Python Programming

by Wikibooks contributors

From **Wikibooks**,
the open-content textbooks collection

Contents

The current version of this Wikibook may be found at:
http://en.wikibooks.org/wiki/Python_Programming

This PDF was created on 2006-07-15
based on the 2006-07-14 version of the book.

OVERVIEW

live version • discussion • edit lesson • comment • report an error • ask a question

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 an interpreted programming language that is automatically compiled into bytecode before execution (the bytecode is then normally saved to disk, just as automatically, so that compilation need not happen again until and unless the source gets changed). It is also a dynamically typed language that includes (but does not require one to use) object oriented features and constructs.

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,
...
1 2 3 5 8 13 21 34 55 89 144
```

Another interesting aspect in Python is reflection. 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.

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.

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.

live version • discussion • edit lesson • comment • report an error • ask a question

Getting Python

In order to program in Python you need the Python software.

Installing Python in Windows

Go to http://www.python.org/download/ or the ActiveState website[1] 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 DrPython[2], eric[3], or PyScripter[4].

Installing Python in Unix

Python is standard equipment in many Unix-like operating systems; just type `which python` to check for it. If present, it may not be the latest, but it should be enough to get you started.

If it's not installed, check your operating system's web page for the proper package. Failing that, you will need to download the appropriate file from http://www.python.org/download or the ActiveState website[5].

If you decide to compile Python from source, make sure you compile in the tk extension if you want to use IDLE.

On Debian based Linux systems, you can download it by starting the command line, changing to the superuser mode using *su* - and then by typing *apt-get install python*.

Installing Python PyDEV Plug-in for Eclipse IDE

You can use the Eclipse IDE as your Python IDE. The only requirement is Eclipse and the Eclipse PyDEV Plug-in.

Go to http://download.eclipse.org/downloads and get the proper Eclipse IDE version for your OS platform. Download and install it. The install just requires you to unpack the downloaded Eclipse install file onto your system.

You can install PyDEV Plug-in two ways

- Go to http://pydev.sourceforge.net and get the latest PyDEV Plug-in version. Download it, install by unpacking into the Eclipse base folder.
- The better way is to use Eclipse's Eclipse update manager. The link address is http://pydev.sf.net/updates/ and let Eclipse do the rest. Once this is done Eclipse will always check for any updates when Eclipse searches for updates.

Python Mode for Emacs

There is also a python mode for Emacs which provides features such as running pieces of code, and changing the tab level for blocks. You can download the mode at http://sourceforge.net/projects/python-mode/

live version • discussion • edit lesson • comment • report an error • ask a question

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 2.3.4 (#2, Aug 29 2004, 02:04:10)
[GCC 3.3.4 (Debian 1:3.3.4-9)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

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

The >>> 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'>
```

However, you need to be careful in the interactive environment. If you aren't confusion may ensue. For example, the following is a valid Python script:

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

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
>>>
```

live version • discussion • edit lesson • comment • report an error • ask a question

Creating Python programs

live version • discussion • edit lesson • comment • report an error • ask a question

Python programs are nothing more than text files, and they may be edited with standard text editors.*

In Windows, notepad will be sufficient for a little while, but you will soon find that a more powerful editor, such as vim, emacs, or python's built-in IDE, IDLE makes editing much easier.

In Unix, nano or pico are respectable beginners' editors, while vim and emacs are used when more power is needed.

Additional editors exist that are Python friendly (e.g., use Python syntax highlighting).

Let's create the first program. It is listed as follows; create a file containing it with the name `hello.py` in your preferred text editor:

```
#!/usr/bin/python

print "Hello, world!"
```

In Windows

- Open your text editor.
- Type in the program.
- Create a temporary directory, such as `C:\pythonpractice`, and save the program in it, with the name `hello.py`.
- Open the MS-DOS prompt. (Or Start > Run > command > enter)
- In the MS-DOS prompt, go into the directory you just created, then run the program.

```
C:\> cd \pythonpractice
C:\pythonpractice> python hello.py
```

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

In Unix

- Make a directory for Python practice, and cd into it:

```
$ mkdir ~/pythonpractice
$ cd ~/pythonpractice
```

- Open the editor and type in the program, then save it as `hello.py`.
- Make it executable, and run it:

```
$ chmod +x hello.py
$ ./hello.py
```

Result

The program should print `Hello, world!`. Congratulations! You're well on your way to becoming a Python programmer.

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 debuging and prototyping.

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

Exercises

- Modify the `hello.py` program to say hello to a historical political leader (or to Ada Lovelace).
- Change the program so that after the greeting, it asks, "How did you get here?".

Solutions

---

* Sometimes, Python programs are distributed in compiled form. We won't have to worry about that for quite a while.

Using variables and math

Using a variable

A *variable* is something with a value that may change. In Python, variables are strongly typed, meaning that if a variable has a number, it can't be treated as a string, or vice versa. Here is a program that uses a variable:

```
#!/usr/bin/python

name = 'Ada Lovelace'
print "Goodbye, " + name + '!'
```

(Oops! I used single quotes for Ada's name, then double quotes around Goodbye. That's OK, however, because these two quotes do exactly the same thing in Python. The only thing you can't do is mix them and try to make a string like this: `"will not work'`.)

This program isn't much use, of course. But what about variables that the program truly can't guess about?

raw_input()

```
#!/usr/bin/python

print 'Please enter your name.'
name = raw_input()
print 'How are you, ' + name + '?'
```

(What's `raw_input()` doing? Evidently, it's getting *input* from *you*. See Input and output.)

Of course, with the power of Python at hand, the urge to determine one's mass in stone is nearly irresistible. A concise program can make short work of this task. Since a stone is 14 pounds, and there are about 2.2 pounds in a kilogram, the following formula should do the trick:

$$m_{stone} = \frac{m_{kg} \times 2.2}{14}$$

Simple math

```
#!/usr/bin/python

print "What is your mass in kilograms?",
mass_kg = int(raw_input())
mass_stone = mass_kg * 2.2 / 14
print "You weigh " + str(mass_stone) + " stone."
```

Run this program and get your weight in stone!

This program is starting to get a little bit cluttered. That's because, in addition to all the math, I snuck in some new features.

- When the previous program asked for your name, you were typing below the question. This time, you're typing at the end of the line that asks, `"What is your mass in kilograms?"`. What's happening here is that, normally, the `print` statement will add a **newline** to the end what you're printing. That's why the cursor went to the next line in the previous program. But in *this* program, I added a little comma to the end. That makes `print` omit the newline.
- `int()` - this handy function takes a string, and returns a number. Remember when you read that Python is strongly typed? Python won't allow us to do math on a string. Whatever you type is a string, even if it consists of digits. But `int()` will recognize a string made of digits and return a number.
- The `str(mass_stone)` in the print statement. It turns out that you can't add together strings and numbers; `"You weigh " + mass_stone` just wouldn't work. So, we have to take the number and turn it into a string. Incidentally, ` would do the same thing as the str() function, but that is deprecated.

Formatting output

In the previous program, we used this line of code to print the result:

```
print "You weigh " + str(mass_stone) + " stone."
```

There are a couple of problems with this. First, it mixes up operators and quotes, and can be a little tough to read. Second, the number won't be printed very nicely, as the following example illustrates:

```
$ ./kg2stone

What is your mass in kilograms? 65
You weigh 10.214285714285714 stone.
```

Not only is that much accuracy unjustified, it doesn't look nice. Python's % operator comes to the rescue. It allows `printf`-like formatting, in the form:

```
STRING % (arg1, arg2, ...)
```

The string contains one format code for each argument. There are several types of format codes; see the strings section for a complete list.

To improve our program, we just need the `%f` format code:

```
print "You weigh %.1f stone." % (mass_stone)
```

The `%.1f` format code causes a floating point number to be printed, with exactly one digit after the decimal. This produces much nicer output:

```
$ ./kg2stone

What is your mass in kilograms? 65
You weigh 10.2 stone.
```

Basic syntax

There are four fundamental concepts in Python.

Case Sensitivity

All variables are case-sensitive. Python treats 'number' and 'Number' as seperate, unrelated entities.

Spaces and tabs don't mix

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. However, tab are read as equal to eight spaces, so changing your tab width to 8 in your editor helps if you find yourself frequently making this mistake.

Objects

In Python, like all object oriented languages, there are aggregations of code and data called Objects, which 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__']
```

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

live version • discussion • edit lesson • comment • report an error • ask a question

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 basic datatypes are:

- Integers, equivalent to C longs
- Floating-Point numbers, equivalant to C doubles
- Long integers of non-limited length
- Complex Numbers.
- Strings
- Some others, such as type and function

Python's composite datatypes are:

- lists
- tuples
- dictionaries, also called dicts, hashmaps, or associative arrays

Literal integers can be entered as in C:

- decimal numbers can be entered directly
- octal numbers can be entered by prepending a 0 (0732 is octal 732, for example)
- hexadecimal numbers can be entered by prepending a 0x (0xff is hex FF, or 255 in decimal)

Floating point numbers can be entered directly.

Long integers are entered either directly (123456789101112131415161718192O is a long integer) or by appending an L (0L is a long integer). Computations involving short integers that overflow are automatically turned into long integers.

Complex numbers are entered by adding a real number and an imaginary one, which is entered by appending a j (i.e. 10+5j is a complex number. So is 10j). Note that j by itself does not constitute a number. If this is desired, use 1j.

Strings can be either single or triple quoted strings. The difference is in the starting and ending delimiters, and in that single quoted strings cannot span more than one line. Single quoted strings are entered by entering either a single quote (') or a double quote (") followed by its match. So therefore

```
'foo' works, and
"moo" works as well,
     but
```

```
'bar" does not work, and
"baz' does not work either.
"quux'' is right out.
```

Triple quoted strings are like single quoted strings, but can span more than one line. Their starting and ending delimiters must also match. They are entered with three consecutive single or double quotes, so

```
'''foo''' works, and
"""moo""" works as well,
     but
'"'bar'"' does not work, and
"""baz''' does not work either.
'"'quux"'" is right out.
```

Tuples are entered in parenthesis, with commas between the entries:

```
(10, 'Mary had a little lamb')
```

Also, the parenthesis can be left out when it's not ambigouous to do so:

```
 10, 'whose fleece was as white as snow'
```

Note that one-element tuples can be entered by surrounding the entry with parentheses and adding a comma like so:

```
('this is a stupid tuple',)
```

Lists are similar, but with brackets:

```
['abc', 1,2,3]
```

Dicts are created by surrounding with curly braces a list of key,value pairs separated from each other by a colon and from the other entries with commas:

```
{ 'hello': 'world', 'weight': 'African or European?' }
```

Any of these composite types can contain any other, to any depth:

```
(((((((((('bob',),['Mary', 'had', 'a', 'little', 'lamb']), { 'hello' : 'world'
 } ),),),),),),),)
```

NUMBERS

live version • discussion • edit lesson • comment • report an error • ask a question

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 equivilant 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.
- *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. It is in engineering style notation.

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.

```
>>> x = 5
>>> type(x)
<type 'int'>
>>> x = 18768765456465897097890986957645 3
>>> type(x)
<type 'long'>
>>> x = 1.34763
>>> type(x)
<type 'float'>
>>> x = 5 + 2j
>>> type(x)
<type 'complex'>
```

However, some expressions may be confusing since in the current version of python, using the / operator on two integers will return another integer, using floor division. For example, 5/2 will give you 2. You have to specify one of the operands as a float to get true division, e.g. 5/2. or 5./2 (the dot specifies you want to work with float) to have 2.5. This behavior is deprecated and will disappear in a future python release as shown from the from __future__ import.

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

live version • discussion • edit lesson • comment • report an error • ask a question

STRINGS

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'
```

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
         -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 constants

String constants can be found in the standard string module. Either single or double quotes may be used to delimit string constants.

String methods

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

- **capitalize**
- **center**
- **count**
- decode
- encode
- endswith
- **expandtabs**
- **find**
- **index**
- **isalnum**
- **isalpha**
- **isdigit**
- **islower**
- **isspace**
- **istitle**
- **isupper**
- **join**
- **ljust**
- **lower**
- **lstrip**
- **replace**
- **rfind**
- **rindex**
- **rjust**
- **rstrip**
- **split**
- **splitlines**
- startswith
- **strip**
- **swapcase**
- **title**
- translate
- **upper**
- zfill

Only emphasized items will be covered.

is*

isalnum(), isalpha(), isdigit(), islower(), isupper(), isspace(), and istitle() fit into this category.

- **isalnum** returns True if the string is entirely composed of alphabetic or numeric characters (i.e. no punctuation).
- **isalpha** and **isdigit** work similarly for alphabetic characters or numeric characters only.
- **islower**, **isupper**, and **istitle** return True if the string is in lowercase, uppercase, or titlecase respectively (titlecase, means the first character of each word is uppercase and the rest are lowercase).
- **isspace** returns True if the string is composed entirely of whitespace.

title, upper, lower, swapcase, capitalize

Returns the string converted to title case, upper case, lower case, inverts case, or capitalizes, respectively.

The **title** method capitalizes the first letter of each word in the string (and makes the rest lower case). Words are identified as substrings of alphabetic characters that are separated by non-alphabetic characters. This can lead to some unexpected behavior. For example, the string "x1x" will be converted to "X1X" instead of "X1x".

The **swapcase** method makes all uppercase letters lowercase and vice versa.

The **capitalize** method is like title except that it considers the entire string to be a word. (i.e. it makes the first character upper case and the rest lower case)

Example:

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

count

Returns the number of the specified substrings in the string. i.e.

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

strip, rstrip, lstrip

Returns a copy of the string with the leading (lstrip) and trailing (rstrip) whitespace removed. strip removes both.

```
>>> s = '\t Hello, world\n\t '
>>> print s
         Hello, world

>>> print s.strip()
Hello, world
>>> print s.lstrip()
Hello, world
        # ends here
>>> print s.rstrip()
         Hello, world
```

Note the leading and trailing tabs and newlines.

Strip methods can also be used to remove other types of characters.

```
import string
s = 'www.wikibooks.org'
print s
print s.strip('w')                  # Removes all w's from outside
print s.strip(string.lowercase)     # Removes all lowercase letters from outside
print s.strip(string.printable)     # Removes all printable characters
```

Outputs:

```
www.wikibooks.org
.wikibooks.org
.wikibooks.
```

Note that string.lowercase and string.printable require an import string statement

ljust, rjust, center

left, right or center justifies a string into a given field size (the rest is padded with spaces).

```
>>> s = 'foo'
>>> s
'foo'
>>> s.ljust(7)
'foo    '
>>> s.rjust(7)
'    foo'
>>> s.center(7)
'  foo  '
```

join

Joins together the given sequence with the string as separator:

```
>>> seq = ['1', '2', '3', '4', '5']
>>> ' '.join(seq)
'1 2 3 4 5'
```

```
>>> '+'.join(seq)
'1+2+3+4+5'
```

map may be helpful here: (it converts numbers in seq into strings)

```
>>> seq = [1,2,3,4,5]
>>> ' '.join(map(str, seq))
'1 2 3 4 5'
```

now arbitrary objects may be in seq instead of just strings.

find, index, rfind, rindex

The find and index functions returns the index of the first found occurrence of the given subsequence. If it is not found, find returns -1 but index raises a ValueError. rfind and rindex are the same as find and index except that they search through the string from right to left (i.e. they find the last occurance)

```
>>> s = 'Hello, world'
>>> s.find('l')
2
>>> s[s.index('l'):]
'llo, world'
>>> s.rfind('l')
10
>>> s[:s.rindex('l')]
'Hello, wor'
>>> s[s.index('l'):s.rindex('l')]
'llo, wor'
```

Because Python strings accept negative subscripts, index is probably better used in situations like the one shown because using find instead would yield an incorrect value.

replace

Replace works just like it sounds. It returns a copy of the string with all occurrences of the first parameter replaced with the second parameter.

```
>>> 'Hello, world'.replace('o', 'X')
'HellX, wXrld'
```

Or, using variable assignment:

```
str = 'Hello, world'
newStr = str.replace('o', 'X')
print str
print newStr
```

Outputs:

```
'Hello, world'
'HellX, wXrld'
```

Notice, the original variable (`str`) remains unchanged after the call to `replace`.

expandtabs

Replaces tabs with the apropriate number of spaces. (default number of spaces per tab = 8; this can be changed by passing the tab size as an argument)

```
s = 'abcdefg\tabc\ta'
print s
print len(s)
t = s.expandtabs()
print t
print len(t)

abcdefg abc     a
13
abcdefg abc     a
17
```

Notice how (although these both look the same) the second string (t) has a different length because each tab is represented by spaces not tab characters.

To use a a tab size of 4 instead of 8:

```
v = s.expandtabs(4)
print v
print len(s)
```

Outputs:

```
abcdefg abc a
13
```

split, splitlines

The **split** method returns a list of the words in the string. It can take a separator argument to use instead of whitespace.

```
 >>> s = 'Hello, world'
 >>> s.split()
 ['Hello, ', 'world']
 >>> s.split('l')
 ['He', '', 'o, wor', 'd']
```

Note that in neither case is the separator included in the split strings, but empty strings are allowed.

The **splitlines** method breaks a multiline string into many single line strings. It is analogous to split('\n') (but accepts '\r' and '\r\n' as delimiters as well) except that if the string ends in a newline character, **splitlines** ignores that final character (see example).

```
>>> s = """
... One line
... Two lines
... Red lines
... Blue lines
... Green lines
... """
>>> s.split('\n')
```

```
['', 'One line', 'Two lines', 'Red lines', 'Blue lines', 'Green lines', '']
>>> s.splitlines()
['', 'One line', 'Two lines', 'Red lines', 'Blue lines', 'Green lines']
```

live version • discussion • edit lesson • comment • report an error • ask a question

Lists

About lists in Python

A list in Python is an ordered group of items. It is a very general structure. You can have any kind of things in a lists, and they don't have to be all of the same type. For instance, you could put numbers, letters, strings and donkeys all on the same list.

If you are using a modern version of Python (and you should be), there is a class called 'list'. If you wish, you can make your own subclass of it, and determine list behaviour which is different than the default standard. But first, you should be familiar with the current behaviour of lists.

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]]
```

Operations on lists

List Attributes

Length:

```
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]
```

Another way to combine lists is with **extend**. If you need to combine lists inside of a lamda, **extend** is the way to go.

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

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']
```

Comparing lists

Lists can be compared for equality.

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

Sorting lists

Sorting lists is easy with a sort method.

```
>>> 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}}

TUPLES

About tuples in Python

A tuple in Python is much like a list except that it is immutable (unchangable) once created. They are generally used for data which should not be edited.

Tuple notation

Tuples may be created directly or converted from lists. Generally, tuples are enclosed in parenthesis.

```
>>> l = [1, 'a', [6, 3.14]]
>>> t = (1, 'a', [6, 3.14])
>>> t
(1, 'a', [6, 3.1400000000000001])
>>> tuple(l)
(1, 'a', [6, 3.1400000000000001])
>>> t == tuple(l)
True
>>> t == l
False
```

Also, tuples will be created from items separated by commas.

```
 >>> t = 'A', 'tuple', 'needs', 'no', 'parens'
 >>> t
 ('A', 'tuple', 'needs', 'no', 'parens')
```

A one item tuple is created by a item in parens followed by a comma:

```
 >>> t = ('A single item tuple',)
 >>> t
 ('A single item tuple',)
```

Packing and Unpacking

You can also perform multiple assignment using tuples.

```
>>> article, noun, verb, adjective, direct_object = t
>>> noun
'tuple'
```

Note that either, or both sides of an assignment operator can consist of tuples.

```
>>> a, b = 1, 2
>>> b
2
```

Assigning a tuple to a several different variables is called “tuple unpacking,” while assigning multiple values to a tuple in one variable is called “tuple packing.” When unpacking a tuple, or performing multiple assignment, you must have the same number of variables being assigned to as values being assigned.

Operations on tuples

These are the same as for lists except that we may not assign to indices or slices, and there is no "append" operator.

```
>>> a = (1, 2)
>>> b = (3, 4)
>>> a + b
(1, 2, 3, 4)
>>> a
(1, 2)
>>> b
(3, 4)
>>> print a.append(3)
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: 'tuple' object has no attribute 'append'
>>> a
(1, 2)
>>> a[0] = 0
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
TypeError: object does not support item assignment
>>> a
(1, 2)
```

For lists we would have had:

```
>>> a = [1, 2]
>>> b = [3, 4]
>>> a + b
[1, 2, 3, 4]
>>> a
[1, 2]
>>> b
[3, 4]
>>> a.append(3)
>>> a
[1, 2, 3]
>>> a[0] = 0
>>> a
[0, 2, 3]
```

Tuple Attributes

Length: Finding the length of a tuple is the same as with lists; use the built in len() method.

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

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]
```

Sets

Python also has a 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.

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])
```

Reference

Python Library Reference on Set Types

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
```

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

Dividing two integers uses integer division, also known as floor division. Using division this way is deprecated because it is intended to change in the future. Instead, if you want floor division, use '//'.

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().

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

>>> x = 'repeat this  '
>>> x  # repeat this
repeat this
>>> x *= 3  # fill with x repeated three times
>>> x
repeat this  repeat this  repeat this
```

Boolean

or: if a or b:

```
do_this
```

else:

```
do_this
```

and: if a and b:

```
do_this
```

else:

```
do_this
```

Flow control

live version • discussion • edit lesson • comment • report an error • ask a question

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 might arguably be considered an advanced form of program flow control, but they are not covered here.

Loops

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

For loops

A for loop iterates over elements of a sequence (tuple or list). A variable is created to represent the object in the sequence. For example,

```
l = [1,2,3,4,5]
for i in l:
    print i
```

This will output

```
1
2
3
4
5
```

The `for` loop loops over each of the elements of a list or iterator, assigning the current element to the variable name given. In the first example above, each of the elements in `l` is assigned to `i`.

A builtin function called range exists to make creating sequential lists such as the one above easier. The loop above is equivalent to either:

```
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,
```

This will output

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

or

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

for loops can have names for each element of a 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
```

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 = x - 1
```

will output

```
5
4
3
2
1
```

Python's while loops can also have an 'else' clause, which is a block of statements that is executed (once) when the statement starts out false. For example:

```
x = 5
y = x
while y > 0:
    print y
    y = y - 1
else:
    print x
```

this will output

```
5
4
3
2
1
5
```

Unlike some languages, there is no postcondition loop.

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 "100 not found in range"
```

Branches

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'
```

Conclusion

Any of these loops, branches, and function calls can be nested in any way desired. A loop can loop over a loop, a branch can branch again, and a function can call other functions, or even call itself.

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.

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("messsage", 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]) for i, key in enumerate(entries.keys()) if
i < max_length])
```

If 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)
```

Lambda Forms

Besides assigning the return value of a function to a variable, we can also create variables that contain functions. Python provides the “lambda” keyword for defining unnamed functions which can be assigned to variables. You place the arguments before the colon, and the return value of the lambda after. If this is assigned to a variable, you can then use that variable as if it were a function with the same parameters and return value as the lambda.

```
>>> square = lambda x: x*x
>>> square(3)
9
```

You can also use variables other than the parameters in the lambda. However, note that the lambda function will also use the values of variables from the scope in which it was created, rather than the scope in which it is run

```
>>> prefix = "Note: "
>>> def return_lambda(prefix):
...     return lambda note: prefix + note
...
>>> prefix = "re: "
>>> f = return_lambda("Attn: ")
>>> f("Carnivorous octopi")
'Attn: Carnivorous octopi'
```

Note that all functions in python can be stored to variables, and are in fact simply variables themselves.

```
>>> make_note = return_lambda
>>> make_note("See: ")("lambda calculus")
'See: lambda calculus'
```

SCOPING

Variables

Variables in Python are automatically declared by assignment. Variables are always references to objects, and are never typed. Like all Algol-derivative languages, 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
```

Here, to correct this problem, one must add y = [] before the for loop.

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:
   theFile = 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 "Congratulations! You have managed to trip a #" + str(ErrorNumber)
+ " error:"
       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):
      def __init__(self,value):
          self.parameter=value
      def __str__(self):
          return repr(self.parameter)
```

And then using that exception:

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

Builtin exception classes

All built-in Python exceptions

Exotic uses of exceptions

Exceptions are good for more than just error handling. If you have a complicated piece of code to choose which of several courses of action to take, it can be useful to use exceptions to jump out of the code as soon as the decision can be made. The Python-based mailing list software Mailman does this in deciding how a message should be handled. Using exceptions like this may seem like it's a sort of GOTO -- and indeed it is, but a limited one called an *escape continuation*. Continuations are a powerful functional-programming tool and it can be useful to learn them.

Just as a simple example of how exceptions make programming easier, say you want to add items to a list but you don't want to write clanky if statements to initialize the list; you can do:

```
for newItem in newItems:
    try:
        self.items.append(newItem)
    except AttributeError:
        self.items = [newItem]
```

This is also much more efficient then an if statement because it assumes the code will succeed. In fact it will work 99% of the time :) An if statement would continue to get executed even after the array has been initialized.

Input and output

Input

Python has two functions designed for accepting data directly from the user:

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

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

raw_input()

raw_input() asks the user for a string of data (ended with a new line), 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 inputted data here>
```

input()

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 arbitary 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.

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,
```

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'
```

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,
```

will output:

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

In order to end this line, it may be necessary to add a print statement without any objects.

```
for i in range(10):
    print i,
print
for i in range(10,20):
    print i,
```

will output:

```
0 1 2 3 4 5 6 7 8 9
10 11 12 13 14 15 16 17 18 19
```

If the bare print statement were not present, the above output would look like:

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
```

If it is not desirable to add spaces between objects, it is necessary to output only one string, by concatenating the string representations of each object:

```
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)>
```

If you want to avoid printing the trailing newline or space (space when you use comma at the end), 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
```

It is also possible to use similar syntax when writing to a file, instead of to standard output, like so:

```
print >> f, 'Hello, world'
```

This will print to any object that implements write(), which includes file objects.

MODULES

live version • discussion • edit lesson • comment • report an error • ask a question

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.

live version • discussion • edit lesson • comment • report an error • ask a question

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 method.

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 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 a 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
```

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')
print bar
```

outputs

```
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'
```

__getattr___

Similar to __setattr__, except this function is called when we try to access a 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
```

Operator Overloading

Operator overloading allows us to use the built-in Python syntax and operators to call functions which we define.

Binary Operators

Binary Operator Override Functions

If a class has the __add__ function, we can use the '+' operator to add instances of the class. This will call __add__ with the two instances of the class passed as parameters, and the return value will be the result of the addition.

```
>>> class FakeNumber:
...     n = 5
...     def __add__(A,B):
...         return A.n + B.n
...
>>> c = FakeNumber()
>>> d = FakeNumber()
>>> d.n = 7
>>> c + d
12
```

To override the augmented assignment operators, merely add 'i' in front of the normal binary operator, i.e. for '+=' use '__iadd__' instead of '__add__'. The function will be given one argument, which will be the object on the right side of the augmented assignment operator. The returned value of the function will then be assigned to the object on the left of the operator.

```
>>> c.__imul__ = lambda B: B.n - 6
>>> c *= d
>>> c
```

```
1
```

It is important to note that the augmented assignment operators will also use the normal operator functions if the augmented operator function hasn't been set directly. This will work as expected, with "__add__" being called for "+=" and so on.

```
>>> c = FakeNumber()
>>> c += d
>>> c
12
```

Function	Operator
__add__	A + B
__sub__	A - B
__mul__	A * B
__div__	A / B
__floordiv__	A // B
__mod__	A % B
__pow__	A ** B
__and__	A & B
__or__	A | B
__xor__	A ^ B
__eq__	A == B
__ne__	A != B
__gt__	A > B
__lt__	A < B
__ge__	A >= B
__le__	A <= B
__lshift__	A << B
__rshift__	A >> B
__contains__	A in B A not in B

Unary Operators

Unary Operator Override Functions

Unary operators will be passed simply the instance of the class that they are called on.

```
>>> FakeNumber.__neg__ = lambda A : A.n + 6
>>> -d
13
```

Function	Operator
__pos__	+A
__neg__	-A
__inv__	~A
__abs__	abs(A)
__len__	len(A)

Item Operators

Item Operator

It is also possible in Python to override the indexing and slicing operators. This allows us to use the class[i] and class[a:b] syntax on our

own objects.

Override Functions

Function	Operator
__getitem__	C[i]
__setitem__	C[i] = v
__delitem__	del C[i]
__getslice__	C[s:e]
__setslice__	C[s:e] = v
__delslice__	del C[s:e]

The simplest form of item operator is __getitem__. This takes as a parameter the instance of the class, then the value of the index.

```
>>> class FakeList:
...     def __getitem__(self,index):
...         return index * 2
...
>>> f = FakeList()
>>> f['a']
'aa'
```

We can also define a function for the syntax associated with assigning a value to an item. The parameters for this function include the value being assigned, in addition to the parameters from __getitem__

```
>>> class FakeList:
...     def __setitem__(self,index,value):
...         self.string = index + " is now " + value
...
>>> f = FakeList()
>>> f['a'] = 'gone'
>>> f.string
'a is now gone'
```

We can do the same thing with slices. Once again, each syntax has a different parameter list associated with it.

```
>>> class FakeList:
...     def __getslice___(self,start,end):
...         return str(start) + " to " + str(end)
...
>>> f = FakeList()
>>> f[1:4]
'1 to 4'
```

Keep in mind that one or both of the start and end parameters can be blank in slice syntax. Here, Python has default value for both the start and the end, as show below.

```
>> f[:]
'0 to 2147483647'
```

Note that the default value for the end of the slice shown here is simply the largest possible signed integer on a 32-bit system, and may vary depending on your system and C compiler.

- __setslice__ has the parameters (self,start,end,value)

We also have operators for deleting items and slices.

- __delitem__ has the parameters (self,index)
- __delslice__ has the parameters (self,start,end)

Note that these are the same as __getitem__ and __getslice__.

Programming Practices

The flexibility of python classes means that classes can adopt a very 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 use 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 and 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 Rossom.

MetaClasses

In python, classes are themselves objects. Just as other objects are instances of a particular class, classes themselves are instances of a metaclass.

Class Factories

The simplest use of python metaclasses is a class factory. This concept makes use of the fact that class definitions in python are first-class objects. Such a function can create or modify a class definition, using the same syntax one would normally use in declaring a class definition. Once again, it is useful to use the model of classes as dictionaries. First, let's look a basic class factory:

```
>>> def StringContainer():
...     # define a class
...     class String:
...             content_string = ""
...             def len(self):
...                     return len(self.content_string)
...     # return the class definition
...     return String
...
>>> # create the class definition
... container_class = StringContainer()
>>>
>>> # create an instance of the class
... wrapped_string = container_class()
>>>
>>> # take it for a test drive
... wrapped_string.content_string = 'emu emissary'
>>> wrapped_string.len()
12
```

Of course, just like any other data in python, class definitions can also be modified. Any modifications to attributes in a class definition will be seen in any instances of that definition, so long as that instance hasn't overriden the attribute that you're modifying.

```
>>> def DeAbbreviate(sequence_container):
...     setattr(sequence_container, 'length', sequence_container.len)
...     delattr(sequence_container, 'len')
...
>>> DeAbbreviate(container_class)
>>> wrapped_string.length()
12
>>> wrapped_string.len()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
AttributeError: String instance has no attribute 'len'
```

You can also delete class definitions, but that will not affect instances of the class.

```
>>> del container_class
>>> wrapped_string2 = container_class()
Traceback (most recent call last):
  File "<stdin>", line 1, in ?
```

```
NameError: name 'container_class' is not defined
>>> wrapped_string.length()
12
```

The type Metaclass

The metaclass for all standard python types is the "type" object.

```
>>> type(object)
<type 'type'>
>>> type(int)
<type 'type'>
>>> type(list)
<type 'type'>
```

Just like list, int and object, "type" is itself a normal python object, and is itself an instance of a class. In this case, it is in fact an instance of itself.

```
>>> type(type)
<type 'type'>
```

It can be instantiated to create new class objects similarly to the class factory example above by passing the name of the new class, the base classes to inherit from, and a dictionary defining the namespace to use.

For instance, the code:

```
>>> class MyClass(BaseClass):
...     attribute = 42
```

Could also be written as:

```
>>> MyClass = type("MyClass", (BaseClass,), {'attribute' : 42})
```

Metaclasses

It is possible to create a class with a different metaclass than type by setting its __metaclass__ attribute when defining. When this is done, the class, and its subclass will be created using your custom metaclass. For example

```
class CustomMetaclass(type):
    def __init__(cls, name, bases, dct):
        print "Creating class %s using CustomMetaclass" % name
        super(CustomMetaclass, cls).__init__(name, bases, dct)

class BaseClass(object):
    __metaclass__ = CustomMetaclass

class Subclass1(BaseClass):
    pass
```

This will print

```
Creating class BaseClass using CustomMetaclass
Creating class Subclass1 using CustomMetaclass
```

By creating a custom metaclass in this way, it is possible to change how the class is constructed. This allows you to add or remove attributes and methods, register creation of classes and subclasses creation and various other manipluations when the class is created.

Aspect Oriented Programming

Wikipedia article on Aspect Oriented Programming [6]

More resources

Unifying types and classes in Python 2.2 [7] O'Reilly Article on Python Metaclasses [8]

live version • discussion • edit lesson • comment • report an error • ask a question

Regular Expression

live version • discussion • edit lesson • comment • report an error • ask a question

Metacharacters

```
. ^ $ * + ? { [ ] \ | ( )
```

Sets of characters

\d

Matches any decimal digit; this is equivalent to the class [0-9].

\D

Matches any non-digit character; this is equivalent to the class [^0-9].

\s

Matches any whitespace character; this is equivalent to the class [\t\n\r\f\v].

\S

Matches any non-whitespace character; this is equivalent to the class [^ \t\n\r\f\v].

\w

Matches any alphanumeric character; this is equivalent to the class [a-zA-Z0-9_].

\W

Matches any non-alphanumeric character; this is equivalent to the class [^a-zA-Z0-9_].

live version • discussion • edit lesson • comment • report an error • ask a question

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.

To create a very simple Tkinter window frame one only needs the following lines of code:

```
import Tkinter

root = Tkinter.Tk()
root.mainloop()
```

gfd From an object-oriented perspective one can do the following:

```
import Tkinter

class App:
   def __init__(self, master):
      button = Tkinter.Button(master, text="I'm a Button.")
      button.pack()

if __name__ == '__main__':
   root = Tkinter.Tk()
   app = App(root)
   root.mainloop()
```

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. When combined with PyORBit and gnome-python, it can be used to write full featured Gnome applications.

Home Page

PyQt

Bindings for the popular Unix/Linux and Windows toolkit. PyKDE can be used to write KDE-based applications.

PyQt

wxPython

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

- wxPython

Other Toolkits

- PyKDE - Part of the kdebindings package, it provides a python wrapper for the KDE libraries.

Game Programming in Python

3D Game Programming

Base techniques

- Sockets

Because 3D programms written fully in python are slower than programms written fully in C++, an often used technique is to use a combination of C++ and Python code together. One layer of 3D graphics is implemented in C++ (for example it can be one standard open source engine), which communicate with Python code client through TCP sockets. In this case, all that a developer needs to do in C++, is to create a server, that can communicate with a client and control the 3D scene drawing. A client on the other hand, has control only over other elements.

3D Game Engine with a Python binding

- Irrlicht Engine[9]
- Ogre Engine [10]

Both are very good free open source C++ 3D game Engine with a Python binding. However the Python binding is an afterthought so most often late versus the C++ engine when usable at all. Python bindings are very unefficient and limited.

3D Game Engines written for Python

Engines designed for Python from scratch.

- Blender is a 3d game engine that uses python to make 3d games
- Soya is a 3d game engine with an easy to understand design. It's written in w:Pyrex programming language and uses Cal3d for animation and ODE for physics. Soya is available under the GNU GPL license.
- Panda3D is a 3D game engine. It's a library written in C++ with Python bindings. Panda3D is designed in order to support a short learning curve and rapid develpement. This software is available for free donwload with source code under Panda3D Public License v2.0. The development was started by [Disney]. Now it exists a lot of project made with Panda3D like ToonTown, Building Virtual World, Schell Games and many others. Panda3D support a lot of features: Procedural Geometry, Animated Texture, Render to texture, Track motion, fog, particle system, and many others.

2D Game Programming

- Pygame is a cross platform Python library which wraps SDL. It provides many features like Sprite groups and sound/image loading and easy changing of an objects position. It also provides the programmer access to key and mouse events.

SOCKETS

live version • discussion • edit lesson • comment • report an error • ask a question

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
```

live version • discussion • edit lesson • comment • report an error • ask a question

FILES

Read lines from file:

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

Determine whether path exists:

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

Database Programming

External links

- SQLAlchemy
- SQLObject
- PEP 249 - Python Database API Specification v2.0
- Database Topic Guide on python.org

THREADING

A minimal example

```
#!/usr/bin/env python
import threading
import time

class MyThread(threading.Thread):
    def run(self):
        print "%s started!" % self.getName()
        time.sleep(1)
        print "%s finished!" % self.getName()

if __name__ == '__main__':
    for x in range(4):
        mythread = MyThread(name = "MyThread %s" % x)
        MyThread().start()
        time.sleep(.2)
```

This shoud output:

```
Thread-1 started!
Thread-2 started!
Thread-3 started!
Thread-4 started!
Thread-1 finished!
Thread-2 finished!
Thread-3 finished!
Thread-4 finished!
```

A minimal example with function call

Make a thread that prints numbers from 1-10, waits for 1 sec between:

```
import thread, time

def loop1_10():
    for i in range(1,10):
        time.sleep(1); print i

thread.start_new_thread(loop1_10, ())
```

Extending with C

This gives a minimal Example on how to Extend Python with C. Linux is used for building (feel free to exten it for other Platforms). If you have any problems, pleas report them (e.g. on the dicussion page), I will check back in a while and try to sort them out.

Using the Python/C API

- http://docs.python.org/ext/ext.html
- http://docs.python.org/api/api.html

A minimal example

The minimal example we will create now is very similar in behaviour to the following python snipet.

```
def say_hello(name):
    "Greet somebody."
    print "Hello %s!" % name
```

The C source code (hellomodule.c)

```
#include <Python.h>

static PyObject* say_hello(PyObject* self, PyObject* args)
{
    const char* name;

    if (!PyArg_ParseTuple(args, "s", &name))
        return NULL;

    printf("Hello %s!\n", name);

    Py_RETURN_NONE;
}

static PyMethodDef HelloMethods[] =
{
     {"say_hello", say_hello, METH_VARARGS, "Greet somebody."},
     {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC

inithello(void)
{
     (void) Py_InitModule("hello", HelloMethods);
}
```

Building the extension module on Linux

To build our extension module we create the file setup.py like:

```
from distutils.core import setup, Extension

module1 = Extension('hello', sources = ['hellomodule.c'])

setup (name = 'PackageName',
        version = '1.0',
        description = 'This is a demo package',
        ext_modules = [module1])
```

Now we can build our module with

```
python setup.py build
```

The module `hello.so` will end up in e.g `build/lib.linux-i686-2.4`.

Building the extension module on Windows

With VC8 distutils is broken. We will use cl.exe from a command prompt instead:

```
cl /LD hellomodule.c /Ic:\Python24\include c:\Python24\libs\python24.lib
/link/out:hello.dll
```

Using the extension module

Change to the subdirectory where the file `hello.so` resists. In an interactive python session you can use the module as follows.

```
>>> import hello
>>> hello.say_hello("World")
Hello World!
```

A module for calculating fibonacci numbers

The C source code (fibmodule.c)

```
#include <Python.h>

int _fib(int n)
{
    if (n < 2)
        return n;
    else
        return _fib(n-1) + _fib(n-2);
}

static PyObject* fib(PyObject* self, PyObject* args)
{
    const char *command;
    int n;

    if (!PyArg_ParseTuple(args, "i", &n))
        return NULL;

    return Py_BuildValue("i", _fib(n));
}

static PyMethodDef FibMethods[] = {
    {"fib", fib, METH_VARARGS, "Calculate the Fibonacci numbers."},
```

```
    {NULL, NULL, 0, NULL}
};

PyMODINIT_FUNC
initfib(void)
{
    (void) Py_InitModule("fib", FibMethods);
}
```

The build script (setup.py)

```
from distutils.core import setup, Extension

module1 = Extension('fib', sources = ['fibmodule.c'])

setup (name = 'PackageName',
        version = '1.0',
        description = 'This is a demo package',
        ext_modules = [module1])
```

How to use it?

```
>>> import fib
>>> fib.fib(10)
55
```

Using SWIG

Creating the previous example using SWIG is much more straight forward. To follow this path you need to get SWIG up and running first. After that create two files.

```
/*hellomodule.c*/

#include <stdio.h>

void say_hello(const char* name) {
    printf("Hello %s!\n", name);
}


/*hello.i*/

%module hello
extern void say_hello(const char* name);
```

Now comes the more difficult part, gluing it all together.

First we need to let SWIG do its work.

```
swig -python hello.i
```

This gives us the files `hello.py` and `hello_wrap.c`.

The next step is compiling (subtitute /usr/include/python2.4/ with the correct path for your setup!).

```
gcc -fpic -c hellomodule.c hello_wrap.c -I/usr/include/python2.4/
```

Now linking and we are done:)

```
gcc -shared hellomodule.o hello_wrap.o -o _hello.so
```

The module is used in the following way.

```
>>> import hello
>>> hello.say_hello("World")
Hello World!
```

Extending with C++

Boost.Python is the de facto standard for writing C++ extension modules. Boost.Python comes bundled with the Boost C++ Libraries.

The C++ source code (hellomodule.cpp)

```
#include <iostream>

using namespace std;

void say_hello(const char* name) {
    cout << "Hello " <<  name << "!\n";
}

#include <boost/python/module.hpp>
#include <boost/python/def.hpp>
using namespace boost::python;

BOOST_PYTHON_MODULE(hello)
{
        def("say_hello", say_hello);
}
```

setup.py

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

from distutils.core import setup
from distutils.extension import Extension

setup(name="blah",
      ext_modules=[
                Extension("hello", ["hellomodule.cpp"],
                libraries = ["boost_python"])
      ])
```

Now we can build our module with

```
python setup.py build
```

The module `hello.so` will end up in e.g `build/lib.linux-i686-2.4`.

Using the extension module

Change to the subdirectory where the file `hello.so` resists. In an interactive python session you can use the module as follows.

```
>>> import hello
>>> hello.say_hello("World")
Hello World!
```

STATEMENTS

live version • discussion • edit lesson • comment • report an error • ask a question

Most programming languages have the concept of a statement. A ***statement*** is a command that the programmer gives to the computer. For example:

```
print "Hi there!"
```

This command has a verb ("print") and other details (what to print). In this case, the command `"print"` means "show on the screen," not "print on the printer." The programmer either gives the statement directly to the computer (by typing it while running a special program), or creates a text file with the command in it. You could create a file called "hi.txt", put the above command in it, and give the file to the computer.

If you have more than one command in the file, each will be performed in order, top to bottom. So the file could contain:

```
print "Hi there!"
print "Strange things are afoot..."
```

The computer will perform each of these commands sequentially. It's invaluable to be able to "play computer" when programming. Ask yourself, "If I were the computer, what would I do with these statements?" If you're not sure what the answer is, then you are very likely to write incorrect code. Stop and check the manual for the programming language you're using.

In the above case, the computer will look at the first statement, determine that it's a `print` statement, look at what needs to be printed, and display that text on the computer screen. It'll look like this:

```
Hi there!
```

Note that the quotation marks aren't there. Their purpose in the program is to tell the computer where the text begins and ends, just like in English prose. The computer will then continue to the next statement, perform its command, and the screen will look like this:

```
Hi there!
Strange things are afoot...
```

When the computer gets to the end of the text file, it stops. There are many different kinds of statements, depending on which programming language is being used. For example, there could be a `beep` statement that causes the computer to output a beep on its speaker, or a `window` statement that causes a new window to pop up.

Also, the way statements are written will vary depending on the programming language. These differences are fairly superficial. The set of rules like the first two is called a programming language's ***syntax***. The set of verbs is called its ***library***.

live version • discussion • edit lesson • comment • report an error • ask a question

External links

- Python books available for free download
- Non-programmers python tutorial donated to this project. Wiki version
- Dive into Python
- How to think Like a Computer Scientist: Learning with Python
- A Byte of Python
- ActiveState Python Cookbook
- Text Processing in Python
- Dev Shed's Python Tutorials
- MakeBot - Simple Python IDE designed for teaching game programming to kids.
- SPE - Stani's Python Editor
- python tutorials
- Python Power Page - It is a page with some of the best resources you need while programing in Python.
- Python IDEs - When coding, half of the work may be done by your IDE ... so choosing a good one might be helpful

Authors

- Artevelde (Contributions)
- Thunderbolt16 (Contributions)
- Flarelocke (Contributions)
- Yath (Contributions)
- Remote (Contributions)
- BobGibson (Contributions)
- LDiracDelta (Contributions)
- 220.101.55.68 (Contributions)
- 131.215.166.102 (Contributions)
- 83.171.176.252 (Contributions)
- 71.236.64.126 (Contributions)

GNU Free Documentation License

Version 1.2, November 2002

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. Secondarily, 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 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 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.

External links

- GNU Free Documentation License (Wikipedia article on the license)
- Official GNU FDL webpage