CHAPTER 19

Playful Programming

At this point, you should have a clearer picture of how Python works than when you started. Now the rubber hits the road, so to speak, and in the next ten chapters you put your newfound skills to work. Each chapter contains a single do-it-yourself project with a lot of room for experimentation, while at the same time giving you the necessary tools to implement a solution.

In this chapter, I give you some general guidelines for programming in Python.

Why Playful?

I think one of the strengths of Python is that it makes programming fun—for me, anyway. It's much easier to be productive when you're having fun; and one of the fun things about Python is that it allows you to be very productive. It's a positive feedback loop, and you get far too few of those in life.

The expression *Playful Programming* is one I invented as a less extreme version of *Extreme Programming*, or XP.[1] I like many of the ideas of the XP movement but have been too lazy to commit completely to their principles. Instead, I've picked up a few things, and combined them with what I feel is a natural way of developing programs in Python.

The Jujitsu of Programming

You have perhaps heard of *jujitsu?* It's a Japanese martial art, which, like its descendants *judo* and *aikido*,[2] focuses on flexibility of response, or "bending instead of breaking." Instead of trying to impose your preplanned moves on an opponent, you go with the flow, using your opponent's movements against him. This way (in theory), you can beat an opponent who is bigger, meaner, and stronger than you.

How does this apply to programming? The key is the syllable "ju," which may be (very roughly) translated as flexibility. When you run into trouble while programming (as you invariably will), instead of trying to cling stiffly to your initial designs and ideas, be flexible. Roll with the punches. Be prepared to change and adapt. Don't treat unforeseen events as frustrating

1. Extreme Programming is an approach to software development that, arguably, has been in use by programmers for years, but that was first named and documented by Kent Beck. For more information, see `http://www.extremeprogramming.org`.
2. Or, for that matter, its Chinese relatives, such as *taijiquan* or *baguazhang*.

interruptions; treat them as stimulating starting points for creative exploration of new options and possibilities.

The point is that when you sit down and plan how your program should be, you don't have any real experience with that specific program. How could you? After all, it doesn't exist yet. By working on the implementation, you gradually learn new things that could have been useful when you did the original design. Instead of ignoring these lessons you pick up along the way, you should use them to redesign (or *refactor*) your software. I'm not saying that you should just start hacking away with no idea of where you are headed, but that you should prepare for change, and accept that your initial design *will* need to be revised. It's like the old writer's saying: "Writing is rewriting."

This practice of flexibility has many aspects; here I'll touch upon two of them:

Prototyping: One of the nice things about Python is that you can write programs quickly. Writing a prototype program is an excellent way to learn more about your problem.

Configuration: Flexibility comes in many forms. The purpose of configuration is to make it easy to change certain parts of your program, both for you and your users.

A third aspect, automated testing, is absolutely essential if you want to be able to change your program easily. With tests in place, you can be sure that your program still works after introducing a modification. Prototyping and configuration are discussed in the following sections. For information about testing, see Chapter 16.

Prototyping

In general, if you wonder how something works in Python, just try it. You don't need to do extensive preprocessing, such as compiling or linking, which is necessary in many other languages. You can just run your code directly. And not only that, you can run it piecemeal in the interactive interpreter, prodding at every corner until you thoroughly understand its behavior.

This kind of exploration doesn't cover only language features and built-in functions. Sure, it's useful to be able to find out exactly how, say, the `iter` function works, but even more important is the ability to easily create a prototype of the program you are about to write, just to see how *that* works.

■ **Note** In this context, the word *prototype* means a tentative implementation, a mock-up that implements the main functionality of the final program, but which may need to be completely rewritten at some later stage—or not. Quite often, what started out as a prototype can be turned into a working program.

After you have put some thought into the structure of your program (such as which classes and functions you need), I suggest implementing a simple version of it, possibly with very limited functionality. You'll quickly notice how much easier the process becomes when you have a running program to play with. You can add features, change things you don't like, and so on. You can really see how it works, instead of just thinking about it or drawing diagrams on paper.

You can use prototyping in any programming language, but the strength of Python is that writing a mock-up is a very small investment, so you're not committed to using it. If you find that your design wasn't as clever as it could have been, you can simply toss out your prototype and start from scratch. The process might take a few hours, or a day or two. If you were programming in C++, for example, much more work would probably be involved in getting something up and running, and discarding it would be a major decision. By committing to one version, you lose flexibility; you get locked in by early decisions that may prove wrong in light of the real-world experience you get from actually implementing it.

In the projects that follow this chapter, I consistently use prototyping instead of detailed analysis and design up front. Every project is divided into two implementations. The first is a fumbling experiment in which I've thrown together a program that solves the problem (or possibly only a part of the problem) in order to learn about the components needed and what's required of a good solution. The greatest lesson will probably be seeing all the flaws of the program in action. By building on this newfound knowledge, I take another, hopefully more informed, whack at it. Of course, you should feel free to revise the code, or even start afresh a third time. Usually, starting from scratch doesn't take as much time as you might think. If you have already thought through the practicalities of the program, the typing shouldn't take too long.

THE CASE AGAINST REWRITING

Although I'm advocating the use of prototypes here, there is reason to be a bit cautious about restarting your project from scratch at any point, especially if you've invested some time and effort into the prototype. It is probably better to refactor and modify that prototype into a more functional system, for several reasons.

One common problem that can occur is "second system syndrome." This is the tendency to try to make the second version so clever or perfect that it's never finished.

The "continual rewriting syndrome," quite prevalent in fiction writing, is the tendency to keep fiddling with your program, perhaps starting from scratch again and again. At some point, leaving well enough alone may be the best strategy—just get something that *works*.

Then there is "code fatigue." You grow tired of your code. It seems ugly and clunky to you after you've worked with it for a long time. Sadly, one of the reasons it may seem hacky and clunky is that it has grown to accommodate a range of special cases, and to incorporate several forms of error handling and the like. These are features you would need to reintroduce in a new version anyway, and they have probably cost you quite a bit of effort (not the least in the form of debugging) to implement in the first place.

In other words, if you think your prototype could be turned into a workable system, by all means, keep hacking at it, rather than restarting. In the project chapters that follow, I have separated the development cleanly into two versions: the prototype and the final program. This is partly for clarity and partly to highlight the experience and insight one can get by writing the first version of a piece of software. In the real world, I might very well have started with the prototype and "refactored myself" in the direction of the final system.

For more on the horrors of restarting from scratch, take a look at Joel Spolsky's article "Things You Should Never Do, Part I" (found on his web site, `http://joelonsoftware.com`). According to Spolsky, rewriting the code from scratch is the single worst strategic mistake that any software company can make.

Configuration

In this section, I return to the ever important principle of abstraction. In Chapters 6 and 7, I showed you how to abstract away code by putting it in functions and methods, and hiding larger structures inside classes. Let's take a look at another, much simpler, way of introducing abstraction in your program: extracting *symbolic constants* from your code.

Extracting Constants

By *constants*, I mean built-in literal values such as numbers, strings, and lists. Instead of writing these repeatedly in your program, you can gather them in global variables. I know I've been warning you about those, but problems with global variables occur primarily when you start changing them, because it can be difficult to keep track of which part of your code is responsible for which change. I'll leave these variables alone, however, and use them as if they were constant (hence the term *symbolic constants*). To signal that a variable is to be treated as a symbolic constant, you can use a special naming convention, using only capital letters in their variable names and separating words with underscores.

Let's take a look at an example. In a program that calculates the area and circumference of circles, you could keep writing 3.14 every time you needed the value π. But what if you, at some later time, wanted a more exact value, say 3.14159? You would need to search through the code and replace the old value with the new. This isn't very hard, and in most good text editors, it could be done automatically. However, what if you had started out with the value 3? Would you later want to replace every occurrence of the number 3 with 3.14159? Hardly. A much better way of handling this would be to start the program with the line `PI = 3.14`, and then use the name `PI` instead of the number itself. That way, you could simply change this single line to get a more exact value at some later time. Just keep this in the back of your mind: whenever you write a constant (such as the number 42 or the string "Hello, world!") more than once, consider placing it in a global variable instead.

■ **Note** Actually, the value of π is found in the `math` module, under the name `math.pi`:

```
>> from math import pi
>> pi
3.1415926535897931
```

This may seem agonizingly obvious to you. But the real point of all this comes in the next section, where I talk about configuration files.

Configuration Files

Extracting constants for your own benefit is one thing, but some constants can even be exposed to your users. For example, if they don't like the background color of your GUI program, perhaps you should let them use another color. Or perhaps you could let users decide

what greeting message they would like to get when they start your exciting arcade game or the default starting page of the new web browser you just implemented.

Instead of putting these configuration variables at the top of one of your modules, you can put them in a separate file. The simplest way of doing this is to have a separate module for configuration. For example, if `PI` is set in the module file `config.py`, you can (in your main program) do the following:

```
from config import PI
```

Then, if the user wants a different value for `PI`, she can simply edit `config.py` without having to wade through your code.

■ **Caution** There is a trade-off with the use of configuration files. On the one hand, configuration is useful, but using a central, shared repository of variables for an entire project can make it less modular and more monolithic. Make sure you're not breaking abstractions (such as encapsulation).

Another possibility is to use the standard library module `ConfigParser`, which will allow you to use a reasonably standard format for configuration files. It allows both standard Python assignment syntax, such as this:

```
greeting = 'Hello, world!'
```

(although this would give you two extraneous quotes in your string) and another configuration format used in many programs:

```
greeting: Hello, world!
```

You must divide the configuration file into *sections*, using headers such as `[files]` or `[colors]`. The names can be anything, but you need to enclose them in brackets. A sample configuration file is shown in Listing 19-1, and a program using it is shown in Listing 19-2. For more information about the features of the `ConfigParser` module, consult the library documentation (`http://python.org/doc/lib/module-ConfigParser.html`).

Listing 19-1. *A Simple Configuration File*

```
[numbers]

pi: 3.1415926535897931

[messages]

greeting: Welcome to the area calculation program!
question: Please enter the radius:
result_message: The area is
```

Listing 19-2. *A Program Using ConfigParser*

```
from ConfigParser import ConfigParser

CONFIGFILE = "python.txt"

config = ConfigParser()
# Read the configuration file:
config.read(CONFIGFILE)

# Print out an initial greeting;
# 'messages' is the section to look in:
print config.get('messages', 'greeting')

# Read in the radius, using a question from the config file:
radius = input(config.get('messages', 'question') + ' ')

# Print a result message from the config file;
# end with a comma to stay on same line:
print config.get('messages', 'result_message'),

# getfloat() converts the config value to a float:
print config.getfloat('numbers', 'pi') * radius**2
```

I won't go into much detail about configuration in the following projects, but I suggest you think about making your programs highly configurable. That way, users can adapt the program to their tastes, which can make using it more pleasurable. After all, one of the main frustrations of using software is that you can't make it behave the way you want it to.

LEVELS OF CONFIGURATION

Configurability is an integral part of the UNIX tradition of programming. In Chapter 10 of his excellent book, *The Art of UNIX Programming* (Addison-Wesley, 2003), Eric S. Raymond describes the following three sources of configuration or control information, which (if included) should probably be consulted *in this order*,[3] so the later sources override the earlier ones:

- **Configuration files**: See the "Configuration Files" section in this chapter.
- **Environment variables**: These can be fetched using the dictionary `os.environ`.
- **Switches and arguments passed to the program on the command line**: For handling command-line arguments, you can use `sys.argv` directly. If you want to deal with switches (options), you should check out the `optparse` module (or perhaps `getopt`), as mentioned in Chapter 10.

3. Actually, global configuration files and system-set environment variables come before these. See the book for more details.

Logging

Somewhat related to testing (discussed in Chapter 16), and quite useful when furiously reworking the innards of a program, logging can certainly help you discover problems and bugs. Logging is basically collecting data about your program as it runs, so you can examine it afterward (or as the data accumulates, for that matter). A very simple form of logging can be done with the `print` statement. Just put a statement like this at the beginning of your program:

```
log = open('logfile.txt', 'w')
```

You can then later put any interesting information about the state of your program into this file, as follows:

```
print >> log, ('Downloading file from URL %s' % url)
text = urllib.urlopen(url).read()
print >> log, 'File successfully downloaded'
```

This approach won't work well if your program crashes during the download. It would be safer if you opened and closed your file for every `log` statement (or, at least, flushed the file after writing). Then, if your program crashed, you could see that the last line in your log file said "Downloading file from . . ." and you would know that the download wasn't successful.

The way to go, actually, is using the `logging` module in the standard library. Basic usage is pretty straightforward, as demonstrated by the program in Listing 19-3.

Listing 19-3. *A Program Using the logging Module*

```
import logging

logging.basicConfig(level=logging.INFO, filename='mylog.log')

logging.info('Starting program')

logging.info('Trying to divide 1 by 0')

print 1 / 0

logging.info('The division succeeded')

logging.info('Ending program')
```

Running that program would result in the following log file (called `mylog.log`):

```
INFO:root:Starting program
INFO:root:Trying to divide 1 by 0
```

As you can see, nothing is logged after trying to divide 1 by 0 because this error effectively kills the program. Because this is such a simple error, you can tell what is wrong by the exception traceback that prints as the program crashes. The most difficult type of bug to track down

is one that doesn't stop your program, but simply makes it behave strangely. Examining a detailed log file may help you find out what's going on.

The log file in this example isn't very detailed, but by configuring the `logging` module properly, you can set up just how you want your logging to work. Here are a few examples:

- Log entries of different types (information, debug info, warnings, custom types, and so on). By default, only warnings are let through (which is why I explicitly set the level to `logging.INFO` in Listing 19-3).
- Log just items that relate to certain parts of your program.
- Log information about time, date, and so forth.
- Log to different locations, such as sockets.
- Configure the logger to filter out some or most of the logging, so you get only what you need at any one time, without rewriting the program.

The `logging` module is quite sophisticated, and there is much to be learned in the documentation (`http://python.org/doc/lib/module-logging.html`).

If You Can't Be Bothered

"All this is well and good," you may think, "but there's no way I'm going to put that much effort into writing a simple little program. Configuration, testing, logging—it sounds really boring."

Well, that's fine. You may not need it for simple programs. And even if you're working on a larger project, you may not really *need* all of this at the beginning. I would say that the minimum is that you have some way of testing your program (as discussed in Chapter 16), even if it's not based on automatic unit tests. For example, if you're writing a program that automatically makes you coffee, you should have a coffee pot around, to see if it works.

In the project chapters that follow, I don't write full test suites, intricate logging facilities, and so forth. I present you with some simple test cases to demonstrate that the programs work, and that's it. If you find the core idea of a project interesting, you should take it further—try to enhance and expand it. And in the process, you should consider the issues you read about in this chapter. Perhaps a configuration mechanism would be a good idea? Or a more extensive test suite? It's up to you.

If You Want to Learn More

Just in case you want more information about the art, craft, and philosophy of programming, here are some books that discuss these things more in depth:

- *The Pragmatic Programmer*, by Andrew Hunt and David Thomas (Addison-Wesley, 1999)
- *Refactoring*, by Kent Beck et al. (Addison-Wesley, 1999)
- *Design Patterns*, by the "Gang of Four," Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides (Addison-Wesley, 1994)
- *Test-Driven Development: By Example*, by Kent Beck (Addison-Wesley, 2002)

- *The Art of UNIX Programming*, by Eric S. Raymond (Addison-Wesley, 2003)[4]
- *Introduction to Algorithms, Second Edition*, by Thomas H. Cormen et al. (MIT Press, 2001)
- *The Art of Computer Programming*, Volumes 1–3, by Donald Knuth (Addison-Wesley, 1998)
- *Concepts, Techniques, and Models of Computer Programming*, by Peter Van Roy and Seif Haridi (MIT Press, 2004)

Even if you don't read every page of every book (I know I haven't), just browsing through a few of these can give you quite a lot of insight.

A Quick Summary

In this chapter, I described some general principles and techniques for programming in Python, conveniently lumped under the heading "Playful Programming." Here are the highlights:

Flexibility: When designing and programming, you should aim for flexibility. Instead of clinging to your initial ideas, you should be willing to—and even prepared to—revise and change every aspect of your program as you gain insight into the problem at hand.

Prototyping: One important technique for learning about a problem and possible implementations is to write a simple version of your program to see how it works. In Python, this is so easy that you can write several prototypes in the time it takes to write a single version in many other languages. Still, you should be wary of rewriting your code from scratch if you don't have to—refactoring is usually a better solution.

Configuration: Extracting constants from your program makes it easier to change them at some later point. Putting them in a configuration file makes it possible for your users to configure the program to behave as they would like. Employing environment variables and command-line options can make your program even more configurable.

Logging: Logging can be quite useful for uncovering problems with your program—or just to monitor its ordinary behavior. You can implement simple logging yourself, using the `print` statement, but the safest bet is to use the `logging` module from the standard library.

What Now?

Indeed, what now? Now is the time to take the plunge and really start programming. It's time for the projects.

All ten project chapters have a similar structure, with the following sections:

What's the Problem?: In this section, the main goals of the project are outlined, including some background information.

Useful Tools: Here, I describe modules, classes, functions, and so on that might be useful for the project.

4. Also available online at Raymond's web site (`http://catb.org/~esr/writings/taoup`).

Preparations: This section covers any preparations necessary before starting to program. This may include setting up the necessary framework for testing the implementation.

First Implementation: This is the first whack—a tentative implementation to learn more about the problem.

Second Implementation: After the first implementation, you will probably have a better understanding of things, which will enable you to create a new and improved version.

Further Exploration: Finally, I give pointers for further experimentation and exploration.

Let's get started with the first project, which is to create a program that automatically marks up files for HTML.