Lean Python

Paul Gerrard is a consultant, teacher, author, webmaster, programmer, tester, conference speaker, rowing coach and publisher. He has conducted consulting assignments in all aspects of software testing and quality assurance, specialising in test assurance. He has presented keynote talks and tutorials at testing conferences across Europe, the USA, Australia, South Africa and occasionally won awards for them.

Educated at the universities of Oxford and Imperial College London, he is a Principal of Gerrard Consulting Limited, the host of the UK Test Management Forum and the Programme Chair for the 2014 EuroSTAR testing conference.

In 2010 he won the EuroSTAR Testing Excellence Award and in 2013 he won the inaugural TESTA Lifetime Achievement Award.

He's been programming since the mid-1970s and loves using the Python programming language.

Lean Python

Just Enough Python to Build Useful Tools

1st Revision

Paul Gerrard

The Tester's Press
Maidenhead UK

Publishers note

Every possible effort has been made to ensure that the information contained in this book is accurate at the time of going to press, and the publishers and authors cannot accept any responsibility for any errors or omissions, however caused. No responsibility for loss or damage occasioned by any person acting, or refraining from action, as a result of the material in this publication can be accepted by the editor, the publisher or the authors.

First published in Great Britain in 2014 by
THE TESTER'S PRESS
1 Old Forge Close
Maidenhead
Berkshire SL6 2RD
United Kingdom
Web: testers-press.com

ISBN-10 0956-19622-4
ISBN-13 978-0956-19622-4

LeanPython150325.docx

Front cover image:
http://www.123rf.com/photo_5574051_3d-rendering-of-a-green-tree-python.html

Typeset by Paul Gerrard
Printed and bound by Lulu.com

Contents

Preface

The fun of programming

My first exposure to computer programming was at school nearly 40 years ago. My maths teacher was a fan of computing and he established the first A-Level Computer Science course in the sixth form college. I didn't take the CS A-Level as I was committed to Maths, Physics and Chemistry. But my Maths teacher invited all the scientists to do an informal class in programming, once a week, after-hours. It sounded interesting so I enrolled.

We were introduced to a programming language called CESIL[1]. CESIL was a cut-down version of an Assembler language[2] with instructions that had more meaningful names like LOAD, STORE, ADD and JUMP. We were given green cards on which the instructions and numbers were printed. Next to each instruction was a small oval shape. Beyond that, there was a shape for every letter and numeric value.

Filling in the shapes with a pencil indicated the instructions and data we wanted to use. To make the 'job' work we topped and tailed our card deck with some standard instructions on more cards.

---

[1] Computer Education in Schools Instruction Language, **http://en.wikipedia.org/wiki/Cesil**. If you are curious, you can download a fully working CESIL interpreter [18].

[2] Assembler is a very low-level language close to actual 'machine code'.

Our card decks were secured with rubber bands and sent off to Manchester University for processing. A week later, we usually (but not always) got our cards back together with a printout of the results. If we were lucky, our trivial programs generated some results. More often, our programs did not work, or did not even compile – that is, the computer did not understand our stumbling attempts to write meaningful program code.

I can't remember what programs I wrote in those days. Probably calculating squares of integers or factorials or if I was really ambitious, the Sine of an angle using Taylor series. Looping (and more often, infinite looping) was a wonderful feature that had to be taken advantage of. Doing something that simply could not be done by humans was fascinating to me.

The challenge of 'thinking like the computer' and of treating the mysterious machine in Manchester as an infallible wizard that must be obeyed – or at least communicated with in its own pedantic, arcane language sticks in my mind. You could, with some practice, treat the wizard as your very own, tireless slave. Those after-hours classes were great and I looked forward to them every week.

Programming was great fun – if you had a certain interest in control, procedure and systematic thinking. Nearly forty years later, I still enjoy battling with code. My programming language of choice nowadays is Python[3].

Introducing Python

The Python programming language was created by Dutchman Guido van Rossum in the late 1980s. See [1] for an interview

[3] Throughout the book, I will use the term Python as shorthand for "The Python Programming Language".

with Guido on the origins and design philosophy of the language.

Here is a concise summary of Python from Wikipedia [2]:

> *"Python is a widely used general-purpose, high-level programming language. Its design philosophy emphasizes code readability, and its syntax allows programmers to express concepts in fewer lines of code than would be possible in languages such as C. The language provides constructs intended to enable clear programs on both a small and large scale."*

If you choose to learn Python as your first or your fifteenth programming language, you are making an excellent choice.

Of all the languages I have used (and I think it is about fifteen, over the years) Python is my favourite. I can't say exactly why, and I don't pretend to be an expert in these matters, but things I like about Python are:

- Programs are not cluttered up with braces ({...}) and semi-colons (;)
- Python implements structure using indentation (white space) rather than punctuation
- The Python keywords are powerful, limited in number and do what you imagine they do
- If you can't work out a way to do something in your code, there is always a library somewhere that does it for you
- You can get an awful lot done with a limited knowledge of the language.

It is this last feature that I like the most.

Lean Python

I freely admit that I don't know all the features of this wonderful language off by heart. In that way, I am a less-than perfect programmer and I beat myself up about it regularly. Not. I have written about forty thousand lines of Python in the past five years, but I discovered recently that actually, I only need a distinct sub-set of the language to get things done. I do use all the core elements of the language of course – lists, dictionaries, objects and so on, but I don't (can't) memorise all of the standard functions for each element. I haven't needed them.

I'm looking at a list of the functions and methods for sequences. There are 58 listed in my main Python source book [13]. I have only used 15 of them – I haven't found a need for the rest.

I call this subset ***Lean Python*** and it is all you need to know as a beginner and some way beyond.

> Lean Python is not "the best way to write code". I offer it as a way of learning the essential aspects of the language without cluttering up your mind with features you may never use.

Now, the code I have written with the Lean Python subset of language features means that on occasion, I have written less optimal code. For example, I discovered only recently that there is a **`reverse()`** function that provides a list in reverse order. Of course there is, and why wouldn't there be? Needless to say, I had overlooked this neat feature and have written code to access list elements in reverse order more than once.

These things happen to all programmers. In general we don't consult the manual unless we have to. So it's a good idea, every now and then, to review the standard list of features for the language to see what might be useful in the future.

Beyond Lean Python

There are many excellent resources available that provide more comprehensive content than this little book. Web sites I would recommend as 'essential' are:

- python.org - the official site for the Python language is often the best starting point
- docs.python.org – the definitive documentation of the standard Python libraries

There are several excellent sites that offer free, online tutorials. Search for 'python tutorial' and you'll see what I mean.

Regarding books, there are three that sit on a shelf right above my desk at all times:

- Core Python Programming, Wesley Chun
- The Python Standard Library by Example, Doug Hellmann
- Python Cookbook, Martelli, Ravenscroft, Ascher

There are many other excellent books, and you might find better ones, but these are the three that I use myself.

Code examples in the book

In this book, you will see quite a lot of example code. Early on you'll see some small code fragments with some narrative text. All code listings are presented in the **`Courier New`** font. The shaded text is the code, the unshaded txt to the right provides some explanation.

```
#
# some comments and code
# in here
#
myName = 'Paul'
```

Some explanation appears on the right hand side.

```
myAge = 21 # if only
```

Later on you'll see longer listings and whole programs. These appear in the book as shaded areas. Some listings have line numbers on the left for reference – the line numbers are not part of the program code. For example:

```
1    def len(seq):
2        if type(seq) in [list,dict]:  # is it a seq?
3            return -1                 # if not, fail!
4        nelems=0                      # length is zero
5        for elem in seq:              # for each elem
6            nelems+=1                 # +1 to length
7
8        return nelems                 # return length
```

There are also some examples of interactions with the Python command-line shell. The shell gives you the **>>>** prompt. Here's an example:

```
>>> type(23)
<type 'int'>
>>> type('some text')
<type 'str'>
>>> c=[1,2,'some more text']
>>> type(c)
<type 'list'>
```

The lines are not numbered. The lines without the **>>>** prompt are the outputs printed by the shell.

Use the code fragments in the shaded sections to practice on in the interactive interpreter or run the programs for yourself.

Target audience

This book is aimed at three categories of reader:

- The experienced programmer – if you already know a program language, this book gives you a shortcut to understanding the Python language and some of its design philosophy.

- You work in IT and need a programming primer – you might be a tester who needs to have more informed technical discussions with programmers. Working through the examples will help you to appreciate the challenge of good programming.
- First-timer – you want a first book on programming that you can assimilate quickly to help you decide whether programming is for you.

If you require a full-fat 1000-page reference book for the Python language, this book is not for you. If you require a primer, appetiser or basic reference, this book should satisfy your need.

What this Book is

This little book provides a sequential learning guide to a useful and usable subset of the Python programming language. Its scope and content is deliberately limited and based on my own experience of using Python to build interactive websites (using the Web2py web development framework [3]) and many command-line utilities.

This book accompanies the 1 and 2-day programming courses that I created to help people to grasp the basics of a programming language quickly. It isn't a full language reference book, but a reference for people on the course and for whom the ***Lean Python*** subset is enough (at least initially).

What this Book is Not

This book is not intended to be a definitive guide to Python.

Code Comprehension

The initial motivation for writing this book was to help non-technical (i.e. non-programmer) testers to have an appreciation of programming so they could work more closely with the professional programmers on their teams. Critical to this, is the skill I call ***Code Comprehension*** which is your ability to read and understand program code.

Like spoken and written languages, it is usually easier to comprehend written language than write it from scratch. If the book helps you to appreciate and understand written program code then the book will have succeeded in its first goal.

Python Style Guidelines

One of the most important attributes of code is that it is written to be read by people, not just computers. The Python community give this goal a high priority. In your own company, you may already have programming or Python guidelines; the Python team have provided some that are widely used [4].

I have tried to follow the guidelines in the sample code and programs. However in the pocketbook book format, there is less horizontal space so sometimes I have had to squeeze code a little to fit it on the page. I tend to use mixed case e.g. **`addTwoNumbers`** in my variable and function names[4].

Some of my code comments, particularly in the early pages, are there to explain what, for example, an assignment does. You would not normally expect to see such 'stating the obvious' comments in real code.

---

[4] The guideline suggests **`lower_case_with_underscores`**.

'Pythonistas' take the readability goal seriously; so should you.

There is also a set of design principles you might consult. The Zen of Python sets them out [5].

I'm sure I could have written better examples – if you see an opportunity to improve readability or design, let me know.

Structure

The first seven chapters cover the core features of Python. The later chapters introduce some key libraries and how you can use them to write useful applications.

Chapter 1 introduces the interpreter, the basic syntax of the language, the normal layout and conventions of Python.

Chapter 2 describes the core Python objects that you will use and need to understand.

Chapter 3 sets out how programs are structured and controlled using decisions and loops.

Chapter 4 tells you how to get data into and out of your programs with the command line, display and disk files.

Chapter 5 introduces modules that help you to manage your own code and access the thousands of existing libraries.

Chapter 6 gives you a flavour for Object Orientation. Objects and classes are the key building blocks that programmers use.

Chapter 7 presents methods for trapping errors and exceptions to allow your programs to be 'under control' whatever happens.

Chapter 8 describes how you can use the **`unittest`** framework to test your code in a professional manner.

Chapter 9 introduces libraries allowing you to create a 'web client' and download pages from web sites.

Chapter 10 presents regular expressions as the mechanism for more sophisticated searching and pattern-matching.

Chapter 11 gives you techniques for creating and using the SQLite relational database for persistent storage.

Chapter 12 asks 'What Next?' and offers some suggestions for further development of your Python programming skills.

An Appendix contains references to websites, books and tools, and the Python exception hierarchy.

An Index is included at the end of the Pocketbook.

Using Python

Downloading Python

All Python downloads can be found here:

https://www.python.org/downloads/

You need to choose a Python version before you download. There are currently two versions:

- Version 2 is coming towards the end of its life but is still widely used.
- Version 3 has been around for some time; people have been slow to convert but it is gaining a following.

The example code in this book assumes you are using Version 3. If you use Python version 2 you will notice a few differences. You can read a discussion of the two Python versions in [6].

Sample Programs download

Downloadable sample programs can be found at:

http://leanpy.com/?page_id=37

All the sample programs have been tested on Windows 8, Ubuntu Linux 13 and my trusty Raspberry Pi running Linux. If you use a Mac, you should not have problems.

External Libraries

A major benefit to using Python is the enormous range of free libraries that are available for use. The vast majority of these libraries can be found on the PyPI site [7]. When I last looked, there were 46,554 packaged libraries hosted there.

Depending on your operating system (Windows, Mac or Linux), there are several ways of performing installations of Python libraries. The one I find easiest to use is the PIP installer [19] which works nicely with the PyPI site.

Editing your Python code

We recommend using either a language-sensitive editor or the editor that comes with your Python installation.

- On Windows – use the IDLE Integrated Development Environment (IDE) or perhaps Notepad++.
- On Linux, there is a selection of editors – **`vi`**, **`vim`**, **`emacs`**, **`gedit`** and so on – I use **`gedit`**.
- On OS X, TextMate works fine, but there are other options.

When you are experienced, you might upgrade to using an Integrated Development Environment. There is a list of Python compatible IDEs here:

https://wiki.python.org/moin/IntegratedDevelopmentEnvironments

Feedback, please!

I am very keen to receive your feedback and experience to enhance the format and content of the book. Give me feedback and I'll acknowledge you in the next edition.

Any errors or omissions are my fault entirely. Please let me know how I can improve this Pocketbook. Email me at **paul@gerrardconsulting.com** with suggestions or errors.

Downloads, errata, further information and a reading list can be found on the book's website: **leanpy.com**.

Acknowledgements

For their helpful feedback, guidance and encouraging comments, I'd like to thank James Lyndsay, Corey Goldberg, Simon Knight, Neil Studd, Srinivas Kadiyala, Julian Harty, Fahad Ahmed.

1st Revision Notes

The following changes were made to the first edition:

- Some typos corrected
- The Chapter 9 program example uses the more modern requests library.

"Everyone knows that debugging is twice as hard as writing a program in the first place. So if you're as clever as you can be when you write it, how will you ever debug it?"
Brian W. Kernighan

"Talk is cheap. Show me the code"
Linus Torvalds

"Programs must be written for people to read, and only incidentally for machines to execute"
Abelson / Sussman

"First, solve the problem. Then, write the code"
John Johnson

"Sometimes it pays to stay in bed on Monday, rather than spending the rest of the week debugging Monday's code"
Dan Salomon

"This project is seriously ahead of schedule"
Perplexed IT director

"The most disastrous thing that you can ever learn is your first programming language"
Alan Kay

1 Getting Started

The Python Interpreter

The Python interpreter is a program that reads Python program statements and executes them immediately. There is full documentation here [8]. To use the interpreter, you will need to open a terminal window or command prompt on your workstation. The interpreter operates in two modes[5].

Interactive mode

You can use the interpreter as an interactive tool. In interactive mode, you run the python program and you will see a new prompt **>>>** and you can then enter Python statements one by one. In Windows, you might see something like this:

```
C:\Users\Paul>python
Python 3.4.3 (v3.4.3:9b73f1c3e601, Feb 24 2015,
22:43:06) [MSC v.1600 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for
more information.
>>> _
```

The interpreter executes program statements immediately. Interactive mode is really useful when you want to experiment or try things out. For example, sometimes, you need to see how a particular function (that you haven't used before) behaves. On other occasions, you might need to see exactly what a piece of failing code does in isolation.

[5] There are a number of flags and options you can use with the interpreter, but we won't need them.

The **>>>** prompt can be used to enter one-line commands, or code blocks that define classes or functions (see later). Some example commands are shown below:

```
1   >>> dir(print)
2   ['__call__', '__class__', '__delattr__',
    '__dir__', '__doc__', '__eq__', '__format__',
    '__ge__', '__getattribute__', '__gt__',
    '__hash__', '__init__', '__le__', '__lt__',
    '  module  ','  name  ','  ne  ', '  new  ',
    '  qualname  ', '  reduce  ', '  reduce ex  ',
    '  repr  ', '  self  ', '  setattr  ',
    '__sizeof__', '__str__', '__subclasshook__',
    '__text_signature__']
3   >>>
4   >>> 123.456 + 987.654
5   1111.11
6   >>>
7   >>> 'This'+'is'+'a'+'joined'+'up'+'string.'
8   'Thisisajoinedupstring.'
9   >>>
10  >>> len('Thisisajoinedupstring.')
11  22
```

The **dir()** command on line 1 lists all the attributes of an object – helpful if you need to know what you can do with an object type. **dir()** run without an argument tells you what modules you have available. **dir(print)** shows a list of all the built-in methods for **print()** (most of which you'll never need).

If you type an expression value, as in line 4, **123.456+987.654** the interpreter will execute the calculation and provide the result. The expression on line 7 joins the strings of characters into one long string. The **len()** function on line 10 gives you the length of a string in characters.

If you define a new function[6] in the interactive mode, the interpreter prompts you to complete the definition and will treat a blank line as the end of the function.

[6] We'll cover these later, of course.

```
1    >>> def addTwoNumbers(a,b):
2    ...      result = a + b
3    ...      return result
4    ...
5    >>> addTwoNumbers(3,6)
6    9
7    >>>
```

Note that when the interpreter expects more code to be supplied in a function for example, it prints the ellipsis prompt "`...`". In the case of function definitions, a blank line (see line 4 above) completes the function definition.

We define the function in lines 1-3 (note the indentation), the blank line 4 ends the definition. We call the function on line 5 and add 6+3, and the result is (correctly) 9.

One other feature of the interactive interpreter is the **`help()`** function. You can use this to see the documentation of built-in keywords and functions. For example:

```
>>> help(open)
Help on built-in function open in module io:

open(...)
    open(file, mode='r', buffering=-1, encoding=None,
errors=None, newline=None, closefd=True, opener=None)
-> file object

... etc. etc.
```

The Python interactive interpreter is really handy to try things out and explore features of the language.

Command line mode

In command line mode, the Python program is still run at the command line but you add the name of a file (that contains your program) to the command:

`python myprogram.py`

The interpreter reads the contents of the file (**`myprogram.py`** in this case), scans and validates the code, compiles the program

and then executes the program. If the interpreter encounters a fault in the syntax of your program, it will report a 'compilation error'. If the program fails during execution you will see a 'run time error'. If the program executes successfully, you will see the output(s) of the program.

You don't need to worry about how the interpreter does what it does, but you do need to be familiar with the types of error messages it produces.

We use command-line mode to execute our programs in files.

Coding, testing and debugging Python programs

The normal sequence of steps, when creating a new program is:

1. Create a new .py file that will contain the Python program (sometimes called 'source code')
2. Edit your .py file to create new code (or amend existing code)
3. Run your program at the command prompt to test it, and interpret the outcome
4. If the program does not work as required, or you need to add more features, figure out what changes are required and go to step 2.

It's usually a good idea to document your code with comments. This is part of the editing process, step 2. If you need to make changes to a working program, again, you start at step 2.

Writing new programs is often called *coding*. When your programs don't work properly, getting programs to do exactly what you want them to do is often called *debugging*.

Comments, Code Blocks and Indentation

Python, like all programming languages has conventions that we must follow. Some programming languages use punctuation such as braces {} and semi-colons ; to structure code blocks. Python is somewhat different (and easier on the eye) because it uses white space and indentation to define code structure[7]. Sometimes code needs a little explanation so we use comments to help readers of the code (including you) to understand it.

We'll introduce indentation and comments with some examples.

```
#
# some text after hashes
#

brdr = 2 # thick border

def my_func(a, b, c):
    d = a + b + c
    ...
    ...

if this var==23:
    doThis()
    doThat()
    ...
else:
    do_other()
    ...


def addTwoNumbers(a, b):
    "adds two numbers"
    return a + b
```

Any text that appears after a hash character (#) is ignored by the interpreter and treated as a comment. We use comments to provide documentation.

The colon character (:) denotes the end of a header line that demarks a code block. The statements that follow the header line should be indented.

Colons are most often used at the end of **`if`**, **`elif`**, **`else`**, **`while`** and **`for`** statements, and function definitions (that start with the **`def`** keyword). In this example the text in quotes is a 'docsctring'. This text is what a **`help(addTwoNumbers)`** command would display in the interactive interpreter.

[7] Python 3 disallows mixed spaces and tabs, by the way (Unlike v2).

```
if long_var is True && \
    middle==10 && \
    small var is False:
    …
    …
```

The backslash character \ at the end of the line indicates that the statement extends onto the next line. Some very long statements might extend over several lines.

```
xxxxxxxxxxxxxx:
    xxxxxxxxxxxx
    xxxxxxxxxxxx
    xxxxxxxxxxxx
xxxxxxxxx:
    xxxxxxxxxxx:
        xxxxxxx
        xxxxxxxxxxx:
            xxxx
            xxxx
```

All code blocks are indented, once a header line with a colon appears. All the statements in that block must have the same indentation.

Code blocks can be nested within each other. Same rule - all code in a block has the same indentation.

Indentation is most often achieved using 4-space increments.

```
a = b + c ; p = q + r

a = b + c
p = q + r
```

The semicolon character (;) can be used to join multiple statements in a single line. The first line is equivalent to the two lines that follow it.

Variables

A variable is a named location in the program's memory that can be used to store some data. There are some rules for naming variables:

- First character must be a letter or underscore(_)
- Additional characters may be alphanumeric or underscore
- Names are case-sensitive.

Common Assignment Operations

When you store data in a variable it is called assignment. An assignment statement places a value or the result of an expression into variable(s). The general format of an assignment is:

```
var = expression
```

An **expression** could be a literal, a calculation or a call to a function or a combination of all three. Some expressions generate a list of values for example:

```
var1, var2, var3 = expression
```

See some more examples below:

```
>>> # 3 into integer myint
>>> myint = 3
>>>
>>> # a string of characters into a string variable
>>> text = 'Some text'
>>> # a floating point number
>>> cost = 3 * 123.45
>>> # a longer string
>>> Name = 'Mr' + ' ' + 'Fred' + ' ' + 'Bloggs'
>>> # a list
>>> shoppingList = ['ham','eggs','mushrooms']
>>> # multiple assignment (a=1, b=2, b=3)
>>> a, b, c = 1, 2, 3
```

Other assignment operations

Augmented assignment provides a slightly shorter notation, where a variable has its value adjusted in some way.

This assignment	*Is equivalent to*
`x+=1`	`x = x + 1`
`x-=23`	`x = x - 23`
`x/=6`	`x = x / 6`
`x*=2.3`	`x = x * 2.3`

Multiple assignment provides a slightly shorter notation, where several variables are given the same value at once.

This assignment	*Is equivalent to*

`a = b = c = 1`	`a = 1`
	`b = 1`
	`c = 1`

So-called multuple assignment provides a slightly shorter notation, where several variables are given their values at once.

This assignment	*Explanation*
`x, y, z = 99, 100, 'OK'`	`Results in:` `x=99, y= 100 and z='OK'`
`p, q, r = myFunc()`	`If myFunc() returns three values, p, q and r are assigned those three values.`

Python keywords

Like all programming languages, some words have defined meanings and are reserved for the Python interpreter. You must not use these words as variable names. Note that they are all lowercase.

and	**as**	**assert**	**break**
class	**continue**	**def**	**del**
elif	**else**	**except**	**exec**
finally	**for**	**from**	**global**
if	**import**	**in**	**is**
lambda	**not**	**or**	**pass**
print	**raise**	**return**	**try**
while	**with**	**yield**	

There are a large number of 'built-in' names that you must not use except for their intended purpose. The case of **`True`**, **`False`** and **`None`** are important. Common ones are listed below.

True	**False**	**None**	**abs**
all	**any**	**chr**	**dict**
dir	**eval**	**exit**	**file**

float	**format**	**input**	**int**
max	**min**	**next**	**object**
open	**print**	**quit**	**range**
round	**set**	**str**	**sum**
tuple	**type**	**vars**	**zip**

If you want to see a list of these built-ins, list the contents of the __builtins__ module in the shell like this:

```
>>> dir(__builtins__)
```

Special identifiers

Python also provides some special identifiers that use underscores. Their name will be of the form:

_xxx
__xxx__
__xxx

Mostly, you can ignore these[8]. However one that you might encounter in your programming is the special system variable:

__name__

This variable specifies how the module was called. **__name__** contains:

- The name of the module if imported
- The string '__main__' if executed directly.

You often see the code below at the bottom of modules. The interpreter loads your program and runs it if necessary.

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

[8] The only ones you need to know really are the **__name__** variable and the **__init__()** method called when a new object is created. Don't start your variable names with an underscore and you'll be fine.

Python Modules

Python code is usually stored in text files that are read by the Python interpreter at run time. Often, programs get so large that it makes sense to split large programs into smaller ones called modules. One module can be imported into others using the import statement.

```
import othermod     # makes the code in othermod
import mymodule     # and mymodule available
```

Typical Program Structure

The same program or module structure appears again and again, so you should try and follow it. In this way, you know what to expect from other programmers and they will know what to expect from you.

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

Used only in Linux/Unix environments (tells the shell where to find the Python program).

```
#
# this module does
# interesting things like
# calculate salaries
#
```

Modules should have some explanatory text describing or documenting its behaviour.

```
from datetime import datetime
```

Module imports come first so their content can be used later in the module.

```
now = datetime.now()
```

Create a global variable that is accessible to all classes and functions in the module.

```
class bookClass(object):
    "Book object"
    def __init__(self,title):
        self.title=title
        return


def testbook():
    "testing testing..."
    title="How to test Py"
    book=bookClass(title)
    print("Tested the book")

if   name  =='  main  ':
    testBook()
```

Class definitions appear first. Code that imports this module may then use these classes.

Functions are defined next. When imported, functions are accessed as module.function().

If imported, the module defines classes and functions. If this module is run, the code here (e.g. testBook()) is executed.