Extracted from:

Practical Programming

An Introduction to Computer Science Using Python

The Pragmatic Programmers

Practical Programming

An Introduction to Computer Science Using Python

Jennifer Campbell
Paul Gries
Jason Montojo
Greg Wilson

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at

http://www.pragprog.com

Printed in the United States of America.

ISBN-10: 1-934356-27-1

ISBN-13: 978-1-934356-27-2

Printed on acid-free paper.

P1.0 printing, April 2009

Version: 2009-5-9

Chapter 4

Modules

Mathematicians don't prove every theorem from scratch. Instead, they build their proofs on the truths their predecessors have already established. In the same way, it's vanishingly rare for someone to write all of a program herself; it's much more common—and productive—to make use of the millions of lines of code that other programmers have written before.

A *module* is a collection of functions that are grouped together in a single file. Functions in a module are usually related to each other in some way; for example, the math module contains mathematical functions such as cos (cosine) and sqrt (square root). This chapter shows you how to use some of the hundreds of modules that come with Python and how to create new modules of your own. You will also see how you can use Python to explore and view images.

4.1 Importing Modules

When you want to refer to someone else's work in a scientific paper, you have to cite it in your bibliography. When you want to use a function from a module, you have to *import* it. To tell Python that you want to use functions in the math module, for example, you use this import statement:

Download modules/import.cmd

```
>>> import math
```

Once you have imported a module, you can use the built-in help function to see what it contains:[1]

Download modules/help_math.cmd

```
>>> help(math)
Help on built-in module math:

NAME
    math

FILE
    (built-in)

DESCRIPTION
    This module is always available.  It provides access to the
    mathematical functions defined by the C standard.

FUNCTIONS
    acos(...)
        acos(x)

        Return the arc cosine (measured in radians) of x.

    asin(...)
        asin(x)

        Return the arc sine (measured in radians) of x.
...
```

Great—our program can now use all the standard mathematical functions. When we try to calculate a square root, though, we get an error telling us that Python is still unable to find the function sqrt:

Download modules/sqrt.cmd

```
>>> sqrt(9)
Traceback (most recent call last):
  File "<string>", line 1, in <string>
NameError: name 'sqrt' is not defined
```

The solution is to tell Python explicitly to look for the function in the math module by combining the module's name with the function's name using a dot:

Download modules/sqrt2.cmd

```
>>> math.sqrt(9)
3.0
```

1. When you do this interactively, Python displays only a screenful of information at a time. Press the spacebar when you see the "More" prompt to go to the next page.

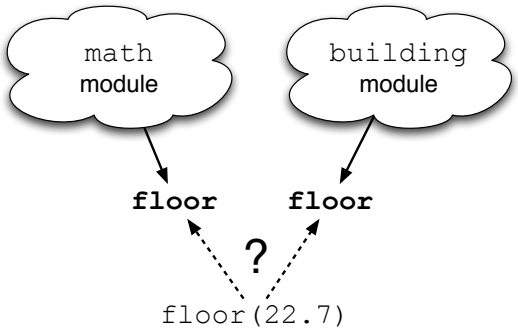

Figure 4.1: How import works

The reason we have to join the function's name with the module's name is that several modules might contain functions with the same name. For example, does the following call to floor refer to the function from the math module that rounds a number down or the function from the (completely fictional) building module that calculates a price given an area (see Figure 4.1)?

Download modules/import_ambiguity.cmd

```
>>> import math
>>> import building
>>> floor(22.7)
```

Once a module has been imported, it stays in memory until the program ends. There are ways to "unimport" a module (in other words, to erase it from memory) or to reimport a module that has changed while the program is running, but they are rarely used. In practice, it's almost always simpler to stop the program and restart it.

Modules can contain more than just functions. The math module, for example, also defines some variables like pi. Once the module has been imported, you can use these variables like any others:

Download modules/pi.cmd

```
>>> math.pi
3.1415926535897931
>>> radius = 5
>>> print 'area is %6f' % (math.pi * radius ** 2)
area is 78.539816
```

You can even assign to variables imported from modules:

Download modules/pi_change.cmd

```
>>> import math
>>> math.pi = 3 # would turn circles into hexagons
>>> radius = 5
>>> print 'circumference is', 2 * math.pi * radius
circumference is 30
```

Don't do this! Changing the value of π is not a good idea. In fact, it's such a bad idea that many languages allow programmers to define unchangeable *constants* as well as variables. As the name suggests, the value of a constant cannot be changed after it has been defined: π is always 3.14159 and a little bit, while SECONDS_PER_DAY is always 86,400. The fact that Python doesn't allow programmers to "freeze" values like this is one of the language's few significant flaws.

Combining the module's name with the names of the things it contains is safe, but it isn't always convenient. For this reason, Python lets you specify exactly what you want to import from a module, like this:

Download modules/from.cmd

```
>>> from math import sqrt, pi
>>> sqrt(9)
3.0
>>> radius = 5
>>> print 'circumference is %6f' % (2 * pi * radius)
circumference is 31.415927
```

This can lead to problems when different modules provide functions that have the same name. If you import a function called spell from a module called magic and then you import another function called spell from the module grammar, the second replaces the first. It's exactly like assigning one value to a variable, then another: the most recent assignment or import wins.

This is why it's usually *not* a good idea to use import *, which brings in everything from the module at once. It saves some typing:

Download modules/from2.cmd

```
>>> from math import *
>>> '%6f' % sqrt(8)
'2.828427'
```

but using it means that every time you add anything to a module, you run the risk of breaking every program that uses it.

The standard Python library contains several hundred modules to do everything from figuring out what day of the week it is to fetching data from a website. The full list is online at http://docs.python.org/modindex.html; although it's far too much to absorb in one sitting (or even one course), knowing how to use the library well is one of the things that distinguishes good programmers from poor ones.

4.2 Defining Your Own Modules

Section 2.1, *The Big Picture*, on page 19 explained that in order to save code for later use, you can put it in a file with a .py extension. You can then tell Python to run the code in that file, rather than typing commands in at the interactive prompt. What we didn't tell you then is that every Python file can be used as a module. The name of the module is the same as the name of the file, but without the .py extension.

For example, the following function is taken from Section 2.6, *Function Basics*, on page 32:

Download modules/convert.py

```
def to_celsius(t):
    return (t - 32.0) * 5.0 / 9.0
```

Put this function definition in a file called temperature.py, and then add another function called above_freezing that returns True if its argument's value is above freezing (in Celsius), and False otherwise:

Download modules/freezing.py

```
def above_freezing(t):
    return t > 0
```

Congratulations—you have now created a module called temperature:

Download modules/temperature.py

```
def to_celsius(t):
    return (t - 32.0) * 5.0 / 9.0

def above_freezing(t):
    return t > 0
```

Now that you've created this file, you can now import it like any other module:

Download modules/import_temp.cmd

```
>>> import temperature
>>> temperature.above_freezing(temperature.to_celsius(33.3))
True
```

The __builtins__ Module

Python's built-in functions are actually in a module named __builtins__. The double underscores before and after the name signal that it's part of Python; we'll see this convention used again later for other things. You can see what's in the module using help(__builtins__), or if you just want a directory, you can use dir instead (which works on other modules as well):

Download modules/dir1.cmd

```
>>> dir(__builtins__)
['ArithmeticError', 'AssertionError', 'AttributeError',
'BaseException', 'DeprecationWarning', 'EOFError', 'Ellipsis',
'EnvironmentError', 'Exception', 'False', 'FloatingPointError',
'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
'ImportWarning', 'IndentationError', 'IndexError', 'KeyError',
'KeyboardInterrupt', 'LookupError', 'MemoryError', 'NameError',
'None', 'NotImplemented', 'NotImplementedError', 'OSError',
'OverflowError', 'PendingDeprecationWarning', 'ReferenceError',
'RuntimeError', 'RuntimeWarning', 'StandardError',
'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
'SystemExit', 'TabError', 'True', 'TypeError',
'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning',
'UserWarning', 'ValueError', 'Warning', 'ZeroDivisionError', '_',
'__debug__', '__doc__', '__import__', '__name__', 'abs', 'all',
'any', 'apply', 'basestring', 'bool', 'buffer', 'callable',
'chr', 'classmethod', 'cmp', 'coerce', 'compile', 'complex',
'copyright', 'credits', 'delattr', 'dict', 'dir', 'divmod',
'enumerate', 'eval', 'execfile', 'exit', 'file', 'filter',
'float', 'frozenset', 'getattr', 'globals', 'hasattr', 'hash',
'help', 'hex', 'id', 'input', 'int', 'intern', 'isinstance',
'issubclass', 'iter', 'len', 'license', 'list', 'locals', 'long',
'map', 'max', 'min', 'object', 'oct', 'open', 'ord', 'pow',
'property', 'quit', 'range', 'raw_input', 'reduce', 'reload',
'repr', 'reversed', 'round', 'set', 'setattr', 'slice', 'sorted',
'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'unichr',
'unicode', 'vars', 'xrange', 'zip']
```

As of Python 2.5, 32 of the 135 things in __builtins__ are used to signal errors of particular kinds, such as SyntaxError and ZeroDivisionError. There are also functions called copyright, which tells you who holds the copyright on Python, and license, which displays Python's rather complicated license. We'll meet some of this module's other members in later chapters.

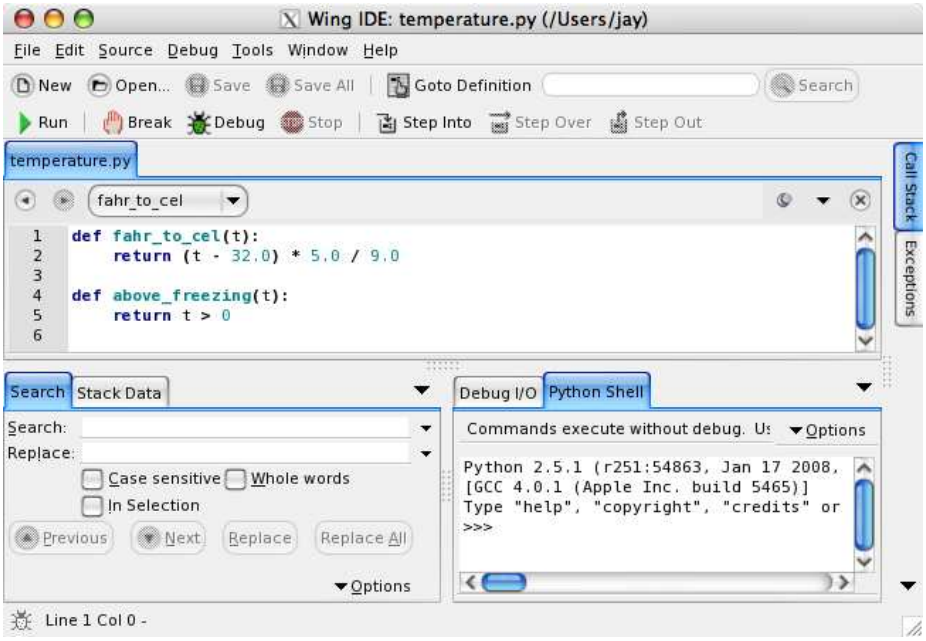

Figure 4.2: The temperature module in Wing 101

What Happens During Import

Let's try another experiment. Put the following in a file called experiment.py:

Download modules/experiment.py

```
print "The panda's scientific name is 'Ailuropoda melanoleuca'"
```

and then import it (or click Wing 101's Run button):

Download modules/import_experiment.cmd

```
>>> import experiment
The panda's scientific name is 'Ailuropoda melanoleuca'
```

What this shows is that *Python executes modules as it imports them*. You can do anything in a module you would do in any other program, because as far as Python is concerned, it's just another bunch of statements to be run.

Let's try another experiment. Start a fresh Python session, and try importing the experiment module twice in a row:

Download modules/import_twice.cmd

```
>>> import experiment
The panda's scientific name is 'Ailuropoda melanoleuca'
>>> import experiment
>>>
```

Notice that the message wasn't printed the second time. That's because Python loads modules only the first time they are imported. Internally, Python keeps track of the modules it has already seen; when it is asked to load one that's already in that list, it just skips over it. This saves time and will be particularly important when you start writing modules that import other modules, which in turn import other modules—if Python didn't keep track of what was already in memory, it could wind up loading commonly used modules like math dozens of times.

Using __main__

As we've now seen, every Python file can be run directly from the command line or IDE or can be imported and used by another program. It's sometimes useful to be able to tell inside a module which is happening, in other words, whether the module is the main program that the user asked to execute or whether some other module has that honor.

Python defines a special variable called __name__ in every module to help us figure this out. Suppose we put the following into echo.py:

Download modules/echo.py

```
print "echo: __name__ is", __name__
```

If we run this file, its output is as follows:

Download modules/echo.out

```
echo: __name__ is __main__
```

As promised, Python has created the variable __name__. Its value is "__main__", meaning, "This module is the main program."

But look at what happens when we import echo.py, instead of running it directly:

Download modules/echo.cmd

```
>>> import echo
echo: __name__ is echo
```

The same thing happens if we write a program that does nothing but import our echoing module:

Download modules/import_echo.py

```
import echo
print "After import, __name__ is", __name__, "and echo.__name__ is", echo.__name__
```

which, when run from the command line, produces this:

Download modules/import_echo.out

```
echo: __name__ is echo
After import, __name__ is __main__ and echo.__name__ is echo
```

What's happening here is that when Python imports a module, it sets that module's __name__ variable to be the name of the module, rather than the special string "__main__". This means that a module can tell whether it is the main program:

Download modules/test_main.py

```
if __name__ == "__main__":
    print "I am the main program"
else:
    print "Someone is importing me"
```

Try it, and see what happens when you run it directly and when you import it.

Knowing whether a module is being imported or not turns out to allow a few handy programming tricks. One is to provide help on the command line whenever someone tries to run a module that's meant to be used as a library. For example, think about what happens when you run the following on the command line vs. importing it into another program:

Download modules/main_help.py

```
'''
This module guesses whether something is a dinosaur or not.
'''

def is_dinosaur(name):
    '''
    Return True if the named create is recognized as a dinosaur,
    and False otherwise.
    '''
    return name in ['Tyrannosaurus', 'Triceratops']

if __name__ == '__main__':
    help(__name__)
```

We will see other uses in the following sections and in later chapters.

Providing Help

Let's return to the temperature module for a moment and modify it to round temperatures off. We'll put the result in temp_round.py:

Download modules/temp_round.py

```
def to_celsius(t):
    return round((t - 32.0) * 5.0 / 9.0)

def above_freezing(t):
    return t > 0
```

What happens if we ask for help on the function to_celsius?

Download modules/help_temp.cmd

```
>>> import temp_round
>>> help(temp_round)
Help on module temp_round:

NAME
    temp_round

FILE
    /home/pybook/modules/temp_round.py

FUNCTIONS
    above_freezing(t)

    to_celsius(t)
```

That's not much use: we know the names of the functions and how many parameters they need, but not much else. To provide something more useful, we should add *docstrings* to the module and the functions it contains and save the result in temp_with_doc.py:

Download modules/temp_with_doc.py

```
'''Functions for working with temperatures.'''

def to_celsius(t):
    '''Convert Fahrenheit to Celsius.'''
    return round((t - 32.0) * 5.0 / 9.0)

def above_freezing(t):
    '''True if temperature in Celsius is above freezing, False otherwise.'''
    return t > 0
```

Asking for help on this module produces a much more useful result.

Download modules/help_temp_with_doc.cmd

```
>>> import temp_with_doc
>>> help(temp_with_doc)
Help on module temp_with_doc:

NAME
    temp_with_doc - Functions for working with temperatures.

FILE
    /home/pybook/modules/temp_with_doc.py

FUNCTIONS
    above_freezing(t)
        True if temperature in Celsius is above freezing, False otherwise.

    to_celsius(t)
        Convert Fahrenheit to Celsius.
```

The term *docstring* is short for "documentation string." Docstrings are easy to create: if the first thing in a file or a function is a string that isn't assigned to anything, Python saves it so that help can print it later.

You might think that a module this small doesn't need much documentation. After all, it has only two functions, and their names are pretty descriptive of what they do. But writing documentation is more than a way to earn a few extra marks—it's essential to making software usable. Small programs have a way of turning into larger and more complicated ones. If you don't document as you go along and keep the documentation in the same file as the program itself, you will quickly lose track of what does what.

4.3 Objects and Methods

Numbers and strings may have been enough to keep programmers happy back in the twentieth century, but these days, people expect to work with images, sound, and video as well. A Python module called media provides functions for manipulating and viewing pictures; it isn't in the standard library, but it can be downloaded for free from http://code.google.com/p/pygraphics/. (One of the exercises discusses why it needs a separate download.)

In order to understand how media works, we first have to introduce two concepts that are fundamental to modern program design. And to do *that*, we have to back up and take another look at strings.

The Pragmatic Bookshelf

The Pragmatic Bookshelf features books written by developers for developers. The titles continue the well-known Pragmatic Programmer style and continue to garner awards and rave reviews. As development gets more and more difficult, the Pragmatic Programmers will be there with more titles and products to help you stay on top of your game.

Visit Us Online

Practical Programming's Home Page
http://pragprog.com/titles/gwpy
Source code from this book, errata, and other resources. Come give us feedback, too!

Register for Updates
http://pragprog.com/updates
Be notified when updates and new books become available.

Join the Community
http://pragprog.com/community
Read our weblogs, join our online discussions, participate in our mailing list, interact with our wiki, and benefit from the experience of other Pragmatic Programmers.

New and Noteworthy
http://pragprog.com/news
Check out the latest pragmatic developments, new titles and other offerings.

Buy the Book

If you liked this eBook, perhaps you'd like to have a paper copy of the book. It's available for purchase at our store: pragprog.com/titles/gwpy.

Contact Us

Online Orders:	www.pragprog.com/catalog
Customer Service:	support@pragprog.com
Non-English Versions:	translations@pragprog.com
Pragmatic Teaching:	academic@pragprog.com
Author Proposals:	proposals@pragprog.com
Contact us:	1-800-699-PROG (+1 919 847 3884)