to format it, as in whether or not it should have trailing zeros if it's a number. The value (or variable with that value) that you then want to print has to be preceded by another percent sign. If you are printing more than one value then you want to have your values in a tuple list.

```
%s - prints string or object  
%d - prints integer (whole number)  
%f - prints float (decimal number)
```

Example:

```
print "hello %s, %d is an integer, %f is a float" % ('harry', 10, 0.5)  
'hello harry, 10 is an integer, 0.5 is a float'
```

### **Number functions:**

float – make sure number is float, or converts string to float value

```
float('0.5')  
0.5
```

```
float('p')  
Traceback (most recent call last):  
  File "<stdin>", line 1, in ?  
ValueError: invalid literal for float(): p
```

max - return maximum value

```
max(8,6)  
8
```

int - makes sure number is integer, or converts string to integer value

```
int('3')  
3
```

min - return minimum value

```
min(8,6)  
6
```

abs - return absolute value

```
abs(6 - 8)  
2
```

### File functions:

A file is just something on your hard drive, and you can create those yourself from programs. They are great, because as you've noticed as soon as you exit the Python interpreter all your hard work disappears into the cybernetic ether, never to be seen again. Instead, you can write that important variables down in a file, and then read that file again.

Opening a file. The mysterious 'r' stands for 'read', and opening assigns a filename like 'myfile.txt' and a mode like 'reading' with the 'file' variable. Then you can read in commands from the file using its read command.

```
file = open(filename, 'r')  
content = file.read()  
print content  
file.close()
```

Writing to a file is just as simple, but instead of the mysterious 'r' for 'read' mode, you want to use a 'w' for writing mode, and then write to the file with the file's write command.

```
file = open(filename, 'w')  
content = 'blah blah'  
file.write(content)  
file.close()
```

You can write your Python code in a text file. This makes it much easier to write multi-line commands and complex programs than just staying in the interpreter. To run them, you just type in a program into your favorite text editor, such as emacs, vi, or even gedit (just make sure you can save as plain text, not with fancy formatting!). Note we can use the raw\_input command to get data from the user.

```
print 'Hello! What's your name?'
yourname = raw_input('Enter your name, then press Enter:')
print 'Nice to meet you, %s. Hope you enjoy Python!' % yourname
```

To run just type: `python myfile.py`

If something went wrong, the Python interpreter will kindly tell you the location in terms of line numbers. Go there, and try to figure out what went wrong by checking your code.

### **Control statements**

*Controlling complexity is the essence of computer programming.*

B. Kernighan, co-creator of the C programming language

Control statements give structure to a program. They enable decisions to be made and actions to be looped. In essence, every control statement consists of a condition, which is either true or false. These conditions are things like:

```
myString == 'harry'
```

which if you have the statement: `myString = 'harry'`

Somewhere, this should be true. These conditions can involve numbers, like

```
temp > 45
```

It is very important that you put the colon (:) after the condition in a list command, and that the text after a list command be formatted by indenting it EXACTLY one indentation over for every loop you are in. Then you exit the code by just looping out by just removing the extra indentation.

```
loop <condition>:
    <doing something in condition>
    <doing something as well in loop>
<outside loop now>
```

**If-else:** making decisions among. Usually when one thing is true, and the other is a default mode. You only have to make the decision once.

```
if <condition>:
    <do something>
else:
    <do something else>
```

```
if <condition>:
    <do something>
elif <condition>:
    <do something different>
else:
    <do something else>
```

**For:** Looping through a series of items, i.e you want to keep doing something so many times, and then stop. Often you want to do something a number of times times, in which case there is a nice magic command called “range” that will, given the number 10, produce a list of [1,2,3,4,5,6,7,8,9,10]. You can also do something to every member of the list as well if the list is not composed of numbers.

```
for i in range(<number>):
    <do something>
```



```
for item in list:  
    <do something with item>
```

```
for key, value in dictionary.items():  
    <do something with with key and value>
```

**While:** Looping until something changes. This is great for doing things when you don't know how many times you know what you want.

```
while <condition>:  
    <do something>
```

Here's a simple example that prints the Fibonacci sequence up till 2000, that magic number that gives us the Golden Mean and classical architecture.

```
a = 0  
b = 1  
while b < 2000:  
    print b  
    a = b  
    b = a+b
```

However, you must beware of the dreaded **infinite loop**, a loop that never ends. If you change your condition from "a < 2000" to "b > 0" you will see what I mean! Exit from the never-ending program by typing CTRL-C.

### Logic operators

Logic operators can be used in combination with control statements. They are primarily used to test if things are true or false.

not – test if an item is false or has no value.

and – test if two items are true

or – test if one of two items is true

You can quickly build up very complicated conditions using conditionals, such as when someone does not have the name George and is over age 18 or has less than 10 pounds. It's good to keep them separated with parentheses so you remember what your doing! The computer checks the conditional in the innermost parentheses first, and slowly builds outwards to figure out if the condition is true or false.

```
(name == "George") and ((age > 18) or (pounds < 10))
```

### Comments

*Programs must be written for people to read, and only incidentally for machines to execute.*

H. Abelson and G. Sussman in [The Structure and Interpretation of Computer Programs](#)

Comments are lines in a program which are ignored when running the program. They can used to put notes into a program. This is important because often you forget what you're doing, and if you want other programmers to understand your code you more or less have to do this. :

```
# this bit doesn't quite work yet!  
velX = locX % 45
```

Or temporarily disable some lines of code. This is great when something isn't working, so you have to "debug" it. You can also even comment out your entire program and comment it out bit by bit to isolate which parts of it have problems.:

```
#velX = locX % 45
```

A comment only applies to the line it start on. Multiple lines of test need to have a comment on the start of each line:

```
if not partitionName: return
partition = self.createPartitionFromName(partitionName)
#partition.addStructure(structure)
#self.bounds.addPoint(partition.bounds.minX, partition.bounds.minY)
#self.bounds.addPoint(partition.bounds.maxX, partition.bounds.maxY)
```

## **Functions**

*Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it in the same way twice.*

-C. Alexander

Functions allow you to bundle a series of code statements into a form that can be re-used. You can send data to a function and get it to send back the result of whatever it does to that data. All the commands we've been talking about, like "print" and "raw\_input" are actually functions.

Defining a function. Remember to use "def" before the function name, and to always use parenthesis after the function when defining it.:

```
def functionName(inputValue):
    <do something>
```

Calling a function:

```
functionName(9)
functionOne(functionTwo(9))
```

The data you send to a function is defined by the function 'parameters', these are like sockets or funnels which identify what is being sent to a function. A parameter is defined as a name which is then used in the function code. Note that in Python when you create a new variable inside a function, once you are finished with the function that variable is destroyed. However, variables defined outside functions can be accessed from anywhere. Often you want your function to return just one particular value, like this:

```
myVariable = functionName(100)
```

This can be done with the "result" keyword, followed by the name of the variable.

```
def functionName(inputValue):
    result = inputValue * 5
    return result
```

Python also has 'doc strings'. these can be used to add explanations to the code which can be called up in the Python shell:

```
def funkshun(value):
    """
    This function just prints whatever is sent to it.
    """
    print "funkshun:", value
```

```
funkshun.__doc__
'This function just prints whatever is sent to it.'
```

## Objects

*Classes struggle, some classes triumph, others are eliminated.*

Mao Zedong

An object is like a more elaborate type of function that combines groups of functions with the data they operate on. Whilst a function just processes data, an object can store data as well as process it. Object allow you think about a program in a very metaphorical way. You can think of a group of objects as team of people each doing different jobs.

The code that defines an object is called a 'class'. This is like the abstract template and the object is like an 'active' version of it. In philosophical terms, a class is a type, and objects are tokens of a type.

Defining a class:

```
class ClassName:
    def __init__(self):
        print "class created"
```

The init function (which is surrounded by two underscores so you know its special) is run every time you create a new object.

Creating an object from a class consists of using the name of the class like a function:

```
c = ClassName()
```

When an object stores data this is called a 'property' of the object. These are a special form of variable that belongs to the object. They are always written:

```
self.propertyName
```

The word 'self' shows that the property belongs to the object. This is very different than the way things are done in other programming languages in objects.

Objects have their own functions, often these are called 'methods' in order to separate them from functions, but they perform the same role. However, make sure you don't forget that the first argument to every function is an implicit "self" variable, and that this variable isn't needed when you call the function from the object.

Defining a method:

```
def methodName(self, inputValue):
    do something
    return result
```

Calling a method:

```
c.methodName(9)

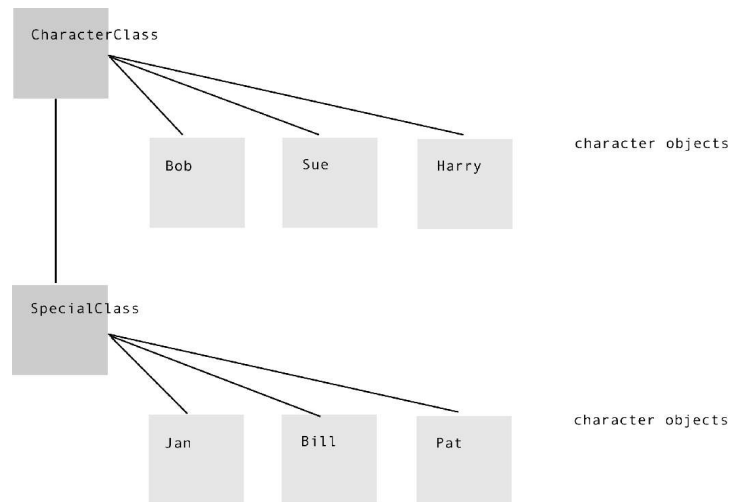
c.methodName(functionOne(9))
```

This should start helping make like files make sense, like file input and output and even adding things and subtracting things from lists. When you "open" a file ("myFile = open('myfile.txt', 'r')") you pass the open function, the name of the file and a character like "r" for "read" that determines its parameters, and in return the built-in "open" function returns. Then when you want to read. In fact, not only are files and lists objects, all of Python consists of objects. Some, like numbers, just aren't very complicated!

New classes for objects can also be created by adapting an existing class. This is known as 'inheritance'. This allows you to take a class and refine its functions or add new ones. In philosophy and biology we can use these to create sort of taxonomies or ontologies, with the most abstract type (Animals can move) having its properties inherited by the progressively less abstract types (Mammals have warm blood, Mammls are animals, therefore mammals can move too!):

```
class ClassTwo(ClassOne):  
    def __init__(self):  
        ClassOne.__init__(self)
```

Classes and objects:



## Modules

*Good programmers know what to write. Great ones know what to use.*

Eric Raymond, author of Hackers and Painters

Modules are individual Python scripts containing a set of classes and functions. These are useful to bundle up, and these allow you to extend the language by importing new modules. If you write some general purpose code that you think another Python programmer might use, you too can bundle it up as a module.

Anatomy of a module:

<pre>import math from time import time</pre>	import statements for module
<pre>mCount = 0 mName = 'harry'</pre>	module variables
<pre>def saySomething(words):     """     This function makes the character speak.     """     print "%s says: '%s'" % (mName, words)     mCount += 1</pre>	function
<pre>class MyClass:     """     This is a simple object.     """     def __init__(self, name):         """         This is the object constructor.         """         self.name = name         self.count = 0         print "%s has been created." % self.name      def saySomething(self, words='boo'):         """         This function makes the character speak.         """         print "%s says: '%s'" % (self.name, words)         self.count += 1</pre>	object

**Importing a module:** Note that you only do this once from a file, but in an interpreter if you load a module and change something, and want to go back to the original, you have to “reload(module)”.

```
import module
import module as x
from module import x
```

**Getting info from a modules**

```
import random
```

```
dir(random)
['BPF', 'LOG4', 'NV_MAGICCONST', 'Random', 'SG_MAGICCONST', 'TWOPI',
'WichmannHill', '_BuiltinMethodType', '__all__', '__builtins__', '__doc__', '__file__',
'__name__', '_acos', '_cos', '_e', '_exp', '_floor', '_inst', '_log', '_pi', '_random', '_sin',
'_sqrt', '_test', '_test_generator', 'betavariate', 'choice', 'cunifvariate', 'expovariate',
'gammavariate', 'gauss', 'getstate', 'jumpahead', 'lognormvariate', 'normalvariate',
'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle',
'stdgamma', 'uniform', 'vonmisesvariate', 'weibullvariate']
```

```
random.randint.__doc__
```

'Return random integer in range [a, b], including both end points.'

### Displaying a module help guide:

```
help(modulename)
```

### Making an executable script.

This must be the first line in the script (or at least it should be!):

```
#!/usr/bin/env python
```

To define code that should be automatically executed when the script starts:

```
if __name__ == '__main__':
    script statement go here
```

On Unix (Linux and OS X) systems, to make the script executable, enter the following command in a standard terminal:

```
chmod 755 modulename
```

To run the module script:

```
./modulename
```

### XML

*Every new level of abstraction draws the computer-based world closer to the concepts we talk about in the real world. We've moved from waves to bits to data to information to infosets to application objects. As this process continues, some ambitious Comp Sci graduate student will realize that somebody already created the tree structure mapping the highest level of reality. That person was, of course, G. W. F. Hegel.*

Frank Wilson, editor at O'Reilly

**XML** stands for eXtensible Markup Language, and it's sort of a universal data format for transferring things around the Web. It's basically a clean-up version of HTML (Hypertext Markup Language) that many of use. The key difference is that with XML, instead of being stuck with the names that HTML gives you, you can make up your own. This allows you to design custom markup vocabulary for describing all sorts of data.

XML data must be well-formed. HTML, at least the sloppy HTML that many of use, does not have to be well formed. For example, we often do things like this:

```
<b>The Meaning of Life in Paragraph</b>
  <p> The answer is <a href = "http://en.wikipedia.org/wiki/Meaning_of_life">42</a>.
```

That's silly & now if we could only find “the question”...  
<b>The Question is!</b>

While this would be perfectly acceptable as HTML for the purposes of having a web browser render it, it is not acceptable as XML (of XHTML, the XML-conformant HTML that you should use!). In both XML and HTML there are tags, such as “b” for bold and “p” for paragraph. Tags always start with <tag>, but they should end with </b>. In XML, every tag must come in pairs, such that for every start-tag there is an end-tag, where the end-tag is not only has the same name but also has the same name preceded by a backwards-slash. Therefore the above code is not well-formed XML until it has the end tag of the <p> paragraph tag in the right place. Also, tags must not nest in XML. This means that you cannot do this:

```
<tag1> My <tag2> content <tag1> is here </tag2>
```

But instead must do this:

```
<tag1> My <tag2> content <tag2> is here </tag1>
```

Also, certain characters are not allowed in XML because they are given special meaning by XML to define tags. The most common of these are quotes, apostrophes, ampersands, and less than and greater than symbols. These can then be replaced by &quot;, &apos;, &lt; and &gt; respectively. This proceeding a character with an ampersand character points to the XML parser that there is actually some sort of special character there, and this can in turn be used to insert Unicode characters in XML that are otherwise unprintable – try putting &#165; in your XML document.

You also have to give XML code (or at least you should) a prolog, which tells the parser what version of XML its reading and what character set its using.

```
<?xml version="1.0" encoding="iso-8859-1"?>
```

means that you are using XML version 1.0 and the ISO 8859 character set. In general, while they often are similar for English, there are big differences between character sets such as printing accents differently, so it's generally more common in the XML world to use UTF-8, or <?xml version="1.0" encoding="utf-8"?>. Also, every XML document must have a root node, or a node on top that encompasseses the entire document except the prolog. Now we can convert it to XML by formatting it like this:

```
<?xml version="1.0" encoding="utf-8"?>
<root>
<b>The Meaning of Life in Paragraph</b>
  <p> The answer is <a href = "http://en.wikipedia.org/wiki/Meaning_of_life">42</a>.
  That&apo; silly &amp; now if we could only find &quot;the question&quot;...</p>
<b>The Question is!</b>
</root>
```

The tags like <b> are called “elements,” and the information between them is called their “element content”, which is often text. Elements are embedded within each other. Also, elements, such as the <a> element in our example, can contain attributes, such as <a>'s “href” attribute, that contain attribute values. Elements are ordered and nested within each other – while attributes are unordered and flat. Also, we often want to combine elements from different documents, so we use “namespaces” to disambiguate tags. These tags add a URI, like “<http://www.w3.org/1999/xhtml>” for XHTML, and we declare these using the xmlns attribute in the root node's tag, like this:

```
<tag xmlns = “http://www.example.com”>
```

This makes all the following tags default to that namespace unless another one is declared explicitly by giving it a name like:

```
<tag xmlns:yans = “http://www.mypage.com/YetAnotherNameSpace”>
```

This means that every XML element that has the prefix “yans” belongs to YetAnotherNameSpace, not the current default one. Prefixes are attached via colons, like this:

```
<yans:tag> My content here is in another namespace. Wow! </yans:tag>
```

. We can then tell the XML parser that we are using an element from a different namespace by giving

by prefixing it by the namespace URI. It's good practice to always give your XML a namespace.

We can then make up our own XML vocabularies, to describe anything we want, such as our schedule or fun workshops, and easily embed valid XHTML elements:

```
<workshops xmlns = "http://www.ibiblio.org/hhalpin/examples/workshopxml/"
  xmlns:html = "www.w3.org/1999/xhtml">
  <workshop name="KnowledgeLab"
    webpage="www.knowledgelab.org/wiki/SecondKnowledgeLab >
    <startdate><month>Feb</month><day> 3</day></startdate>
    <enddate><month>Feb</month><day>5</day></enddate>
    <location>Lancaster</location>
    <comments>I heard the <html:b>last one</html:b> was good!</comment>
  </workshop>
</workshops>
```

What is interesting is that XML is everywhere – and simply put, it allows us a programming language and device independent way of storing complex, structured knowledge using metaphors from the Web many people already understand.

Indeed, one way of thinking about XML is the fact that it makes everything into a tree. You can then traverse a tree by picking out elements by their name, and finding their “parents” and “siblings.” This works better if you treat attributes and elements as all sub-categories of the same thing, the “node”. In fact, the easiest way to access XML data in a file for many people is to use the DOM, or “Document Object Model” does exactly that.

### **RSS (Really Simple Syndication!)**

*In 1995 I woke up from a sort-of Microsoft induced narcoleptic coma and realized there was this thing called “The Web,” and got really excited about it.*

Adam Bosworth, now at Google

One great thing XML has caused is the development of “microformats,” or commonly used XML vocabularies. Perhaps the most popular is RSS, which stands for either Really Simple Syndication, RDF Site Summary, or something else depending on who you talk to. However, regardless of how exactly one interprets the acronym, it's used for a pretty simple idea now: for web-pages with dynamic content, provide a common XML vocabulary that can be sent to other sites and programs updating them something has changed.

RSS feeds divide the types of news into “channels,” which generally can be one per web-site but on more complex web-sites there can be even more. Each channel has a series of “items” or updates it sends out whenever something new happens on to the website.

There are lots of information about news feeds, but in general the most important are “title”, “description”, and “link”. These provide the title of the update, a brief textual description, and a link to the URI of the new news. There's lots more optional information like “author” and “date”.

Here's one sample RSS feed:

```
<?xml version="1.0" ?>
  <rss version="2.0">
    <channel>
      <title>PoetNews</title>
      <description>New</description>
      <link>http://allgadgetsreviewed.com</link>
    </channel>
    <item>
      <title>Celebrating Pablo Neruda</title>
      <description>Is Neruda really the greatest poet of the past century?
      </description>
      <link>www.democracynow.org/article.pl?sid=04/07/16/1442233</link>
    </item>
```



```

        <item>
            <title>On Donne's Poetry</title>
            <author>Samuel Taylor Coleridge</author>
            <description>The classic review of the famous metaphysical
            poet. It's now available as an online text, or e-text.</description>
            <link>etext.lib.virginia.edu/stc/Coleridge/poems/Donne_Poetry.html</link>
        </item>
    </channel>
</rss>

```

You can collect and read RSS newsfeeds into news aggregators, so you don't have to visit each web-page to see the interesting news.

### **Reading RSS Feeds Using Minidom**

*XML is easy. It is the problems people are trying to address with XML that are hard.*

Jonathan Borden on xml-dev mailing list

Now we're going to see how you can use Python to parse these RSS feeds and display them for you in an easy to read format. This code from XML.com is complex and involves using lots of imported modules, but we'll step through it a bit.

First, we need to get the proper modules. We are going to use “urllib” to get the contents from URLs (URIs) like “<http://www.example.com>.” Then we have to deal with XML, so we are going to load the “minidom” DOM XML package, a standard part of the PyXML that lets us deal with XML in Python.

```

from xml.dom import minidom
import urllib

```

Since RSS comes in so many flavors (including one in RDF 1.0 which uses the Resource Description Framework that appears much more complicated but is more or less doing the same thing as vanilla RSS) we're going to make a tuple of the default namespaces we might be running in to. Note that lots of versions of RSS do not even have namespaces! Also, some versions of RSS (RDF-based RSS 1.0) use elements from yet another namespace, Dublin Core, so we have to add these to the tuple as well.

```

DEFAULT_NAMESPACES = \
    (None, # RSS 0.91, 0.92, 0.93, 0.94, 2.0
    'http://purl.org/rss/1.0/', # RSS 1.0
    'http://my.netscape.com/rdf/simple/0.9/' # RSS 0.90
    )
DUBLIN_CORE = ('http://purl.org/dc/elements/1.1/',)

```

Then we need to make a function that given a URI will read the resulting code and then return it. So, we define a function that takes a URI, and then return the result of using minidom's parse function over the results of opening the URI, as done by the function “urllib.urlopen()”.

```

def load(rssURL):
    return minidom.parse(urllib.urlopen(rssURL))

```

We then want to be able to find out each of the elements by name. Since we might be dealing with namespaces, we iterate through the tuples of possible namespace prefixes we created earlier when trying to get the whole name of the tag. We want to pass this function a node from miniDOM (like the root node of the RSS XML document), a tagname (like “author”), and the possible namespace tuple. Then for every possible name combination, we use minidom's “getElementByTagNameNS” feature with the namespace and tag names as arguments in order to discover if any “children” of that node (nodes directly beneath it) match that wanted tag. If there are children that fit this (i.e. the length of a list of these nodes is greater than zero, or it just exists), we return these nodes. Otherwise we return the empty nodes.

```

def getElementsByTagName(node, tagName, possibleNamespaces=DEFAULT_NAMESPACES):

```

```

for namespace in possibleNamespaces:
    children = node.getElementsByTagNameNS(namespace, tagName)
    if len(children): return children
return []

```

However, since there should not according to the RSS specification be multiple authors, titles, and so on, we can define another “helper function” that just returns the first child (i.e. “children[0]”) in most cases.

```

def first(node, tagName, possibleNamespaces=DEFAULT_NAMESPACES):
    children = getElementsByTagName(node, tagName, possibleNamespaces)
    return len(children) and children[0] or None

```

Now that we found some nodes that have content, we still have to get the text content out, and we don't want to throw out any HTML or other markup in the text as well. This clever function uses a feature in Prolog called “list comprehensions” that offer a concise way to create lists by embedding the loop that creates the members of the list in the list index itself. Therefore this simple function just “joins” together the data in each child node (as given by the “child.data” minidom member variable) that the list returns.

```

def textOf(node):
    return node and "".join([child.data for child in node.childNodes]) or ""

```

Lastly, we require the ability to run this thing. So we have a main function after our declared function. It takes the name of an RSS feed from the user's at the command prompt, like this, by virtue of using the “load(sys.argv[1])” function. It's argument, the list “argv”, is how Python communicates data from the command prompt, as in:

```
python rss.py http://www.infoshop.org/inews/backend/news.rdf
```

A second member of the list “argv[2]” would exist if the python program took two arguments, and so on. The rest of the main function doesn't do too much other than use the “textOf” function created earlier to iterate over relevant parts of the RSS feeds. The data the textOf function returns take advantage of with a built in string conversion to UTF-8 to properly use characters via “encode(“utf-8”).

```

import sys
rssDocument = load(sys.argv[1])
for item in getElementsByTagName(rssDocument, 'item'):
    print 'Title:', textOf(first(item, 'title')).encode("utf-8")
    print 'Link:', textOf(first(item, 'link')).encode("utf-8")
    print 'Description:', textOf(first(item, 'description')).encode("utf-8")
    print 'Date:', textOf(first(item, 'date', DUBLIN_CORE)).encode("utf-8")
    print 'Author:', textOf(first(item, 'creator', DUBLIN_CORE)).encode("utf-8")
print

```

There's lots to do to expand this program. You could add in user prompts, or save and load files that keep track of lists of often read RSS feeds, and allow users to add and subtract feeds by name. You could upload and run the Python program on the Web using either a CGI (Common Gateway Interface) to print off the formatting as HTML, or just have it run as a UNIX job and have the program output straight HTML instead of text. Various error-checks on the command prompt could be added, or even a graphical user interface for adding, subtracting, and reading feeds. The best way to learn to code is often not to code everything from scratch yourself, but to modify and extend free and open source code to suit your particular desires and needs. And likely other people will want to use your code as well! Enjoy Python and happy hacking!

### **Reference:**

#### Websites

Python site: <http://www.python.org>

Dive Into Python, tutorials: <http://www.diveintopython.org>

Vaults of Parnassus, Python libraries: <http://www.vex.net/parnassus>  
Python Cookbook: <http://aspn.activestate.com/ASPN/Python/Cookbook/>  
PyGame, gaming libraries: <http://www.pygame.org>  
AIMA, Artificial Intelligence examples: <http://aima.cs.berkeley.edu/python/readme.html>  
'war' simple Python game: <http://www.pythonpros.com/gstein/war/>  
XML.com: Latest news on XML <http://www.xml.com>  
World Wide Web Consortium: Crazy people that keep making XML-standards: <http://www.w3.org>

#### Books About Python and XML

Alex Martelli, David Ascher (editors), Python Cookbook, O'Reilly, 2002  
Mark Lutz, David Ascher, Learning Python, O'Reilly, 2004  
Guido van Rossum, et al., Programming Python, O'Reilly, 2001  
Mark Pilgrim, Dive Into Python, Apress, 2004  
Python and XML by Christopher Jones et. al. O'Reilly, 2002  
(Watch out – the examples in the above book are kinda outdated and have some typos. Check out the columns on XML.com by Uche Ogbuji for corrections and great tips!)  
Effective XML by Elliotte Rusty Harold, Addison Wesley

#### Books about Philosophy and Computing

Douglas Hofstadter, Godel, Escher, Bach: An Eternal Golden Braid, Basic Books 1979  
Brian Cantwell Smith, On the Origin of Objects. MIT Press, 2005  
Paul Graham, Hackers and Painters: Big Ideas from the Computer Age, O'Reilly 2004  
Andy Clark, Natural-born Cyborgs, Oxford Press 2005  
Tim Berners-Lee, Weaving the Web, 2000

## GNU Free Documentation License

Version 1.2, November 2002

Copyright (C) 2000,2001,2002 Free Software Foundation, Inc.  
51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA  
Everyone is permitted to copy and distribute verbatim copies  
of this license document, but changing it is not allowed.

### 0. PREAMBLE

The purpose of this License is to make a manual, textbook, or other functional and useful document "free" in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondly, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of "copyleft", which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

### 1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "Document", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "you". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "Modified Version" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "Secondary Section" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "Invariant Sections" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "Cover Texts" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "Transparent" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "Opaque".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The "Title Page" means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, "Title Page" means the text near the most prominent appearance of the work's title, preceding the beginning of the body of the text.

A section "Entitled XYZ" means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as "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 redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

## 4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

- \* A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.
- \* B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.
- \* C. State on the Title page the name of the publisher of the Modified Version, as the publisher.
- \* D. Preserve all the copyright notices of the Document.
- \* E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.
- \* F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.
- \* G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.
- \* H. Include an unaltered copy of this License.
- \* I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.
- \* J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.
- \* K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.
- \* L. Preserve all the Invariant Sections of the Document, unaltered in their text and in 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 retile 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 sections Entitled "Endorsements."

## 6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

## 7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an "aggregate" if the copyright resulting from the compilation is not used to limit the legal rights of the compilation's users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document's Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

## 8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled "Acknowledgements", "Dedications", or "History", the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

## 9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

## 10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See <http://www.gnu.org/copyleft/>.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered

version of this License "or any later version" applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.