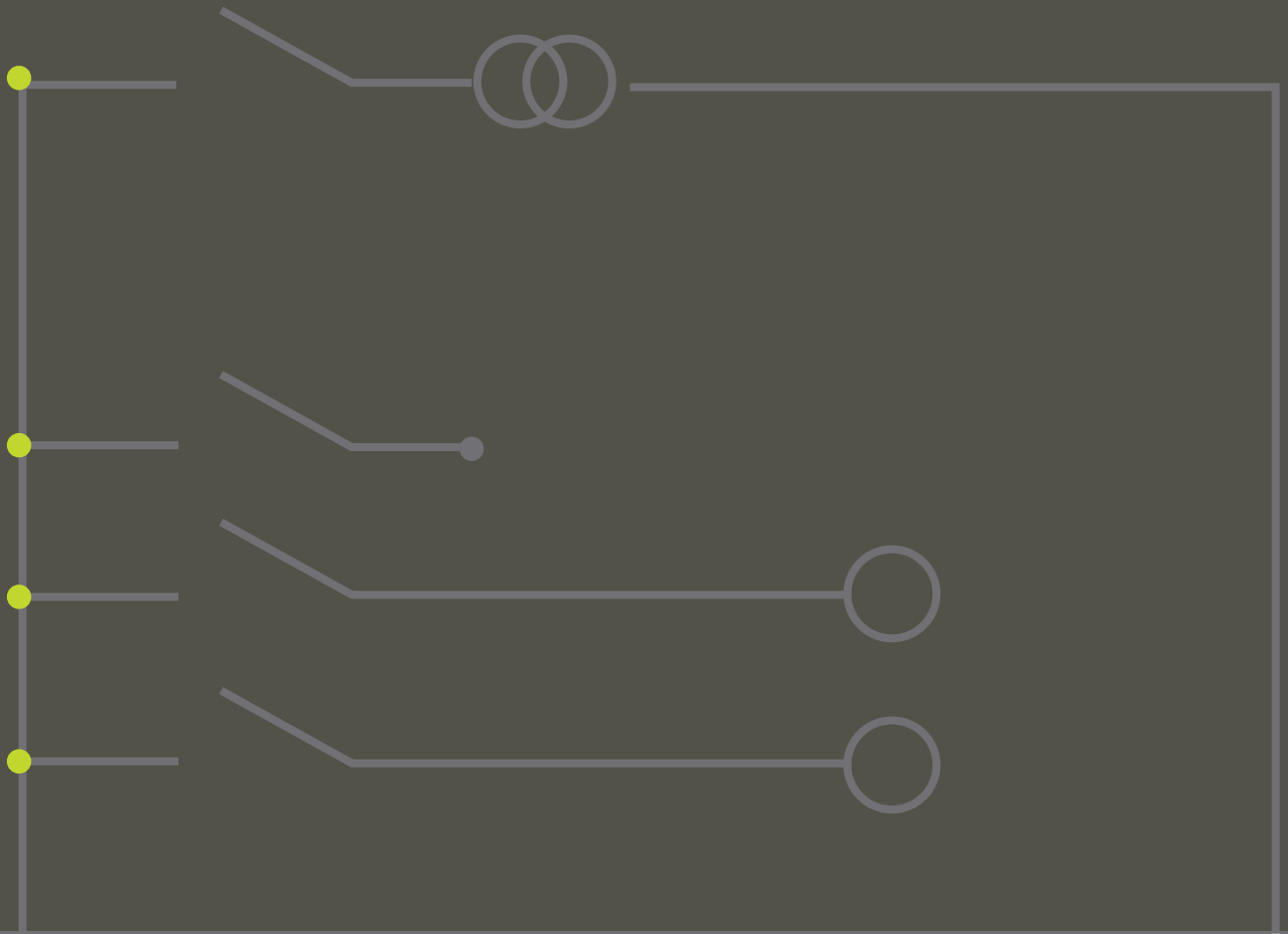




whit.

Python for Power Systems Engineers

Examples and reference material

Contents

1	Warming Up With Python	1
1.1	Interactive Mode ¹	1
1.2	An Informal Introduction to Python ²	1
1.2.1	Using Python as a Calculator	2
1.3	Example: Python's Data Types	10
1.3.1	Tasks	10
1.4	Sources	11
2	Controlling Python	13
2.1	While Loop ¹	13
2.2	More Control Flow Tools ²	14
2.2.1	if Statements	14
2.2.2	for Statements	15
2.2.3	The range() Function	15
2.2.4	break and continue Statements, and else Clauses on Loops	17
2.2.5	pass Statements	18
2.2.6	Defining Functions	18
2.2.7	More on Defining Functions	20
2.2.8	Intermezzo: Coding Style	26
2.3	The zip function	27
2.4	Example	29
2.4.1	Task 1	29
2.4.2	Task 2	29
2.5	Sources	30
3	Data Types	31
3.1	Data Structures ¹	31
3.1.1	More on Lists	31
3.1.2	The del Statement	36
3.1.3	Tuples and Sequences	36

3.1.4	Sets	37
3.1.5	Dictionaries	38
3.1.6	Looping Techniques	40
3.1.7	More on Conditions	41
3.1.8	Comparing Sequences and Other Types	42
3.2	Example: Using Lists and Dictionaries	43
3.2.1	Introduction	43
3.2.2	Task 1	43
3.2.3	Task 2	44
3.3	Sources	46
4	PSS[®]E	47
4.1	Initialising PSS [®] E From Python	47
4.1.1	About the PSS [®] E Subsystem Data Retrieval API	48
4.1.2	About Changing Power Flow Data	49
4.2	About the <code>matplotlib</code> API ¹	51
4.2.1	Your First <code>matplotlib</code> Chart ²	51
4.3	Example: Retrieving Data From PSS [®] E	53
4.3.1	Introduction	53
4.3.2	Task 1	53
4.3.3	Task 2	54
4.3.4	Task 3	54
4.4	Sources	55
5	Importing Modules & File Structure	57
5.1	Modules ¹	57
5.1.1	More on Modules	58
5.1.2	Standard Modules	61
5.1.3	The <code>dir()</code> Function	62
5.1.4	Packages	63
5.2	A Word on Scope	67
5.2.1	What's in a Scope	67
5.2.2	Copious Scopes	67
5.2.3	The Takeaway Concepts	70
5.3	Structuring Code	71
5.3.1	Coding Style	71
5.3.2	Documentation	72
5.3.3	Constants and Configurations	72
5.3.4	Functions	73

5.3.5	Creating Libraries	74
5.4	Sources	75
6	Input / Output	77
6.1	Input and Output ¹	77
6.1.1	Fancier Output Formatting	77
6.1.2	Reading and Writing Files	81
6.2	Example	85
6.2.1	Introduction	85
6.2.2	Task 1	85
6.3	Sources	87
7	When Things Go Wrong	89
7.1	Errors and Exceptions ¹	89
7.1.1	Syntax Errors	89
7.1.2	Exceptions	89
7.1.3	Handling Exceptions	90
7.1.4	Raising Exceptions	92
7.1.5	User-defined Exceptions	93
7.1.6	Defining Clean-up Actions	94
7.1.7	Predefined Clean-up Actions	95
7.2	Example	97
7.2.1	Task 1	98
7.2.2	Task 2	98
7.3	Sources	99
8	PSS[®]E Macros and Custom Toolbar Buttons	101
8.1	What is a macro button	101
8.2	The macro file structure	101
8.3	Adding the macro button to PSS [®] E	102
8.4	Example	103
8.4.1	Introduction	103
8.4.2	Task 1	103
8.4.3	Task 2	103
8.4.4	Task 3	104
9	Standard Library Modules	105
9.1	datetime	105
9.1.1	Creating time, date and datetime Objects	106

9.1.2	Date Arithmetic	110
9.1.3	Comparing Times and Dates	111
9.1.4	Formatted Output of <code>time</code> , <code>date</code> and <code>datetime</code> Objects	112
9.1.5	Reading Time From a String	112
9.2	<code>csv</code>	115
9.2.1	Introduction ²	115
9.2.2	<code>reader</code> Objects	115
9.2.3	<code>writer</code> Objects	117
9.2.4	Handling Headers	119
9.2.5	<code>dialect</code> Objects	120
9.2.6	<code>dialect</code> Properties	122
9.2.7	<code>csv</code> Errors ²	123
9.3	<code>os.path</code>	124
9.3.1	Dissecting Paths	124
9.3.2	Building Paths	125
9.3.3	Inspecting the Filesystem	126
9.4	<code>glob</code> ¹	127
9.5	Example	128
9.5.1	Introduction	128
9.5.2	Task 1	128
9.5.3	Task 2	128
9.5.4	Task 3	129
9.6	Sources	130
10	PSS[®]E and Exception Handling	131
10.1	The Python Debugger — <code>pdb</code>	131
10.2	PSS [®] E Exception Handling	133
10.3	Example: Find the Bug in the Haystack	134
10.3.1	Introduction	134
10.3.2	Task 1	134
11	Logging	135
11.1	A Simple Logging Example ¹	135
11.1.1	Logging to a File ¹	136
11.2	Example	137
11.2.1	Task 1	137
11.2.2	Task 2	138
11.3	Sources	139

12 Introducing the GUI	141
12.1 Ask the User for Input	141
12.1.1 Ask for Filename	141
12.1.2 Ask for Save File Location	142
12.1.3 Ask for Directory	143
12.1.4 Ask a Question	143
12.1.5 Ask About Anything	144
12.2 Example	149
12.2.1 Introduction	149
12.2.2 Task 1	149
12.2.3 Task 2	149
13 Send Your Data To Excel	151
13.1 Introducing excelpy	151
13.1.1 Create a New Workbook or Worksheet	152
13.1.2 Writing Data to Excel	153
13.1.3 Reading Data From Excel	154
13.1.4 Formatting Your Excel Files	155
13.2 Introducing pssexcel	155
13.3 Example	157
13.3.1 Introduction	157
13.3.2 Task 1	157
13.3.3 Task 2	157
13.3.4 Task 3	158
14 A QV Curve Generator	159
14.1 Adding Multiple Axes and Subplots to <code>matplotlib</code> Charts ¹	159
14.2 About the <code>qv_engine</code> API	161
14.2.1 Subsystem Description Data File	161
14.2.2 Monitored Element Data File	161
14.2.3 Contingency Description Data File	161
14.2.4 How to Setup <code>qv_engine()</code>	161
14.3 About the <code>pssarrays</code> Library	162
14.4 Example: Voltage Stability and Dealing With a Difficult API	164
14.4.1 Task 1	165
14.4.2 Task 2	165
14.4.3 Task 3	165
14.5 Sources	166

15 Automating your Dynamic Studies	167
15.1 About the Dynamics API	167
15.2 A Short Note About Preparing Dynamics Data	167
15.3 Loading a Dynamic Case and Snapshot	167
15.4 Running a dynamic study	168
15.4.1 Getting Dynamics Results	170
15.4.2 Putting it together	171
15.5 Example	173
15.5.1 Introduction	173
15.5.2 Task 1	173
15.5.3 Task 2	173
15.5.4 Task 3	174
15.5.5 Task 4	174

Chapter 1

Warming Up With Python

You Will Learn

- How to use Python as an interactive session and from a script.
- About the basic datatypes of Python: numbers, strings and lists.

1.1 Interactive Mode¹

When commands are read directly from a keyboard, the interpreter is said to be in *interactive mode*. In this mode it prompts for the next command with the *primary prompt*, usually three greater-than signs (`>>>`); for continuation lines it prompts with the *secondary prompt*, by default three dots (`...`). The interpreter prints a welcome message stating its version number and a copyright notice before printing the first prompt:

```
1 python3.3
2 Python 3.3 (default, Sep 24 2012, 09:25:04)
3 Type "help", "copyright", "credits" or "license" for more information.
4 >>>
```

Continuation lines are needed when entering a multi-line construct. As an example, take a look at this **if** statement:

```
1 >>> the_world_is_flat = 1
2 >>> if the_world_is_flat:
3 ...     print("Be careful not to fall off!")
4 ...
5 Be careful not to fall off!
```

1.2 An Informal Introduction to Python²

In the following examples, input and output are distinguished by the presence or absence of prompts (`>>>` and `...`): to repeat the example, you must type everything after the

prompt, when the prompt appears; lines that do not begin with a prompt are output from the interpreter. Note that a secondary prompt on a line by itself in an example means you must type a blank line; this is used to end a multi-line command.

Many of the examples in this manual, even those entered at the interactive prompt, include comments. Comments in Python start with the hash character, #, and extend to the end of the physical line. A comment may appear at the start of a line or following whitespace or code, but not within a string literal. A hash character within a string literal is just a hash character. Since comments are to clarify code and are not interpreted by Python, they may be omitted when typing in examples.

Some examples:

```
1 # this is the first comment
2 spam = 1 # and this is the second comment
3         # ... and now a third!
4 text = "# This is not a comment because it's inside quotes."
```

1.2.1 Using Python as a Calculator

Let's try some simple Python commands. Start the interpreter and wait for the primary prompt, >>>. (It shouldn't take long.)

1.2.1.1 Numbers

The interpreter acts as a simple calculator: you can type an expression at it and it will write the value. Expression syntax is straightforward: the operators +, -, * and / work just like in most other languages (for example, Pascal or C); parentheses (()) can be used for grouping. For example:

```
1 >>> 2 + 2
2 4
3 >>> 50 - 5*6
4 20
5 >>> (50 - 5*6) / 4
6 5.0
7 >>> 8 / 5 # division always returns a floating point number
8 1.6
```

The integer numbers (e.g. 2, 4, 20) have type **int**, the ones with a fractional part (e.g. 5.0, 1.6) have type **float**. We will see more about numeric types later in the tutorial.

Division (/) always returns a float. To do *floor division* and get an integer result (discarding any fractional result) you can use the // operator; to calculate the remainder you can use %:

```
1 >>> 17 / 3 # classic division returns a float
2 5.666666666666667
3 >>>
4 >>> 17 // 3 # floor division discards the fractional part
5 5
```

```
6 >>> 17 % 3 # the % operator returns the remainder of the division
7 2
8 >>> 5 * 3 + 2 # result * divisor + remainder
9 17
```

With Python, it is possible to use the ****** operator to calculate powers¹:

```
1 >>> 5 ** 2 # 5 squared
2 25
3 >>> 2 ** 7 # 2 to the power of 7
4 128
```

The equal sign (=) is used to assign a value to a variable. Afterwards, no result is displayed before the next interactive prompt:

```
1 >>> width = 20
2 >>> height = 5 * 9
3 >>> width * height
4 900
```

If a variable is not “defined” (assigned a value), trying to use it will give you an error:

```
1 >>> n # try to access an undefined variable
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 NameError: name 'n' is not defined
```

There is full support for floating point; operators with mixed type operands convert the integer operand to floating point:

```
1 >>> 3 * 3.75 / 1.5
2 7.5
3 >>> 7.0 / 2
4 3.5
```

In interactive mode, the last printed expression is assigned to the variable `_`. This means that when you are using Python as a desk calculator, it is somewhat easier to continue calculations, for example:

```
1 >>> tax = 12.5 / 100
2 >>> price = 100.50
3 >>> price * tax
4 12.5625
5 >>> price + _
6 113.0625
7 >>> round(_, 2)
8 113.06
```

¹Since ****** has higher precedence than **-**, `-3**2` will be interpreted as `-(3**2)` and thus result in `-9`. To avoid this and get `9`, you can use `(-3)**2`.

This variable should be treated as read-only by the user. Don't explicitly assign a value to it — you would create an independent local variable with the same name masking the built-in variable with its magic behavior.

In addition to **int** and **float**, Python supports other types of numbers, such as **Decimal** and **Fraction**. Python also has built-in support for *complex numbers*, and uses the **j** or **J** suffix to indicate the imaginary part (e.g. `3+5j`).

1.2.1.2 Strings

Besides numbers, Python can also manipulate strings, which can be expressed in several ways. They can be enclosed in single quotes (`'...'`) or double quotes (`"..."`) with the same result². `\` can be used to escape quotes:

```
1 >>> 'spam eggs' # single quotes
2 'spam eggs'
3 >>> 'doesn\'t' # use \' to escape the single quote...
4 "doesn't"
5 >>> "doesn't" # ...or use double quotes instead
6 "doesn't"
7 >>> '"Yes," he said.'
8 '"Yes," he said.'
9 >>> "\"Yes,\" he said."
10 '"Yes," he said.'
11 >>> '"Isn\'t," she said.'
12 '"Isn\'t," she said.'
```

In the interactive interpreter, the output string is enclosed in quotes and special characters are escaped with backslashes. While this might sometimes look different from the input (the enclosing quotes could change), the two strings are equivalent. The string is enclosed in double quotes if the string contains a single quote and no double quotes, otherwise it is enclosed in single quotes. The **print()** function produces a more readable output, by omitting the enclosing quotes and by printing escaped and special characters:

```
1 >>> '"Isn\'t," she said.'
2 '"Isn\'t," she said.'
3 >>> print('"Isn\'t," she said.')
4 "Isn't," she said.
5 >>> s = 'First line.\nSecond line.' # \n means newline
6 >>> s # without print(), \n is included in the output
7 'First line.\nSecond line.'
8 >>> print(s) # with print(), \n produces a new line
9 First line.
10 Second line.
```

If you don't want characters prefaced by `\` to be interpreted as special characters, you can use *raw strings* by adding an **r** before the first quote:

²Unlike other languages, special characters such as `\n` have the same meaning with both single (`'...'`) and double (`"..."`) quotes. The only difference between the two is that within single quotes you don't need to escape `"` (but you have to escape `\`) and vice versa.

```
1 >>> print('C:\some\name') # here \n means newline!
2 C:\some
3 ame
4 >>> print(r'C:\some\name') # note the r before the quote
5 C:\some\name
```

String literals can span multiple lines. One way is using triple-quotes: `"""..."""` or `'''...'''`. End of lines are automatically included in the string, but it's possible to prevent this by adding a `\` at the end of the line. The following example:

```
1 print("""\
2 Usage: thingy [OPTIONS]
3     -h                Display this usage message
4     -H hostname       Hostname to connect to
5 """)
```

produces the following output (note that the initial newline is not included):

```
1 Usage: thingy [OPTIONS]
2     -h                Display this usage message
3     -H hostname       Hostname to connect to
```

Strings can be concatenated (glued together) with the `+` operator, and repeated with `*`:

```
1 >>> # 3 times 'un', followed by 'ium'
2 >>> 3 * 'un' + 'ium'
3 'unununium'
```

Two or more *string literals* (i.e. the ones enclosed between quotes) next to each other are automatically concatenated.

```
1 >>> 'Py' 'thon'
2 'Python'
```

This only works with two literals though, not with variables or expressions:

```
1 >>> prefix = 'Py'
2 >>> prefix 'thon' # can't concatenate a variable and a string literal
3 ...
4 SyntaxError: invalid syntax
5 >>> ('un' * 3) 'ium'
6 ...
7 SyntaxError: invalid syntax
```

If you want to concatenate variables or a variable and a literal, use `+`:

1. WARMING UP WITH PYTHON

```
1 >>> prefix + 'thon'
2 'Python'
```

This feature is particularly useful when you want to break long strings:

```
1 >>> text = ('Put several strings within parentheses '
2           'to have them joined together.')
3 >>> text
4 'Put several strings within parentheses to have them joined together.'
```

Strings can be *indexed* (subscripted), with the first character having index 0. There is no separate character type; a character is simply a string of size one:

```
1 >>> word = 'Python'
2 >>> word[0] # character in position 0
3 'P'
4 >>> word[5] # character in position 5
5 'n'
```

Indices may also be negative numbers, to start counting from the right:

```
1 >>> word[-1] # last character
2 'n'
3 >>> word[-2] # second-last character
4 'o'
5 >>> word[-6]
6 'P'
```

Note that since -0 is the same as 0, negative indices start from -1.

In addition to indexing, *slicing* is also supported. While indexing is used to obtain individual characters, *slicing* allows you to obtain substring:

```
1 >>> word[0:2] # characters from position 0 (included) to 2 (excluded)
2 'Py'
3 >>> word[2:5] # characters from position 2 (included) to 5 (excluded)
4 'tho'
```

Note how the start is always included, and the end always excluded. This makes sure that `s[:i] + s[i:]` is always equal to `s`:

```
1 >>> word[:2] + word[2:]
2 'Python'
3 >>> word[:4] + word[4:]
4 'Python'
```

Slice indices have useful defaults; an omitted first index defaults to zero, an omitted second index defaults to the size of the string being sliced.

```
1 >>> word[:2] # character from the beginning to position 2 (excluded)
2 'Py'
3 >>> word[4:] # characters from position 4 (included) to the end
4 'on'
5 >>> word[-2:] # characters from the second-last (included) to the end
6 'on'
```

One way to remember how slices work is to think of the indices as pointing *between* characters, with the left edge of the first character numbered 0. Then the right edge of the last character of a string of n characters has index n , for example:

```
1 +---+---+---+---+---+
2 | P | y | t | h | o | n |
3 +---+---+---+---+---+
4 0   1   2   3   4   5   6
5 -6  -5  -4  -3  -2  -1
```

The first row of numbers gives the position of the indices 0...6 in the string; the second row gives the corresponding negative indices. The slice from i to j consists of all characters between the edges labeled i and j , respectively.

For non-negative indices, the length of a slice is the difference of the indices, if both are within bounds. For example, the length of `word[1:3]` is 2.

Attempting to use an index that is too large will result in an error:

```
1 >>> word[42] # the word only has 7 characters
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in <module>
4 IndexError: string index out of range
```

However, out of range slice indexes are handled gracefully when used for slicing:

```
1 >>> word[4:42]
2 'on'
3 >>> word[42:]
4 ''
```

Python strings cannot be changed — they are *immutable*. Therefore, assigning to an indexed position in the string results in an error:

```
1 >>> word[0] = 'J'
2 ...
3 TypeError: 'str' object does not support item assignment
4 >>> word[2:] = 'py'
5 ...
6 TypeError: 'str' object does not support item assignment
```

If you need a different string, you should create a new one:

```
1 >>> 'J' + word[1:]
2 'Jython'
3 >>> word[:2] + 'py'
4 'Pypy'
```

The built-in function `len()` returns the length of a string:

```
1 >>> s = 'supercalifragilisticexpialidocious'
2 >>> len(s)
3 34
```

1.2.1.3 Lists

Python knows a number of *compound* data types, used to group together other values. The most versatile is the *list*, which can be written as a list of comma-separated values (items) between square brackets. Lists might contain items of different types, but usually the items all have the same type.

```
1 >>> squares = [1, 2, 4, 9, 16, 25]
2 >>> squares
3 [1, 2, 4, 9, 16, 25]
```

Like strings (and all other built-in *sequence* type), lists can be indexed and sliced:

```
1 >>> squares[0] # indexing returns the item
2 1
3 >>> squares[-1]
4 25
5 >>> squares[-3:] # slicing returns a new list
6 [9, 16, 25]
```

All slice operations return a new list containing the requested elements. This means that the following slice returns a new (shallow) copy of the list:

```
1 >>> squares[:]
2 [1, 2, 4, 9, 16, 25]
```

Lists also supports operations like concatenation:

```
1 >>> squares + [36, 49, 64, 81, 100]
2 [1, 2, 4, 9, 16, 25, 36, 49, 64, 81, 100]
```

Unlike strings, which are *immutable*, lists are a *mutable* type, i.e. it is possible to change their content:

```
1 >>> cubes = [1, 8, 27, 65, 125] # something's wrong here
2 >>> 4 ** 3 # the cube of 4 is 64, not 65!
3 64
4 >>> cubes[3] = 64 # replace the wrong value
5 >>> cubes
6 [1, 8, 27, 64, 125]
```

You can also add new items at the end of the list, by using the **append()** *method* (we will see more about methods later):

```
1 >>> cubes.append(216) # add the cube of 6
2 >>> cubes.append(7 ** 3) # and the cube of 7
3 >>> cubes
4 [1, 8, 27, 64, 125, 216, 343]
```

Assignment to slices is also possible, and this can even change the size of the list or clear it entirely:

```
1 >>> letters = ['a', 'b', 'c', 'd', 'e', 'f', 'g']
2 >>> letters
3 ['a', 'b', 'c', 'd', 'e', 'f', 'g']
4 >>> # replace some values
5 >>> letters[2:5] = ['C', 'D', 'E']
6 >>> letters
7 ['a', 'b', 'C', 'D', 'E', 'f', 'g']
8 >>> # now remove them
9 >>> letters[2:5] = []
10 >>> letters
11 ['a', 'b', 'f', 'g']
12 >>> # clear the list by replacing all the elements with an empty list
13 >>> letters[:] = []
14 >>> letters
15 []
```

The built-in function **len()** also applies to lists:

```
1 >>> letters = ['a', 'b', 'c', 'd']
2 >>> len(letters)
3 4
```

It is possible to nest lists (create lists containing other lists), for example:

```
1 >>> a = ['a', 'b', 'c']
2 >>> n = [1, 2, 3]
3 >>> x = [a, n]
4 >>> x
5 [['a', 'b', 'c'], [1, 2, 3]]
6 >>> x[0]
7 ['a', 'b', 'c']
8 >>> x[0][1]
9 'b'
```

1.3 Example: Python's Data Types

A quick interactive session with Python, designed to refresh your memory and give you some familiarity with the Python interactive mode. We will be using the interactive mode extensively throughout this course.

1.3.1 Tasks

- Open up an interactive Python shell.
- Enter the following commands and predict what the outcomes will be.
- Did the outputs match your expectations?

What is the output of:

```
1 >>> 1 / 0
2 >>> 1 / 2
3 >>> 1.0 / 2
```

```
1 >>> a = 'apple'
2 >>> len(a)
3 >>> a[:5]
4 >>> a[:10]
```

```
1 >>> b = range(10,20)
2 >>> len(b)
3 >>> b[0::2]
4 >>> b[0:4]
5 >>> b[:4]
6 >>> b[4:]
7 >>> b[-1]
8 >>> b[-2]
```

```
1 >>> comp = complex(1,2)
2 >>> comp.real
```

1.4 Sources

- [1] Python Foundation. Official Python Tutorial, Chapter 2.1.2, 2011. <http://docs.python.org/3.3/tutorial/interpreter.html#interactive-mode>.
- [2] Python Foundation. Official Python Tutorial, Chapter 3, 2011. <http://docs.python.org/3.3/tutorial/introduction.html>.

Chapter 2

Controlling Python

You Will Learn

- Writing loops;
- Defining and calling functions.

2.1 While Loop¹

Of course, we can use Python for more complicated tasks than adding two and two together. For instance, we can write an initial sub-sequence of the *Fibonacci* series as follows:

```
1 >>> # Fibonacci series:
2 ... # the sum of two elements defines the next
3 ... a, b = 0, 1
4 >>> while b < 10:
5 ...     print(b)
6 ...     a, b = b, a+b
7 ...
8 1
9 1
10 2
11 3
12 5
13 8
```

This example introduces several new features.

- The first line contains a *multiple assignment*: the variables `a` and `b` simultaneously get the new values 0 and 1. On the last line this is used again, demonstrating that the expressions on the right-hand side are all evaluated first before any of the assignments take place. The right-hand side expressions are evaluated from the left to the right.
- The `while` loop executes as long as the condition (here: `b < 10`) remains true. In Python, like in C, any non-zero integer value is true; zero is false. The condition

may also be a string or list value, in fact any sequence; anything with a non-zero length is true, empty sequences are false. The test used in the example is a simple comparison. The standard comparison operators are written the same as in C: < (less than), > (greater than), == (equal to), <= (less than or equal to), >= (greater than or equal to) and != (not equal to).

- The *body* of the loop is *indented*: indentation is Python's way of grouping statements. At the interactive prompt, you have to type a tab or space(s) for each indented line. In practice you will prepare more complicated input for Python with a text editor; all decent text editors have an auto-indent facility. When a compound statement is entered interactively, it must be followed by a blank line to indicate completion (since the parser cannot guess when you have typed the last line). Note that each line within a basic block must be indented by the same amount.
- The **print()** function writes the value of the argument(s) it is given. It differs from just writing the expression you want to write (as we did earlier in the calculator examples) in the way it handles multiple arguments, floating point quantities, and strings. Strings are printed without quotes, and a space is inserted between items, so you can format things nicely, like this:

```
1 >>> i = 256*256
2 >>> print('The value of i is', i)
3 The value of i is 65536
```

The keyword argument *end* can be used to avoid the newline after the output, or end the output with a different string:

```
1 >>> a, b = 0, 1
2 >>> while b < 1000:
3 ...     print(b, end=',')
4 ...     a, b = b, a+b
5 ...
6 1,1,2,3,5,8,13,21,34,55,89,144,233,377,610,987,
```

2.2 More Control Flow Tools²

Besides the **while** statement just introduced, Python knows the usual control flow statements known from other languages, with some twists.

2.2.1 if Statements

Perhaps the most well-known statement type is the **if** statement. For example:

```
1 >>> x = int(input("Please enter an integer: "))
2 Please enter an integer: 42
3 >>> if x < 0:
4 ...     x = 0
```

```
5 ...     print('Negative changed to zero')
6 ... elif x == 0:
7 ...     print('Zero')
8 ... elif x == 1:
9 ...     print('Single')
10 ... else:
11 ...     print('More')
12 ...
13 More
```

There can be zero or more **elif** parts, and the **else** part is optional. The keyword **elif** is short for ‘else if’, and is useful to avoid excessive indentation. An **if ... elif ... elif ...** sequence is a substitute for the **switch** or **case** statements found in other languages.

2.2.2 for Statements

The **for** statement in Python differs a bit from what you may be used to in C or Pascal. Rather than always iterating over an arithmetic progression of numbers (like in Pascal), or giving the user the ability to define both the iteration step and halting condition (as C), Python’s **for** statement iterates over the items of any sequence (a list or a string), in the order that they appear in the sequence. For example (no pun intended):

```
1 >>> # Measure some strings:
2 ... words = ['cat', 'window', 'defenestrate']
3 >>> for w in words:
4 ...     print(w, len(w))
5 ...
6 cat 3
7 window 6
8 defenestrate 12
```

If you need to modify the sequence you are iterating over while inside the loop (for example to duplicate selected items), it is recommended that you first make a copy. Iterating over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```
1 >>> for w in words[:]: # Loop over a slice copy of the entire list.
2 ...     if len(w) > 6:
3 ...         words.insert(0, w)
4 ...
5 >>> words
6 ['defenestrate', 'cat', 'window', 'defenestrate']
```

2.2.3 The range() Function

If you do need to iterate over a sequence of numbers, the built-in function **range()** comes in handy. It generates arithmetic progressions:

```
1 >>> for i in range(5):
2 ...     print(i)
3 ...
4 0
5 1
6 2
7 3
8 4
```

The given end point is never part of the generated sequence; `range(10)` generates 10 values, the legal indices for items of a sequence of length 10. It is possible to let the range start at another number, or to specify a different increment (even negative; sometimes this is called the ‘step’):

```
1 range(5, 10)
2     5 through 9
3
4 range(0, 10, 3)
5     0, 3, 6, 9
6
7 range(-10, -100, -30)
8    -10, -40, -70
```

To iterate over the indices of a sequence, you can combine `range()` and `len()` as follows:

```
1 >>> a = ['Mary', 'had', 'a', 'little', 'lamb']
2 >>> for i in range(len(a)):
3 ...     print(i, a[i])
4 ...
5 0 Mary
6 1 had
7 2 a
8 3 little
9 4 lamb
```

In most such cases, however, it is convenient to use the `enumerate()` function, see *Looping Techniques*.

A strange thing happens if you just print a range:

```
1 >>> print(range(10))
2 range(0, 10)
```

In many ways the object returned by `range()` behaves as if it is a list, but in fact it isn’t. It is an object which returns the successive items of the desired sequence when you iterate over it, but it doesn’t really make the list, thus saving space.

We say such an object is *iterable*, that is, suitable as a target for functions and constructs that expect something from which they can obtain successive items until the supply is exhausted. We have seen that the `for` statement is such an *iterator*. The function `list()` is another; it creates lists from iterables:

```
1 >>> list(range(5))
2 [0, 1, 2, 3, 4]
```

Later we will see more functions that return iterables and take iterables as argument.

2.2.4 **break** and **continue** Statements, and **else** Clauses on Loops

The **break** statement, like in C, breaks out of the smallest enclosing **for** or **while** loop.

Loop statements may have an **else** clause; it is executed when the loop terminates through exhaustion of the list (with **for**) or when the condition becomes false (with **while**), but not when the loop is terminated by a **break** statement. This is exemplified by the following loop, which searches for prime numbers:

```
1 >>> for n in range(2, 10):
2 ...     for x in range(2, n):
3 ...         if n % x == 0:
4 ...             print(n, 'equals', x, '*', n//x)
5 ...             break
6 ...         else:
7 ...             # loop fell through without finding a factor
8 ...             print(n, 'is a prime number')
9 ...
10 2 is a prime number
11 3 is a prime number
12 4 equals 2 * 2
13 5 is a prime number
14 6 equals 2 * 3
15 7 is a prime number
16 8 equals 2 * 4
17 9 equals 3 * 3
```

(Yes, this is the correct code. Look closely: the **else** clause belongs to the **for** loop, **not** the **if** statement.)

When used with a loop, the **else** clause has more in common with the **else** clause of a **try** statement than it does that of **if** statements: a **try** statement's **else** clause runs when no exception occurs, and a loop's **else** clause runs when no **break** occurs. For more on the **try** statement and exceptions, see *Handling Exceptions*.

The **continue** statement, also borrowed from C, continues with the next iteration of the loop:

```
1 >>> for num in range(2, 10):
2 ...     if num % 2 == 0:
3 ...         print("Found an even number", num)
4 ...         continue
5 ...     print("Found a number", num)
6 Found an even number 2
7 Found a number 3
8 Found an even number 4
9 Found a number 5
10 Found an even number 6
```

```
11 Found a number 7
12 Found an even number 8
13 Found a number 9
```

2.2.5 `pass` Statements

The **`pass`** statement does nothing. It can be used when a statement is required syntactically but the program requires no action. For example:

```
1 >>> while True:
2 ...     pass # Busy-wait for keyboard interrupt (Ctrl+C)
3 ...
```

This is commonly used for creating minimal classes:

```
1 >>> class MyEmptyClass:
2 ...     pass
3 ...
```

Another place **`pass`** can be used is as a place-holder for a function or conditional body when you are working on new code, allowing you to keep thinking at a more abstract level. The **`pass`** is silently ignored:

```
1 >>> def initlog(*args):
2 ...     pass # Remember to implement this!
3 ...
```

2.2.6 Defining Functions

We can create a function that writes the Fibonacci series to an arbitrary boundary:

```
1 >>> def fib(n): # write Fibonacci series up to n
2 ...     """Print a Fibonacci series up to n."""
3 ...     a, b = 0, 1
4 ...     while a < n:
5 ...         print(a, end=' ')
6 ...         a, b = b, a+b
7 ...     print()
8 ...
9 >>> # Now call the function we just defined:
10 ... fib(2000)
11 0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987 1597
```

The keyword **`def`** introduces a function *definition*. It must be followed by the function name and the parenthesized list of formal parameters. The statements that form the body of the function start at the next line, and must be indented.

The first statement of the function body can optionally be a string literal; this string literal is the function's documentation string, or *docstring*. (More about docstrings can

be found in the section *Documentation Strings*.) There are tools which use docstrings to automatically produce online or printed documentation, or to let the user interactively browse through code; it's good practice to include docstrings in code that you write, so make a habit of it.

The *execution* of a function introduces a new symbol table used for the local variables of the function. More precisely, all variable assignments in a function store the value in the local symbol table; whereas variable references first look in the local symbol table, then in the local symbol tables of enclosing functions, then in the global symbol table, and finally in the table of built-in names. Thus, global variables cannot be directly assigned a value within a function (unless named in a **global** statement), although they may be referenced.

The actual parameters (arguments) to a function call are introduced in the local symbol table of the called function when it is called; thus, arguments are passed using *call by value* (where the *value* is always an object *reference*, not the value of the object).¹ When a function calls another function, a new local symbol table is created for that call.

A function definition introduces the function name in the current symbol table. The value of the function name has a type that is recognized by the interpreter as a user-defined function. This value can be assigned to another name which can then also be used as a function. This serves as a general renaming mechanism:

```
1 >>> fib
2 <function fib at 10042ed0>
3 >>> f = fib
4 >>> f(100)
5 0 1 1 2 3 5 8 13 21 34 55 89
```

Coming from other languages, you might object that `fib` is not a function but a procedure since it doesn't return a value. In fact, even functions without a **return** statement do return a value, albeit a rather boring one. This value is called `None` (it's a built-in name). Writing the value `None` is normally suppressed by the interpreter if it would be the only value written. You can see it if you really want to using **print()**:

```
1 >>> fib(0)
2 >>> print(fib(0))
3 None
```

It is simple to write a function that returns a list of the numbers of the Fibonacci series, instead of printing it:

```
1 >>> def fib2(n): # return Fibonacci series up to n
2 ...     """Return a list containing the Fibonacci series up to n."""
3 ...     result = []
4 ...     a, b = 0, 1
5 ...     while a < n:
```

¹Actually, *call by object reference* would be a better description, since if a mutable object is passed, the caller will see any changes the callee makes to it (items inserted into a list).

```
6 ...         result.append(a)      # see below
7 ...         a, b = b, a+b
8 ...         return result
9 ...
10 >>> f100 = fib2(100)      # call it
11 >>> f100                  # write the result
12 [0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
```

This example, as usual, demonstrates some new Python features:

- The **return** statement returns with a value from a function. **return** without an expression argument returns **None**. Falling off the end of a function also returns **None**.
- The statement **result.append(a)** calls a *method* of the list object **result**. A method is a function that ‘belongs’ to an object and is named **obj.methodname**, where **obj** is some object (this may be an expression), and **methodname** is the name of a method that is defined by the object’s type. Different types define different methods. Methods of different types may have the same name without causing ambiguity. (It is possible to define your own object types and methods, using *classes*, see *Classes*) The method **append()** shown in the example is defined for list objects; it adds a new element at the end of the list. In this example it is equivalent to **result = result + [a]**, but more efficient.

2.2.7 More on Defining Functions

It is also possible to define functions with a variable number of arguments. There are three forms, which can be combined.

2.2.7.1 Default Argument Values

The most useful form is to specify a default value for one or more arguments. This creates a function that can be called with fewer arguments than it is defined to allow. For example:

```
1 def ask_ok(prompt, retries=4, complaint='Yes or no, please!'):
2     while True:
3         ok = input(prompt)
4         if ok in ('y', 'ye', 'yes'):
5             return True
6         if ok in ('n', 'no', 'nop', 'nope'):
7             return False
8         retries = retries - 1
9         if retries < 0:
10            raise IOError('refusenik user')
11    print(complaint)
```

This function can be called in several ways:

- giving only the mandatory argument:
`ask_ok('Do you really want to quit?')`

- giving one of the optional arguments:
`ask_ok('OK to overwrite the file?', 2)`
- or even giving all arguments:
`ask_ok('OK to overwrite the file?', 2, 'Come on, only yes or no!')`

This example also introduces the **in** keyword. This tests whether or not a sequence contains a certain value.

The default values are evaluated at the point of function definition in the *defining* scope, so that

```
1 i = 5
2
3 def f(arg=i):
4     print(arg)
5
6 i = 6
7 f()
```

will print 5.

Important warning: The default value is evaluated only once. This makes a difference when the default is a mutable object such as a list, dictionary, or instances of most classes. For example, the following function accumulates the arguments passed to it on subsequent calls:

```
1 def f(a, L=[]):
2     L.append(a)
3     return L
4
5 print(f(1))
6 print(f(2))
7 print(f(3))
```

This will print

```
1 [1]
2 [1, 2]
3 [1, 2, 3]
```

If you don't want the default to be shared between subsequent calls, you can write the function like this instead:

```
1 def f(a, L=None):
2     if L is None:
3         L = []
4     L.append(a)
5     return L
```

2.2.7.2 Keyword Arguments

Functions can also be called using *keyword arguments* of the form `kwarg=value`. For instance, the following function:

```
1 def parrot(voltage, state='a stiff', action='vroom', type='Norwegian Blue'):
2     print("-- This parrot wouldn't", action, end=' ')
3     print("if you put", voltage, "volts through it.")
4     print("-- Lovely plumage, the", type)
5     print("-- It's", state, "!")
```

accepts one required argument (`voltage`) and three optional arguments (`state`, `action`, and `type`). This function can be called in any of the following ways:

```
1 parrot(1000)                                # 1 positional argument
2 parrot(voltage=1000)                         # 1 keyword argument
3 parrot(voltage=1000000, action='VOOOOOM')    # 2 keyword arguments
4 parrot(action='VOOOOOM', voltage=1000000)    # 2 keyword arguments
5 parrot('a million', 'bereft of life', 'jump') # 3 positional arguments
6 parrot('a thousand', state='pushing up the daisies') # 1 positional, 1 keyword
```

but all the following calls would be invalid:

```
1 parrot()                                # required argument missing
2 parrot(voltage=5.0, 'dead')             # non-keyword argument after a keyword argument
3 parrot(110, voltage=220)                 # duplicate value for the same argument
4 parrot(actor='John Cleese')              # unknown keyword argument
```

In a function call, keyword arguments must follow positional arguments. All the keyword arguments passed must match one of the arguments accepted by the function (e.g. `actor` is not a valid argument for the `parrot` function), and their order is not important. This also includes non-optional arguments (e.g. `parrot(voltage=1000)` is valid too). No argument may receive a value more than once. Here's an example that fails due to this restriction:

```
1 >>> def function(a):
2     ...     pass
3     ...
4 >>> function(0, a=0)
5 Traceback (most recent call last):
6   File "<stdin>", line 1, in ?
7 TypeError: function() got multiple values for keyword argument 'a'
```

When a final formal parameter of the form `**name` is present, it receives a dictionary (see *Mapping Types — dict*) containing all keyword arguments except for those corresponding to a formal parameter. This may be combined with a formal parameter of the form `*name` (described in the next subsection) which receives a tuple containing the positional arguments beyond the formal parameter list. (`*name` must occur before `**name`.) For example, if we define a function like this:

```
1 def cheeseshop(kind, *arguments, **keywords):
2     print("-- Do you have any", kind, "?")
3     print("-- I'm sorry, we're all out of", kind)
4     for arg in arguments:
5         print(arg)
6     print("-" * 40)
7     keys = sorted(keywords.keys())
8     for kw in keys:
9         print(kw, ":", keywords[kw])
```

It could be called like this:

```
1 cheeseshop("Limburger", "It's very runny, sir.",
2           "It's really very, VERY runny, sir.",
3           shopkeeper="Michael Palin",
4           client="John Cleese",
5           sketch="Cheese Shop Sketch")
```

and of course it would print:

```
1 -- Do you have any Limburger ?
2 -- I'm sorry, we're all out of Limburger
3 It's very runny, sir.
4 It's really very, VERY runny, sir.
5 -----
6 client : John Cleese
7 shopkeeper : Michael Palin
8 sketch : Cheese Shop Sketch
```

Note that the list of keyword argument names is created by sorting the result of the keywords dictionary's `keys()` method before printing its contents; if this is not done, the order in which the arguments are printed is undefined.

2.2.7.3 Arbitrary Argument Lists

Finally, the least frequently used option is to specify that a function can be called with an arbitrary number of arguments. These arguments will be wrapped up in a tuple (see *Tuples and Sequences*). Before the variable number of arguments, zero or more normal arguments may occur.

```
1 def write_multiple_items(file, separator, *args):
2     file.write(separator.join(args))
```

Normally, these *variadic* arguments will be last in the list of formal parameters, because they scoop up all remaining input arguments that are passed to the function. Any formal parameters which occur after the `*args` parameter are ‘keyword-only’ arguments, meaning that they can only be used as keywords rather than positional arguments.

```
1 >>> def concat(*args, sep="/"):
2 ...     return sep.join(args)
3 ...
4 >>> concat("earth", "mars", "venus")
5 'earth/mars/venus'
6 >>> concat("earth", "mars", "venus", sep=".")
7 'earth.mars.venus'
```

2.2.7.4 Unpacking Argument Lists

The reverse situation occurs when the arguments are already in a list or tuple but need to be unpacked for a function call requiring separate positional arguments. For instance, the built-in `range()` function expects separate *start* and *stop* arguments. If they are not available separately, write the function call with the `*`-operator to unpack the arguments out of a list or tuple:

```
1 >>> list(range(3, 6))           # normal call with separate arguments
2 [3, 4, 5]
3 >>> args = [3, 6]
4 >>> list(range(*args))         # call with arguments unpacked from a list
5 [3, 4, 5]
```

In the same fashion, dictionaries can deliver keyword arguments with the `**`-operator:

```
1 >>> def parrot(voltage, state='a stiff', action='voom'):
2 ...     print("-- This parrot wouldn't", action, end=' ')
3 ...     print("if you put", voltage, "volts through it.", end=' ')
4 ...     print("E's", state, "!")
5 ...
6 >>> d = {"voltage": "four million", "state": "bleedin' demised", "action": "VOOM"}
7 >>> parrot(**d)
8 -- This parrot wouldn't VOOM if you put four million volts through it. E's bleedin' demised !
```

2.2.7.5 Lambda Expressions

Small anonymous functions can be created with the `lambda` keyword. This function returns the sum of its two arguments: `lambda a, b: a+b`. Lambda functions can be used wherever function objects are required. They are syntactically restricted to a single expression. Semantically, they are just syntactic sugar for a normal function definition. Like nested function definitions, lambda functions can reference variables from the containing scope:

```
1 >>> def make_incrementor(n):
2 ...     return lambda x: x + n
3 ...
4 >>> f = make_incrementor(42)
5 >>> f(0)
6 42
7 >>> f(1)
8 43
```

The above example uses a lambda expression to return a function. Another use is to pass a small function as an argument:

```
1 >>> pairs = [(1, 'one'), (2, 'two'), (3, 'three'), (4, 'four')]
2 >>> pairs.sort(key=lambda pair: pair[1])
3 >>> pairs
4 [(4, 'four'), (1, 'one'), (3, 'three'), (2, 'two')]
```

2.2.7.6 Documentation Strings

Here are some conventions about the content and formatting of documentation strings.

The first line should always be a short, concise summary of the object's purpose. For brevity, it should not explicitly state the object's name or type, since these are available by other means (except if the name happens to be a verb describing a function's operation). This line should begin with a capital letter and end with a period.

If there are more lines in the documentation string, the second line should be blank, visually separating the summary from the rest of the description. The following lines should be one or more paragraphs describing the object's calling conventions, its side effects, etc.

The Python parser does not strip indentation from multi-line string literals in Python, so tools that process documentation have to strip indentation if desired. This is done using the following convention. The first non-blank line *after* the first line of the string determines the amount of indentation for the entire documentation string. (We can't use the first line since it is generally adjacent to the string's opening quotes so its indentation is not apparent in the string literal.) Whitespace "equivalent" to this indentation is then stripped from the start of all lines of the string. Lines that are indented less should not occur, but if they occur all their leading whitespace should be stripped. Equivalence of whitespace should be tested after expansion of tabs (to 8 spaces, normally).

Here is an example of a multi-line docstring:

```
1 >>> def my_function():
2 ...     """Do nothing, but document it.
3 ...
4 ...     No, really, it doesn't do anything.
5 ...     """
6 ...     pass
7 ...
8 >>> print(my_function.__doc__)
9 Do nothing, but document it.
10
11     No, really, it doesn't do anything.
```

2.2.7.7 Function Annotations

Function annotations are completely optional, arbitrary metadata information about user-defined functions. Neither Python itself nor the standard library use function annotations in any way; this section just shows the syntax. Third-party projects are free to use function annotations for documentation, type checking, and other uses.

Annotations are stored in the `__annotations__` attribute of the function as a dictionary and have no effect on any other part of the function. Parameter annotations are defined by a colon after the parameter name, followed by an expression evaluating to the value of the annotation. Return annotations are defined by a literal `->`, followed by an expression, between the parameter list and the colon denoting the end of the `def` statement. The following example has a positional argument, a keyword argument, and the return value annotated with nonsense:

```
1 >>> def f(ham: 42, eggs: int = 'spam') -> "Nothing to see here":
2 ...     print("Annotations:", f.__annotations__)
3 ...     print("Arguments:", ham, eggs)
4 ...
5 >>> f('wonderful')
6 Annotations: {'eggs': <class 'int'>, 'return': 'Nothing to see here', 'ham': 42}
7 Arguments: wonderful spam
```

2.2.8 Intermezzo: Coding Style

Now that you are about to write longer, more complex pieces of Python, it is a good time to talk about *coding style*. Most languages can be written (or more concise, *formatted*) in different styles; some are more readable than others. Making it easy for others to read your code is always a good idea, and adopting a nice coding style helps tremendously for that.

For Python, **PEP 8** has emerged as the style guide that most projects adhere to; it promotes a very readable and eye-pleasing coding style. Every Python developer should read it at some point; here are the most important points extracted for you:

- Use 4-space indentation, and no tabs.
4 spaces are a good compromise between small indentation (allows greater nesting depth) and large indentation (easier to read). Tabs introduce confusion, and are best left out.
- Wrap lines so that they don't exceed 79 characters.
This helps users with small displays and makes it possible to have several code files side-by-side on larger displays.
- Use blank lines to separate functions and classes, and larger blocks of code inside functions.
- When possible, put comments on a line of their own.
- Use docstrings.
- Use spaces around operators and after commas, but not directly inside bracketing constructs: `a = f(1, 2) + g(3, 4)`.
- Name your classes and functions consistently; the convention is to use **CamelCase** for classes and **lower_case_with_underscores** for functions and methods. Always use `self` as the name for the first method argument (see *A First Look at Classes* for more on classes and methods).

- Don't use fancy encodings if your code is meant to be used in international environments. Python's default, UTF-8, or even plain ASCII work best in any case.
- Likewise, don't use non-ASCII characters in identifiers if there is only the slightest chance people speaking a different language will read or maintain the code.

2.3 The zip function

The `zip()` function provides a convenient way of grouping elements of separate lists together. `zip()` takes an arbitrary number of lists and returns an object containing tuples where the i^{th} element from each list is grouped into i^{th} tuple of the new object.

For instance, say we have three lists, `names`, `gender`, `age`, and we want to group a person's name, gender and age into one entry per person. The benefit of arranging the data like this is that we can now loop over this new object and operate on all the data for a particular person without having to do any list indexing.

```
1 >>> names = ['Brad', 'Sue', 'Mark', 'Jan']
2 >>> gender = ['M', 'F', 'M', 'F']
3 >>> age = [20, 22, 49, 32]
4 >>>
5 >>> details = zip(names, gender, age)
6 >>> print(list(details))
7 [('Brad', 'M', 20), ('Sue', 'F', 22), ('Mark', 'M', 49), ('Jan', 'F', 32)]
8 >>>
9 >>> details = zip(names, gender, age) # a zip object is a one use object
10 >>> for detail in details:
11 ...     print(detail)
12 ...
13 ('Brad', 'M', 20)
14 ('Sue', 'F', 22)
15 ('Mark', 'M', 49)
16 ('Jan', 'F', 32)
```

It is possible to combine the `zip()` call into the `for` statement to produce the shorter code:

```
1 >>> for detail in zip(names, gender, age):
2 ...     print(detail)
3 ...
4 ('Brad', 'M', 20)
5 ('Sue', 'F', 22)
6 ('Mark', 'M', 49)
7 ('Jan', 'F', 32)
```

Furthermore, we can use tuple unpacking in the `for` statement to break the elements of the tuple into individual variables for use within the `for`-block:

```
1 >>> for name, gen, oldness in zip(names, gender, age):
2 ...     if gen is 'M':
```

2. CONTROLLING PYTHON

```
3 ...     gen_str = 'male'
4 ...     else:
5 ...         gen_str = 'female'
6 ...     print(name + ' is ' + str(oldness) + ' years old and ' + gen_str)
7 ...
8 Brad is 20 years old and male
9 Sue is 22 years old and female
10 Mark is 49 years old and male
11 Jan is 32 years old and female
```

Be careful to only use the `zip()` function in cases where the lists you are zipping are of the same length or you don't care about retaining the values of the longest lists as `zip()` will truncate the results to only include elements where there are items in all arguments provided.

```
1 >>> names = ['Brad', 'Sue', 'Mark', 'Jan', 'Fred']
2 >>> gender = ['M', 'F', 'M', 'F', 'M']
3 >>> age = [20, 22, 49, 32] # Note that there is no age for Fred. Poor Fred :(
4 >>>
5 >>> details = zip(names, gender, age)
6 >>> print(list(details))
7 [('Brad', 'M', 20), ('Sue', 'F', 22), ('Mark', 'M', 49), ('Jan', 'F', 32)]
8 >>> # Note how since there was no age for Fred to zip, Fred has been truncated
9 >>> # from the results.
```

To keep the results of a `zip()` call for multiple uses or to store the new zipped tuples, it is possible to evaluate the *zip object* using the `list()` function. This returns the results of `zip()` as a list of tuples, where the i^{th} tuple contains the i^{th} element from each of the arguments passed to `zip()`.

```
1 >>> age = [20, 22, 49, 32, 42] # give Frank an age
2 >>> details = list(zip(names, gender, age))
3 >>> print(details)
4 [('Brad', 'M', 20), ('Sue', 'F', 22), ('Mark', 'M', 49), ('Jan', 'F', 32),
5  ('Frank', 'M', 42)]
6 >>> brad = details[0]
7 >>> sue = details[1]
8 >>> mark = details[2]
9 >>> jan = details[3]
10 >>> frank = details[4]
11 >>> # or using multiple assignment
12 >>> brad, sue, mark, jan, frank = details
```

While it may not seem revolutionary to begin with, using `zip()` in `for` loops can make your code significantly shorter, easier to read and, correspondingly, less buggy.

2.4 Example

Recently, as a result of a series of accidents caused by a sleepy truck driver, a portion of the grid near the border between New South Wales and Queensland has been adversely affected. It is your job to track down all the information relating to this area stored on your company's servers.

As a starting point, you must identify all the CSV files stored in the in the variable `FILES`. `FILES` is a list of all the files present on one of your company's servers.

You must return and print the paths of only the files which end in `'.csv'`.

You will be given the following files:

- `check_files.py`

You will modify this file in order to complete this example.

- `test_check_files.py`

The unit test case for this example. There is a separate test case in this file for task one and task two. You can run the test case like any other Python file:

```
C:\Windows\py.exe -3.7-32 tests\test-check-files.py
```

2.4.1 Task 1

Write a function `is_csv_ext` which takes a filename as an argument and returns `True` if the filename ends with `'.csv'` or `False` otherwise.

Example usage:

```
1 >>> is_csv_ext('csv_files\\generator.csv')
2 True
3 >>> is_csv_ext('csv_files\\generator.doc')
4 False
```

2.4.2 Task 2

Write a function `filter_list_for_csvs` which takes a list of filenames as its only argument and returns a new list containing only the paths which end in `'.csv'`. This function should make use of the `is_csv_ext` function you created in Task 1.

Example usage:

```
1 >>> filter_list_for_csvs(['csv_files\\generator.csv', 'csv_files\\generator.pdf'])
2 ['csv_files\\generator.csv']
```

2.5 Sources

- [1] Python Foundation. Official Python Tutorial, Chapter 3.2, 2011. <http://docs.python.org/3.3/tutorial/introduction.html#first-steps-towards-programming>.
- [2] Python Foundation. Official Python Tutorial, Chapter 4, 2011. <http://docs.python.org/3.3/tutorial/controlflow.html>.

Chapter 3

Data Types

You Will Learn

- Advanced usage of lists, tuples, sets and strings;
- Practical uses for Python dictionaries.

3.1 Data Structures¹

This chapter describes some things you’ve learned about already in more detail, and adds some new things as well.

3.1.1 More on Lists

The list data type has some more methods. Here are all of the methods of list objects:

`list.append(x)` Add an item to the end of the list. Equivalent to `a[len(a):] = [x]`.

`list.extend(L)` Extend the list by appending all the items in the given list. Equivalent to `a[len(a):] = L`.

`list.insert(i, x)` Insert an item at a given position. The first argument is the index of the element before which to insert, so `a.insert(0, x)` inserts at the front of the list, and `a.insert(len(a), x)` is equivalent to `a.append(x)`.

`list.remove(x)` Remove the first item from the list whose value is *x*. It is an error if there is no such item.

`list.pop([i])` Remove the item at the given position in the list, and return it. If no index is specified, `a.pop()` removes and returns the last item in the list. (The square brackets around the *i* in the method signature denote that the parameter is optional, not that you should type square brackets at that position. You will see this notation frequently in the Python Library Reference.)

`list.clear()` Remove all items from the list. Equivalent to `del a[:]`.

`list.index(x)` Return the index in the list of the first item whose value is *x*. It is an error if there is no such item.

`list.count(x)` Return the number of times *x* appears in the list.

`list.sort()` Sort the items of the list in place.

`list.reverse()` Reverse the elements of the list in place.

`list.copy()` Return a shallow copy of the list. Equivalent to `a[:]`.

An example that uses most of the list methods:

```
1 >>> a = [66.25, 333, 333, 1, 1234.5]
2 >>> print(a.count(333), a.count(66.25), a.count('x'))
3 2 1 0
4 >>> a.insert(2, -1)
5 >>> a.append(333)
6 >>> a
7 [66.25, 333, -1, 333, 1, 1234.5, 333]
8 >>> a.index(333)
9 1
10 >>> a.remove(333)
11 >>> a
12 [66.25, -1, 333, 1, 1234.5, 333]
13 >>> a.reverse()
14 >>> a
15 [333, 1234.5, 1, 333, -1, 66.25]
16 >>> a.sort()
17 >>> a
18 [-1, 1, 66.25, 333, 333, 1234.5]
```

You might have noticed that methods like `insert`, `remove` or `sort` that modify the list have no return value printed – they return `None`.¹ This is a design principle for all mutable data structures in Python.

3.1.1.1 Using Lists as Stacks

The list methods make it very easy to use a list as a stack, where the last element added is the first element retrieved (“last-in, first-out”). To add an item to the top of the stack, use `append()`. To retrieve an item from the top of the stack, use `pop()` without an explicit index. For example:

```
1 >>> stack = [3, 4, 5]
2 >>> stack.append(6)
3 >>> stack.append(7)
4 >>> stack
5 [3, 4, 5, 6, 7]
6 >>> stack.pop()
7 7
8 >>> stack
9 [3, 4, 5, 6]
10 >>> stack.pop()
```

¹Other languages may return the mutated object, which allows method chaining, such as `d->insert("a")->remove("b")->sort();`.

```
11 6
12 >>> stack.pop()
13 5
14 >>> stack
15 [3, 4]
```

3.1.1.2 Using Lists as Queues

It is also possible to use a list as a queue, where the first element added is the first element retrieved (“first-in, first-out”); however, lists are not efficient for this purpose. While appends and pops from the end of list are fast, doing inserts or pops from the beginning of a list is slow (because all of the other elements have to be shifted by one).

To implement a queue, use **collections.deque** which was designed to have fast appends and pops from both ends. For example:

```
1 >>> from collections import deque
2 >>> queue = deque(["Eric", "John", "Michael"])
3 >>> queue.append("Terry")           # Terry arrives
4 >>> queue.append("Graham")        # Graham arrives
5 >>> queue.popleft()                # The first to arrive now leaves
6 'Eric'
7 >>> queue.popleft()                # The second to arrive now leaves
8 'John'
9 >>> queue                          # Remaining queue in order of arrival
10 deque(['Michael', 'Terry', 'Graham'])
```

3.1.1.3 List Comprehensions

List comprehensions provide a concise way to create lists. Common applications are to make new lists where each element is the result of some operations applied to each member of another sequence or iterable, or to create a subsequence of those elements that satisfy a certain condition.

For example, assume we want to create a list of squares, like:

```
1 >>> squares = []
2 >>> for x in range(10):
3 ...     squares.append(x**2)
4 ...
5 >>> squares
6 [0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
```

We can obtain the same result with:

```
1 squares = [x**2 for x in range(10)]
```

This is also equivalent to `squares = list(map(lambda x: x**2, range(10)))`, but it’s more concise and readable.

3. DATA TYPES

A list comprehension consists of brackets containing an expression followed by a **for** clause, then zero or more **for** or **if** clauses. The result will be a new list resulting from evaluating the expression in the context of the **for** and **if** clauses which follow it. For example, this listcomp combines the elements of two lists if they are not equal:

```
1 >>> [(x, y) for x in [1,2,3] for y in [3,1,4] if x != y]
2 [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

and it's equivalent to:

```
1 >>> combs = []
2 >>> for x in [1,2,3]:
3 ...     for y in [3,1,4]:
4 ...         if x != y:
5 ...             combs.append((x, y))
6 ...
7 >>> combs
8 [(1, 3), (1, 4), (2, 3), (2, 1), (2, 4), (3, 1), (3, 4)]
```

Note how the order of the **for** and **if** statements is the same in both these snippets.

If the expression is a tuple (e.g. the (x, y) in the previous example), it must be parenthesized.

```
1 >>> vec = [-4, -2, 0, 2, 4]
2 >>> # create a new list with the values doubled
3 >>> [x*2 for x in vec]
4 [-8, -4, 0, 4, 8]
5 >>> # filter the list to exclude negative numbers
6 >>> [x for x in vec if x >= 0]
7 [0, 2, 4]
8 >>> # apply a function to all the elements
9 >>> [abs(x) for x in vec]
10 [4, 2, 0, 2, 4]
11 >>> # call a method on each element
12 >>> freshfruit = [' banana', ' loganberry ', 'passion fruit ']
13 >>> [weapon.strip() for weapon in freshfruit]
14 ['banana', 'loganberry', 'passion fruit']
15 >>> # create a list of 2-tuples like (number, square)
16 >>> [(x, x**2) for x in range(6)]
17 [(0, 0), (1, 1), (2, 4), (3, 9), (4, 16), (5, 25)]
18 >>> # the tuple must be parenthesized, otherwise an error is raised
19 >>> [x, x**2 for x in range(6)]
20 File "<stdin>", line 1, in ?
21     [x, x**2 for x in range(6)]
22         ^
23 SyntaxError: invalid syntax
24 >>> # flatten a list using a listcomp with two 'for'
25 >>> vec = [[1,2,3], [4,5,6], [7,8,9]]
26 >>> [num for elem in vec for num in elem]
27 [1, 2, 3, 4, 5, 6, 7, 8, 9]
```

List comprehensions can contain complex expressions and nested functions:

```
1 >>> from math import pi
2 >>> [str(round(pi, i)) for i in range(1, 6)]
3 ['3.1', '3.14', '3.142', '3.1416', '3.14159']
```

3.1.1.4 Nested List Comprehensions

The initial expression in a list comprehension can be any arbitrary expression, including another list comprehension.

Consider the following example of a 3x4 matrix implemented as a list of 3 lists of length 4:

```
1 >>> matrix = [
2 ...     [1, 2, 3, 4],
3 ...     [5, 6, 7, 8],
4 ...     [9, 10, 11, 12],
5 ... ]
```

The following list comprehension will transpose rows and columns:

```
1 >>> [[row[i] for row in matrix] for i in range(4)]
2 [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

As we saw in the previous section, the nested listcomp is evaluated in the context of the **for** that follows it, so this example is equivalent to:

```
1 >>> transposed = []
2 >>> for i in range(4):
3 ...     transposed.append([row[i] for row in matrix])
4 ...
5 >>> transposed
6 [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

which, in turn, is the same as:

```
1 >>> transposed = []
2 >>> for i in range(4):
3 ...     # the following 3 lines implement the nested listcomp
4 ...     transposed_row = []
5 ...     for row in matrix:
6 ...         transposed_row.append(row[i])
7 ...     transposed.append(transposed_row)
8 ...
9 >>> transposed
10 [[1, 5, 9], [2, 6, 10], [3, 7, 11], [4, 8, 12]]
```

In the real world, you should prefer built-in functions to complex flow statements. The **zip()** function would do a great job for this use case:

```
1 >>> list(zip(*matrix))
2 [(1, 5, 9), (2, 6, 10), (3, 7, 11), (4, 8, 12)]
```

See *Unpacking Argument Lists* for details on the asterisk in this line.

3.1.2 The `del` Statement

There is a way to remove an item from a list given its index instead of its value: the **`del`** statement. This differs from the **`pop()`** method which returns a value. The **`del`** statement can also be used to remove slices from a list or clear the entire list (which we did earlier by assignment of an empty list to the slice). For example:

```
1 >>> a = [-1, 1, 66.25, 333, 333, 1234.5]
2 >>> del a[0]
3 >>> a
4 [1, 66.25, 333, 333, 1234.5]
5 >>> del a[2:4]
6 >>> a
7 [1, 66.25, 1234.5]
8 >>> del a[:]
9 >>> a
10 []
```

`del` can also be used to delete entire variables:

```
1 >>> del a
```

Referencing the name **`a`** hereafter is an error (at least until another value is assigned to it). We'll find other uses for **`del`** later.

3.1.3 Tuples and Sequences

We saw that lists and strings have many common properties, such as indexing and slicing operations. They are two examples of *sequence* data types (see *Sequence Types* — *list, tuple, range*). Since Python is an evolving language, other sequence data types may be added. There is also another standard sequence data type: the *tuple*.

A tuple consists of a number of values separated by commas, for instance:

```
1 >>> t = 12345, 54321, 'hello!'
2 >>> t[0]
3 12345
4 >>> t
5 (12345, 54321, 'hello!')
6 >>> # Tuples may be nested:
7 ... u = t, (1, 2, 3, 4, 5)
8 >>> u
9 ((12345, 54321, 'hello!'), (1, 2, 3, 4, 5))
10 >>> # Tuples are immutable:
```

```
11 ... t[0] = 88888
12 Traceback (most recent call last):
13   File "<stdin>", line 1, in <module>
14   TypeError: 'tuple' object does not support item assignment
15 >>> # but they can contain mutable objects:
16 ... v = ([1, 2, 3], [3, 2, 1])
17 >>> v
18 ([1, 2, 3], [3, 2, 1])
```

As you see, on output tuples are always enclosed in parentheses, so that nested tuples are interpreted correctly; they may be input with or without surrounding parentheses, although often parentheses are necessary anyway (if the tuple is part of a larger expression). It is not possible to assign to the individual items of a tuple, however it is possible to create tuples which contain mutable objects, such as lists.

Though tuples may seem similar to lists, they are often used in different situations and for different purposes. Tuples are *immutable*, and usually contain an heterogeneous sequence of elements that are accessed via unpacking (see later in this section) or indexing (or even by attribute in the case of **namedtuples**). Lists are *mutable*, and their elements are usually homogeneous and are accessed by iterating over the list.

A special problem is the construction of tuples containing 0 or 1 items: the syntax has some extra quirks to accommodate these. Empty tuples are constructed by an empty pair of parentheses; a tuple with one item is constructed by following a value with a comma (it is not sufficient to enclose a single value in parentheses). Ugly, but effective. For example:

```
1 >>> empty = ()
2 >>> singleton = 'hello',    # <-- note trailing comma
3 >>> len(empty)
4 0
5 >>> len(singleton)
6 1
7 >>> singleton
8 ('hello',)
```

The statement `t = 12345, 54321, 'hello!'` is an example of *tuple packing*: the values 12345, 54321 and 'hello!' are packed together in a tuple. The reverse operation is also possible:

```
1 >>> x, y, z = t
```

This is called, appropriately enough, *sequence unpacking* and works for any sequence on the right-hand side. Sequence unpacking requires that there are as many variables on the left side of the equals sign as there are elements in the sequence. Note that multiple assignment is really just a combination of tuple packing and sequence unpacking.

3.1.4 Sets

Python also includes a data type for *sets*. A set is an unordered collection with no duplicate elements. Basic uses include membership testing and eliminating duplicate entries.

3. DATA TYPES

Set objects also support mathematical operations like union, intersection, difference, and symmetric difference.

Curly braces or the `set()` function can be used to create sets. Note: to create an empty set you have to use `set()`, not `{};` the latter creates an empty dictionary, a data structure that we discuss in the next section.

Here is a brief demonstration:

```
1 >>> basket = {'apple', 'orange', 'apple', 'pear', 'orange', 'banana'}
2 >>> print(basket)                # show that duplicates have been removed
3 {'orange', 'banana', 'pear', 'apple'}
4 >>> 'orange' in basket           # fast membership testing
5 True
6 >>> 'crabgrass' in basket
7 False
8
9 >>> # Demonstrate set operations on unique letters from two words
10 ...
11 >>> a = set('abracadabra')
12 >>> b = set('alacazam')
13 >>> a                            # unique letters in a
14 {'a', 'r', 'b', 'c', 'd'}
15 >>> a - b                        # letters in a but not in b
16 {'r', 'd', 'b'}
17 >>> a | b                        # letters in either a or b
18 {'a', 'c', 'r', 'd', 'b', 'm', 'z', 'l'}
19 >>> a & b                        # letters in both a and b
20 {'a', 'c'}
21 >>> a ^ b                        # letters in a or b but not both
22 {'r', 'd', 'b', 'm', 'z', 'l'}
```

Similarly to *list comprehensions*, set comprehensions are also supported:

```
1 >>> a = {x for x in 'abracadabra' if x not in 'abc'}
2 >>> a
3 {'r', 'd'}
```

3.1.5 Dictionaries

Another useful data type built into Python is the *dictionary* (see *Mapping Types — dict*). Dictionaries are sometimes found in other languages as “associative memories” or “associative arrays”. Unlike sequences, which are indexed by a range of numbers, dictionaries are indexed by *keys*, which can be any immutable type; strings and numbers can always be keys. Tuples can be used as keys if they contain only strings, numbers, or tuples; if a tuple contains any mutable object either directly or indirectly, it cannot be used as a key. You can’t use lists as keys, since lists can be modified in place using index assignments, slice assignments, or methods like `append()` and `extend()`.

It is best to think of a dictionary as an unordered set of *key: value* pairs, with the requirement that the keys are unique (within one dictionary). A pair of braces creates an empty dictionary: `{}`. Placing a comma-separated list of key:value pairs within the braces adds initial key:value pairs to the dictionary; this is also the way dictionaries are written on output.

The main operations on a dictionary are storing a value with some key and extracting the value given the key. It is also possible to delete a key:value pair with `del`. If you store using a key that is already in use, the old value associated with that key is forgotten. It is an error to extract a value using a non-existent key.

Performing `list(d.keys())` on a dictionary returns a list of all the keys used in the dictionary, in arbitrary order (if you want it sorted, just use `sorted(d.keys())` instead).² To check whether a single key is in the dictionary, use the `in` keyword.

Here is a small example using a dictionary:

```
1 >>> tel = {'jack': 4098, 'sape': 4139}
2 >>> tel['guido'] = 4127
3 >>> tel
4 {'sape': 4139, 'guido': 4127, 'jack': 4098}
5 >>> tel['jack']
6 4098
7 >>> del tel['sape']
8 >>> tel['irv'] = 4127
9 >>> tel
10 {'guido': 4127, 'irv': 4127, 'jack': 4098}
11 >>> list(tel.keys())
12 ['irv', 'guido', 'jack']
13 >>> sorted(tel.keys())
14 ['guido', 'irv', 'jack']
15 >>> 'guido' in tel
16 True
17 >>> 'jack' not in tel
18 False
```

The `dict()` constructor builds dictionaries directly from sequences of key-value pairs:

```
1 >>> dict([('sape', 4139), ('guido', 4127), ('jack', 4098)])
2 {'sape': 4139, 'jack': 4098, 'guido': 4127}
```

In addition, dict comprehensions can be used to create dictionaries from arbitrary key and value expressions:

```
1 >>> {x: x**2 for x in (2, 4, 6)}
2 {2: 4, 4: 16, 6: 36}
```

When the keys are simple strings, it is sometimes easier to specify pairs using keyword arguments:

```
1 >>> dict(sape=4139, guido=4127, jack=4098)
2 {'sape': 4139, 'jack': 4098, 'guido': 4127}
```

²Calling `d.keys()` will return a *dictionary view* object. It supports operations like membership test and iteration, but its contents are not independent of the original dictionary – it is only a *view*.

3.1.6 Looping Techniques

When looping through dictionaries, the key and corresponding value can be retrieved at the same time using the **items()** method.

```
1 >>> knights = {'gallahad': 'the pure', 'robin': 'the brave'}
2 >>> for k, v in knights.items():
3 ...     print(k, v)
4 ...
5 gallahad the pure
6 robin the brave
```

When looping through a sequence, the position index and corresponding value can be retrieved at the same time using the **enumerate()** function.

```
1 >>> for i, v in enumerate(['tic', 'tac', 'toe']):
2 ...     print(i, v)
3 ...
4 0 tic
5 1 tac
6 2 toe
```

To loop over two or more sequences at the same time, the entries can be paired with the **zip()** function.

```
1 >>> questions = ['name', 'quest', 'favorite color']
2 >>> answers = ['lancelot', 'the holy grail', 'blue']
3 >>> for q, a in zip(questions, answers):
4 ...     print('What is your {0}? It is {1}'.format(q, a))
5 ...
6 What is your name? It is lancelot.
7 What is your quest? It is the holy grail.
8 What is your favorite color? It is blue.
```

To loop over a sequence in reverse, first specify the sequence in a forward direction and then call the **reversed()** function.

```
1 >>> for i in reversed(range(1, 10, 2)):
2 ...     print(i)
3 ...
4 9
5 7
6 5
7 3
8 1
```

To loop over a sequence in sorted order, use the **sorted()** function which returns a new sorted list while leaving the source unaltered.

```
1 >>> basket = ['apple', 'orange', 'apple', 'pear', 'orange', 'banana']
2 >>> for f in sorted(set(basket)):
3 ...     print(f)
4 ...
5 apple
6 banana
7 orange
8 pear
```

To change a sequence you are iterating over while inside the loop (for example to duplicate certain items), it is recommended that you first make a copy. Looping over a sequence does not implicitly make a copy. The slice notation makes this especially convenient:

```
1 >>> words = ['cat', 'window', 'defenestrate']
2 >>> for w in words[:]: # Loop over a slice copy of the entire list.
3 ...     if len(w) > 6:
4 ...         words.insert(0, w)
5 ...
6 >>> words
7 ['defenestrate', 'cat', 'window', 'defenestrate']
```

3.1.7 More on Conditions

The conditions used in **while** and **if** statements can contain any operators, not just comparisons.

The comparison operators **in** and **not in** check whether a value occurs (does not occur) in a sequence. The operators **is** and **is not** compare whether two objects are really the same object; this only matters for mutable objects like lists. All comparison operators have the same priority, which is lower than that of all numerical operators.

Comparisons can be chained. For example, **a < b == c** tests whether **a** is less than **b** and moreover **b** equals **c**.

Comparisons may be combined using the Boolean operators **and** and **or**, and the outcome of a comparison (or of any other Boolean expression) may be negated with **not**. These have lower priorities than comparison operators; between them, **not** has the highest priority and **or** the lowest, so that **A and not B or C** is equivalent to **(A and (not B)) or C**. As always, parentheses can be used to express the desired composition.

The Boolean operators **and** and **or** are so-called *short-circuit* operators: their arguments are evaluated from left to right, and evaluation stops as soon as the outcome is determined. For example, if **A** and **C** are true but **B** is false, **A and B and C** does not evaluate the expression **C**. When used as a general value and not as a Boolean, the return value of a short-circuit operator is the last evaluated argument.

It is possible to assign the result of a comparison or other Boolean expression to a variable. For example,

```
1 >>> string1, string2, string3 = '', 'Trondheim', 'Hammer Dance'
2 >>> non_null = string1 or string2 or string3
```

```
3 >>> non_null
4 'Trondheim'
```

Note that in Python, unlike C, assignment cannot occur inside expressions. C programmers may grumble about this, but it avoids a common class of problems encountered in C programs: typing = in an expression when == was intended.

3.1.8 Comparing Sequences and Other Types

Sequence objects may be compared to other objects with the same sequence type. The comparison uses *lexicographical* ordering: first the first two items are compared, and if they differ this determines the outcome of the comparison; if they are equal, the next two items are compared, and so on, until either sequence is exhausted. If two items to be compared are themselves sequences of the same type, the lexicographical comparison is carried out recursively. If all items of two sequences compare equal, the sequences are considered equal. If one sequence is an initial sub-sequence of the other, the shorter sequence is the smaller (lesser) one. Lexicographical ordering for strings uses the Unicode codepoint number to order individual characters. Some examples of comparisons between sequences of the same type:

```
1 (1, 2, 3) < (1, 2, 4)
2 [1, 2, 3] < [1, 2, 4]
3 'ABC' < 'C' < 'Pascal' < 'Python'
4 (1, 2, 3, 4) < (1, 2, 4)
5 (1, 2) < (1, 2, -1)
6 (1, 2, 3) == (1.0, 2.0, 3.0)
7 (1, 2, ('aa', 'ab')) < (1, 2, ('abc', 'a'), 4)
```

Note that comparing objects of different types with < or > is legal provided that the objects have appropriate comparison methods. For example, mixed numeric types are compared according to their numeric value, so 0 equals 0.0, etc. Otherwise, rather than providing an arbitrary ordering, the interpreter will raise a **TypeError** exception.

3.2 Example: Using Lists and Dictionaries

3.2.1 Introduction

In order to assess the danger of a voltage collapse event, you are required to analyse the voltages at various buses around the affected area. Your colleague has given you a Python file of a list of bus numbers and voltages. The team suspects that buses around Armidale are of key concern and it is your job to see if the buses are operating outside their secure voltage range.

You will be given these files:

- `suspect_buses.py`

Your code for this example should be written into this file. The variables containing the bus numbers and corresponding voltages in `pu` are defined in this file.

- `test_suspect_buses.py`

The unit test case for this example. You can run the test case like any other Python file:

```
c:\Windows\py.exe -3.7-32 tests\test_suspect_buses.py
```

The output on the command line will show where the tests are failing and give you some helpful debugging information.

3.2.2 Task 1

Write a function, `check_bus_volt_loops(...)`, using `for` and `zip()` to loop over the list of bus numbers and voltages.

The function will take three arguments: a list of bus numbers, a list of voltages which correspond to the bus numbers and a list of the buses to be investigated. These lists are supplied for you in `suspect_buses.py` as `bus_num_list`, `voltage_list` and `suspect_bus_nums` respectively.

The function does not return anything, but must neatly print the bus numbers and corresponding voltages (using the same format shown as the example usage shown later) for the buses in `suspect_bus_nums`.

Using the printed data, and the assumption that a voltage less than 0.9pu is operating outside the secure voltage range, assess the team's prediction about the key buses.

Function Signature

```
check_bus_volt_loops(bus_nums, voltages, req_buses)
```

Arguments

- *bus_nums* A list of all bus numbers.
- *voltages* A list of the voltages that correspond to the bus numbers.
- *req_buses* A list of the buses of the buses we wish to know the voltage for.

Example usage

3. DATA TYPES

```
1 >>> # The voltages shown here are not the same as the data in voltage_list
2 >>> check_bus_volt_loops(bus_num_list, voltage_list, suspect_bus_nums)
3 20070: 1.0204
4 21179: 1.0324
5 ...
6 21770: 1.0410
```

Hint: It may be helpful to iterate over both `bus_num_list` and `voltage_list` using `zip()`. For example, try this in the Python interpreter:

```
1 >>> a = range(5)
2 >>> b = range(5,10)
3 >>> for one, two in zip(a,b):
4 ...     print('%s: %s' % (one, two))
5 ...
6 0: 5
7 1: 6
8 2: 7
9 3: 8
10 4: 9
```

Refer to Chapter 2.3 for more info about `zip()`.

3.2.3 Task 2

Instead of looping over the lists of data as in Task 1, use a dictionary to achieve the same result. That is, first convert the data to a dictionary using the values in `bus_num_list` as the keys and the voltages in `voltage_list` as the corresponding values.

Use this dictionary to retrieve the voltages at the suspect buses. Print the bus numbers and voltages to the screen in the same format as Task 1.

Function Signature

```
check_bus_volt_dict(bus_nums, voltages, req_buses)
```

Arguments

- *bus_nums* A list of all bus numbers.
- *voltages* A list of the voltages that correspond to the bus numbers.
- *req_buses* A list of the buses of the buses we wish to know the voltage for.

Example usage

```
1 >>> # The voltages shown here are not the same as the data in voltage_list
2 >>> check_bus_volt_dict(bus_num_list, voltage_list, suspect_bus_nums)
3 20070: 1.0204
4 21179: 1.0324
5 ...
6 21770: 1.0410
```

Hint: There are many ways to populate a dictionary. You may choose to loop over the lists together and set each value separately.

```
1 bus_volt_dict = {}
2 for bus, volt in zip(bus_num_list, voltage_list):
3     bus_volt_dict[bus] = volt
```

or you may want to `zip()` the lists so that they are grouped as a list of pairs and convert the pairs to a dictionary:

```
1 bus_volt_dict = dict(zip(bus_num_list, voltage_list))
```

3.3 Sources

- [1] Python Foundation. Official Python Tutorial, Chapter 5, 2011. <http://docs.python.org/3.3/tutorial/datastructures.html>.

Chapter 4

PSS[®]E

You Will Learn

- Initialising PSS[®]E from Python;
- Using the subsystem data retrieval API;
- Changing power flow data; and
- Charting and saving to file using `matplotlib`.

4.1 Initialising PSS[®]E From Python

The Python library files we have used so far are included with the official distribution of Python. Your code will run on any machine that has a version of Python 2.7 or 3.7 installed whether Mac, Windows or Linux. However the PSS[®]E library files are not part of the standard Python distribution and we must write some additional code so that Python knows where to find the files so it can import them.

It is important to know that the PSS[®]E library files are available for Python version 2.7 or 3.7 only. Fortunately, these are the versions included with the PSS[®]E installation.

Traditionally, to import a Python library you type:

```
1 >>> import psspy
```

For this to work we need to add the directory containing the `psspy.py` file to the list of directories that Python searches for imports. For PSS[®]E 34 the library is (by default) contained in:

```
1 c:\program files (x86)\pti\psse34\psspy37
```

Python searches the directories in the `sys.path` list when `import psspy` is executed. Here is a script you can use to add the PSS[®]E library to the `sys.path` and export it as an environment variable. PSS[®]E will use `psse34` import to set itself up somewhat before you begin.

4. PSS[®]E

```
1 import os, sys
2
3 PSSE_LOCATION = r'c:\program files (x86)\pti\psse34\psspy37'
4 sys.path.append(PSSE_LOCATION)
5 os.environ['PATH'] = os.environ['path'] + ';' + PSSE_LOCATION
6
7 import psse34
8 import psspy
```

Before running any scripts you will still need to initialise PSS[®]E.

```
1 psspy.psseinit(8000)
```

This command initialises PSS[®]E with space for 8000 buses and must be executed before you execute any other PSS[®]E commands.

4.1.1 About the PSS[®]E Subsystem Data Retrieval API

There are two ways to retrieve element information like bus voltage or machine power output from your saved case using the PSS[®]E API:

- The Single Element Data Retrieval API (Chapter 7 of the PSS[®]E API guide); or
- The Subsystem Data Retrieval API (Chapter 8 of the PSS[®]E API guide).

Single Element Data Retrieval

The single element data retrieval API is used to request a single piece of information (e.g. voltage) from a single element at a time. Here is how you might get the voltage at a bus using the single element data retrieval API:

```
1 >>> ierr, voltage = psspy.busdat(ibus=201, string="PU")
```

Subsystem Data Retrieval

The subsystem data retrieval requests multiple pieces of information (e.g. voltage and angle) from multiple elements at a time.

Here is how you might get the voltage and angle from all in-service buses in subsystem number 2:

```
1 >>> ierr, voltage_and_angle = psspy.abusreal(sid=2, string=["PU", "ANGLE"])
```

The `voltage_and_angle` variable will contain two lists, one with the voltages and the other with angles:

```
1 >>> print(voltage_and_angle)
2 [[1.0, 1.01, 0.99, 1.02], [0, 2, 2.3, 2.1, 2.2]]
```

To know the bus numbers for the voltage and angle, you must request the 'NUMBER' attribute from the int API:

```
1 >>> ierr, busnumbers = psspy.abusint(sid=2, string=["NUMBER"])
2 >>> print(busnumbers)
3 [[101, 102, 501, 403]]
```

4.1.2 About Changing Power Flow Data

Use the Power Flow Data Changing API (Chapter 2 of the PSS®E API guide) to edit and add elements.

This section will show you how to:

- Use keyword instead of positional arguments to change data; and
- Look up the API documents to find how to change something.

Keyword arguments and using the API

If you have ever recorded a Python file using PSS®E you have probably come across code that looks like this:

```
1 # using positional arguments.
2 psspy.bus_data_2(20010, [_i, _i, _i, _i], [12.3404, _f, _f], _s)
```

When recording files, PSS®E will produce these lengthy lines listing every possible argument in detail. If you wanted to change only one single piece of data on the bus, like base voltage in this example, then it still writes this long line. Such long lines make it difficult to understand what is going on. The `_i`, `_f` and `_s` are PSS®E's default values. They tell PSS®E that you don't really want to change that value.

`bus_data_2(i, intgar, realar, name)`

This is the signature for the `bus_data_2` function. Compare it with the example above. Can you see that the `intgar` is passed `[_i, _i, _i, _i]`? And `realar` is passed `[12.3404, _f, _f]`? The vast majority of the `bus_data_2` arguments are not changed, in fact most of them are left at their default. Here is another way to do the same thing. This time using keyword arguments.

```
1 # using keyword arguments.
2 psspy.bus_data_2(20010, realar=[12.3404, _f, _f])
```

4. PSS®E

It looks better already, PSS®E understands that you didn't want to change *intgar* or *name* values. This example may look manageable, but the three winding transformer data has seventeen (17) *realari* values. Here is what it looks like if you wanted to change only a handful of the three winding transformer values (like VMSTAR and SBS1-2):

```
1  # using keyword arguments.
2  psspy.three_wnd_impedance_data(20010, 20101, 203030, '1 ',
3      realari=[_f, _f, _f, _f, _f, _f, 100, _f, _f, _f, _f, _f, _f, _f, _f, 1.01, _f])
```

Forget about it. Here is a better way:

```
1  # using keyword arguments, the real way.
2  psspy.three_wnd_impedance_data(20010, 20101, 203030, '1 ',
3      realari7 =100,
4      realari16=1.01)
```

First, a quick aside: notice that I've used *realari* instead of *realari*? Well the `three_wnd_impedance_data` is peculiar in that its function signature is different to the other Power Flow Data Changing functions.

`three_wnd_impedance_data(i, j, k, ckt, intgar, realari, name)`

So it pays to check the signature listed in the documentation before you code.

I've referred to the seventh element of the *realari* array by `realari7`. This is a helpful shortcut that PSS®E have programmed into their functions. You can do the same with the *intgar* array too: `intgar1=1, intgar3=4`.

4.2 About the matplotlib API¹

matplotlib was originally created to emulate the MATLAB[®] graphics commands, but is completely independent of MATLAB[®].

matplotlib is divided into three parts, the `pylab` interface found in the `pyplot` module of the `matplotlib` package, can create plots with code quite similar to the figure generating code of MATLAB[®]. The two other parts encapsulate more fine-grained and advanced features of MATLAB[®], allowing you to change the way lines are drawn and how to display the chart, though we will have little use for these during our tutorial.

4.2.1 Your First matplotlib Chart²

`matplotlib.pyplot` is a collection of command style functions that make `matplotlib` work like MATLAB[®]. Each `pyplot` function makes some change to a figure: eg, create a figure, create a plotting area in a figure, plot some lines in a plotting area, decorate the plot with labels, etc.

`matplotlib.pyplot` is stateful, in that it keeps track of the current figure and plotting area, and the plotting functions are directed to the current axes. The result of the following code is shown in Figure 4.1.

```
1 import matplotlib.pyplot as plt
2 plt.plot([1,2,3,4])
3 plt.ylabel('some numbers')
4 plt.show()
```

You may be wondering why the x-axis ranges from 0-2 and the y-axis from 1-3. If you provide a single list or array to the `plot()` command, `matplotlib` assumes it is a sequence of y values, and automatically generates the x values for you. Since Python

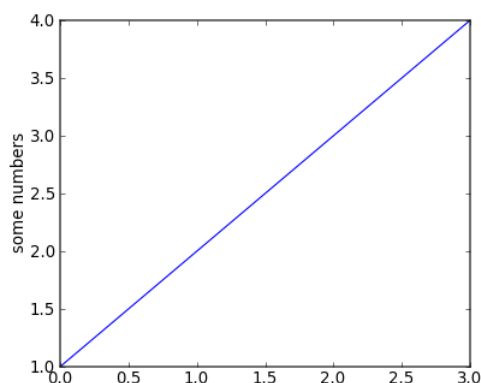


Figure 4.1: When only one list of data is provided, the data is plotted against its index in the data list.

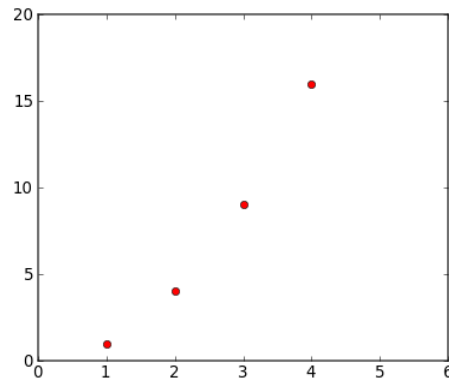


Figure 4.2: Specifying the line type as 'ro', draws red circles for the data points.

ranges start with 0, the default `x` vector has the same length as `y` but starts with 0. Hence the `x` data are `[0,1,2]`.

`plot()` is a versatile command, and will take an arbitrary number of arguments. For example, to plot `x` versus `y`, you can issue the command:

```
1 plt.plot([1,2,3,4], [1,4,9,16])
```

For every `x`, `y` pair of arguments, there is an optional third argument which is the format string that indicates the color and line type of the plot. The letters and symbols of the format string are from MATLAB®, and you concatenate a color string with a line style string. The default format string is 'b-', which is a solid blue line. For example, to plot the above with red circles, you would issue:

```
1 import matplotlib.pyplot as plt
2 plt.plot([1,2,3,4], [1,4,9,16], 'ro')
3 plt.axis([0, 6, 0, 20])
```

The output of this command is illustrated in Figure 4.2. See the `plot()` documentation for a complete list of line styles and format strings.

Also note the use of the `axis()` command in the example above. It takes a list of `[xmin, xmax, ymin, ymax]` to specify the viewport of the axes.

4.3 Example: Retrieving Data From PSS®E

4.3.1 Introduction

In the previous example your team was able to find some of the affected buses through intuition. While this methodology provided some good results, you know deep in your heart that good engineers never rely solely on a guess. You will use the PSS®E API to check all buses and assess whether they were affected.

You will be given these files:

- `model_check.py`

Your code for this example should be written into this file. Some helper functions to take care of some house keeping along the way have been included.

- `nsw.sav`

This is a model of the NSW network that you should be familiar with. We will be using this network throughout our examples as we continue to study the voltage collapse incident.

- `test_model_check.py`

The unit test case for this example. You can run the test case like any other Python file:

```
c:\Windows\py.exe -3.7-32 tests\test_model_check.py
```

The output on the command line will show where the tests are failing and give you some helpful debugging information.

4.3.2 Task 1

Write a function to retrieve bus numbers from the model using the PSS®E Python API. We will extend this function to return more data in the next task.

Function Signature

```
get_bus_data([sid=-1])
```

Arguments

- *sid*: bus subsystem to retrieve data for. If no value is supplied for this argument, the default is `'-1'`. Supplying `'-1'` will cause `psspy` to return data for all subsystems.

Return Value

A list of all the bus numbers.

Example usage

```
1 >>> get_bus_data()
2 [20010,
3  20101,
4  ...
5  40500]
```

Hint: Use the subsystem data retrieval API (Chapter 8 of PSS®E Python API) for buses (Chapter 8.3).

Hint: Remember that you will need to call a specific retrieval function for the different types of data (ie. `abusint(...)` for integer data, `abusreal(...)` for real data and `abuschar(...)` for character data).

4.3.3 Task 2

Extend `get_bus_data` to use the PSS®E Python API to retrieve the bus numbers, bus names and voltages from the model. The function signature and arguments are unchanged, but the return variable will need to contain numbers, names and voltages now.

Return Value

A list of tuples where one tuple will contain the bus number, name and voltage for a bus. Use `zip()` (Chapter 2.3) to group the 3 lists together into a list of tuples.

Example usage

```
1 >>> get_bus_data()
2 [(20010, 'BAYS 16KV' , 1.01),
3  (20101, 'LISM 132KV', 1.04),
4  ...
5  (40500, 'BRAE 275KV', 1.11)]
```

4.3.4 Task 3

Assume that voltages lower than 0.9 pu are operating outside of the secure voltage range. Write a new function, `filter_volt_collapse()`, to filter the list obtained in the previous task and return the buses where secure voltage limits are violated.

Function Signature

`filter_volt_collapse(bus_data)`

Arguments

- *bus_data*: The list returned from `get_bus_data()`.

Return Value

A list containing only the tuples from `get_bus_data()` where voltage is considered to have collapsed.

Example usage

```
1 >>> bus_data = get_bus_data()
2 >>> filter_volt_collapse(bus_data)
3 [(20070, 'ARM 330KV', 0.89),
4  ... ]
```

4.4 Sources

- [1] John Hunter, Darren Dale, and Michael Droettboom. Matplotlib User Guide - Introduction, 2011. <http://matplotlib.sourceforge.net/users/intro.html>.
- [2] John Hunter, Darren Dale, and Michael Droettboom. Matplotlib User Guide - pyplot, 2011. http://matplotlib.sourceforge.net/users/pyplot_tutorial.html.

Chapter 5

Importing Modules & File Structure

You Will Learn

- Importing modules;
- Creating your own modules;
- Scopes;
- Structuring your code to be easier to read.

5.1 Modules¹

If you quit from the Python interpreter and enter it again, the definitions you have made (functions and variables) are lost. Therefore, if you want to write a somewhat longer program, you are better off using a text editor to prepare the input for the interpreter and running it with that file as input instead. This is known as creating a *script*. As your program gets longer, you may want to split it into several files for easier maintenance. You may also want to use a handy function that you've written in several programs without copying its definition into each program.

To support this, Python has a way to put definitions in a file and use them in a script or in an interactive instance of the interpreter. Such a file is called a *module*; definitions from a module can be *imported* into other modules or into the *main* module (the collection of variables that you have access to in a script executed at the top level and in calculator mode).

A module is a file containing Python definitions and statements. The file name is the module name with the suffix `.py` appended. Within a module, the module's name (as a string) is available as the value of the global variable `__name__`. For instance, use your favorite text editor to create a file called `fibonacci.py` in the current directory with the following contents:

```
1 # Fibonacci numbers module
2
3 def fib(n):    # write Fibonacci series up to n
```

5. IMPORTING MODULES & FILE STRUCTURE

```
4     a, b = 0, 1
5     while b < n:
6         print(b, end=' ')
7         a, b = b, a+b
8     print()
9
10 def fib2(n): # return Fibonacci series up to n
11     result = []
12     a, b = 0, 1
13     while b < n:
14         result.append(b)
15         a, b = b, a+b
16     return result
```

Now enter the Python interpreter and import this module with the following command:

```
1 >>> import fibo
```

This does not enter the names of the functions defined in `fibo` directly in the current symbol table; it only enters the module name `fibo` there. Using the module name you can access the functions:

```
1 >>> fibo.fib(1000)
2 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610 987
3 >>> fibo.fib2(100)
4 [1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89]
5 >>> fibo.__name__
6 'fibo'
```

If you intend to use a function often you can assign it to a local name:

```
1 >>> fib = fibo.fib
2 >>> fib(500)
3 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

5.1.1 More on Modules

A module can contain executable statements as well as function definitions. These statements are intended to initialize the module. They are executed only the *first* time the module name is encountered in an import statement.¹ (They are also run if the file is executed as a script.)

Each module has its own private symbol table, which is used as the global symbol table by all functions defined in the module. Thus, the author of a module can use global variables in the module without worrying about accidental clashes with a user's global variables. On the other hand, if you know what you are doing you can

¹In fact function definitions are also 'statements' that are 'executed'; the execution of a module-level function definition enters the function name in the module's global symbol table.

touch a module's global variables with the same notation used to refer to its functions, `modname.itemname`.

Modules can import other modules. It is customary but not required to place all **import** statements at the beginning of a module (or script, for that matter). The imported module names are placed in the importing module's global symbol table.

There is a variant of the **import** statement that imports names from a module directly into the importing module's symbol table. For example:

```
1 >>> from fibo import fib, fib2
2 >>> fib(500)
3 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This does not introduce the module name from which the imports are taken in the local symbol table (so in the example, `fibo` is not defined).

There is even a variant to import all names that a module defines:

```
1 >>> from fibo import *
2 >>> fib(500)
3 1 1 2 3 5 8 13 21 34 55 89 144 233 377
```

This imports all names except those beginning with an underscore (`_`). In most cases Python programmers do not use this facility since it introduces an unknown set of names into the interpreter, possibly hiding some things you have already defined.

Note that in general the practice of importing `*` from a module or package is frowned upon, since it often causes poorly readable code. However, it is okay to use it to save typing in interactive sessions.

Note

For efficiency reasons, each module is only imported once per interpreter session. Therefore, if you change your modules, you must restart the interpreter – or, if it's just one module you want to test interactively, use **imp.reload()**, e.g. `import imp; imp.reload(modulename)`.

5.1.1.1 Executing modules as scripts

When you run a Python module with

```
python fibo.py <arguments>
```

the code in the module will be executed, just as if you imported it, but with the `__name__` set to `"__main__"`. That means that by adding this code at the end of your module:

```
1 if __name__ == "__main__":
2     import sys
3     fib(int(sys.argv[1]))
```

you can make the file usable as a script as well as an importable module, because the code that parses the command line only runs if the module is executed as the “main” file:

```
$ python fibo.py 50
1 1 2 3 5 8 13 21 34
```

If the module is imported, the code is not run:

```
1 >>> import fibo
2 >>>
```

This is often used either to provide a convenient user interface to a module, or for testing purposes (running the module as a script executes a test suite).

5.1.1.2 The Module Search Path

When a module named **spam** is imported, the interpreter first searches for a built-in module with that name. If not found, it then searches for a file named **spam.py** in a list of directories given by the variable **sys.path**. **sys.path** is initialized from these locations:

- the directory containing the input script (or the current directory).
- **PYTHONPATH** (a list of directory names, with the same syntax as the shell variable **PATH**).
- the installation-dependent default.

After initialization, Python programs can modify **sys.path**. The directory containing the script being run is placed at the beginning of the search path, ahead of the standard library path. This means that scripts in that directory will be loaded instead of modules of the same name in the library directory. This is an error unless the replacement is intended. See section *Standard Modules* for more information.

5.1.1.3 “Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the **__pycache__** directory under the name **module.version.pyc**, where the version encodes the format of the compiled file; it generally contains the Python version number. For example, in CPython release 3.3 the compiled version of **spam.py** would be cached as **__pycache__/spam.cpython-33.pyc**. This naming convention allows compiled modules from different releases and different versions of Python to coexist.

Python checks the modification date of the source against the compiled version to see if it’s out of date and needs to be recompiled. This is a completely automatic process. Also, the compiled modules are platform-independent, so the same library can be shared among systems with different architectures.

Python does not check the cache in two circumstances. First, it always recompiles and does not store the result for the module that's loaded directly from the command line. Second, it does not check the cache if there is no source module. To support a non-source (compiled only) distribution, the compiled module must be in the source directory, and there must not be a source module.

Some tips for experts:

- You can use the `-O` or `-OO` switches on the Python command to reduce the size of a compiled module. The `-O` switch removes assert statements, the `-OO` switch removes both assert statements and `__doc__` strings. Since some programs may rely on having these available, you should only use this option if you know what you're doing. "Optimized" modules have a `.pyo` rather than a `.pyc` suffix and are usually smaller. Future releases may change the effects of optimization.
- A program doesn't run any faster when it is read from a `.pyc` or `.pyo` file than when it is read from a `.py` file; the only thing that's faster about `.pyc` or `.pyo` files is the speed with which they are loaded.
- The module `compileall` can create `.pyc` files (or `.pyo` files when `-O` is used) for all modules in a directory.
- There is more detail on this process, including a flow chart of the decisions, in PEP 3147.

5.1.2 Standard Modules

Python comes with a library of standard modules, described in a separate document, the Python Library Reference ("Library Reference" hereafter). Some modules are built into the interpreter; these provide access to operations that are not part of the core of the language but are nevertheless built in, either for efficiency or to provide access to operating system primitives such as system calls. The set of such modules is a configuration option which also depends on the underlying platform. For example, the `winreg` module is only provided on Windows systems. One particular module deserves some attention: `sys`, which is built into every Python interpreter. The variables `sys.ps1` and `sys.ps2` define the strings used as primary and secondary prompts:

```
1 >>> import sys
2 >>> sys.ps1
3 '>>> '
4 >>> sys.ps2
5 '... '
6 >>> sys.ps1 = 'C> '
7 C> print('Yuck!')
8 Yuck!
9 C>
```

These two variables are only defined if the interpreter is in interactive mode.

The variable `sys.path` is a list of strings that determines the interpreter's search path for modules. It is initialized to a default path taken from the environment variable `PYTHONPATH`, or from a built-in default if `PYTHONPATH` is not set. You can modify it using standard list operations:

```
1 >>> import sys
2 >>> sys.path.append('/ufs/guido/lib/python')
```

5.1.3 The `dir()` Function

The built-in function `dir()` is used to find out which names a module defines. It returns a sorted list of strings:

```
1 >>> import fibo, sys
2 >>> dir(fibo)
3 ['__name__', 'fib', 'fib2']
4 >>> dir(sys)
5 ['__displayhook__', '__doc__', '__egginsert__', '__excepthook__',
6  '__loader__', '__name__', '__package__', '__plen__', '__stderr__',
7  '__stdin__', '__stdout__', '__clear_type_cache__', '_current_frames',
8  '_debugmallocstats', '_getframe', '_home', '_mercurial', '_xoptions',
9  'abiflags', 'api_version', 'argv', 'base_exec_prefix', 'base_prefix',
10 'builtin_module_names', 'byteorder', 'call_tracing', 'callstats',
11 'copyright', 'displayhook', 'dont_write_bytecode', 'exc_info',
12 'excepthook', 'exec_prefix', 'executable', 'exit', 'flags', 'float_info',
13 'float_repr_style', 'getcheckinterval', 'getdefaultencoding',
14 'getdlopenflags', 'getfilesystemencoding', 'getobjects', 'getprofile',
15 'getrecursionlimit', 'getrefcount', 'getsizeof', 'getswitchinterval',
16 'gettotalrefcount', 'gettrace', 'hash_info', 'hexversion',
17 'implementation', 'int_info', 'intern', 'maxsize', 'maxunicode',
18 'meta_path', 'modules', 'path', 'path_hooks', 'path_importer_cache',
19 'platform', 'prefix', 'ps1', 'setcheckinterval', 'setdlopenflags',
20 'setprofile', 'setrecursionlimit', 'setswitchinterval', 'settrace',
21 'stderr', 'stdin', 'stdout', 'thread_info', 'version', 'version_info',
22 'warnoptions']
```

Without arguments, `dir()` lists the names you have defined currently:

```
1 >>> a = [1, 2, 3, 4, 5]
2 >>> import fibo
3 >>> fib = fibo.fib
4 >>> dir()
5 ['__builtins__', '__name__', 'a', 'fib', 'fibo', 'sys']
```

Note that it lists all types of names: variables, modules, functions, etc.

`dir()` does not list the names of built-in functions and variables. If you want a list of those, they are defined in the standard module **`builtins`**:

```
1 >>> import builtins
2 >>> dir(builtins)
3 ['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
4  'BlockingIOError', 'BrokenPipeError', 'BufferError', 'BytesWarning',
5  'ChildProcessError', 'ConnectionAbortedError', 'ConnectionError',
6  'ConnectionRefusedError', 'ConnectionResetError', 'DeprecationWarning',
7  'EOFError', 'Ellipsis', 'EnvironmentError', 'Exception', 'False',
```

```

8  'FileExistsError', 'FileNotFoundError', 'FloatingPointError',
9  'FutureWarning', 'GeneratorExit', 'IOError', 'ImportError',
10 'ImportWarning', 'IndentationError', 'IndexError', 'InterruptedError',
11 'IsADirectoryError', 'KeyError', 'KeyboardInterrupt', 'LookupError',
12 'MemoryError', 'NameError', 'None', 'NotADirectoryError', 'NotImplemented',
13 'NotImplementedError', 'OSError', 'OverflowError',
14 'PendingDeprecationWarning', 'PermissionError', 'ProcessLookupError',
15 'ReferenceError', 'ResourceWarning', 'RuntimeError', 'RuntimeWarning',
16 'StopIteration', 'SyntaxError', 'SyntaxWarning', 'SystemError',
17 'SystemExit', 'TabError', 'TimeoutError', 'True', 'TypeError',
18 'UnboundLocalError', 'UnicodeDecodeError', 'UnicodeEncodeError',
19 'UnicodeError', 'UnicodeTranslateError', 'UnicodeWarning', 'UserWarning',
20 'ValueError', 'Warning', 'ZeroDivisionError', '_', '__build_class__',
21 '__debug__', '__doc__', '__import__', '__name__', '__package__', 'abs',
22 'all', 'any', 'ascii', 'bin', 'bool', 'bytearray', 'bytes', 'callable',
23 'chr', 'classmethod', 'compile', 'complex', 'copyright', 'credits',
24 'delattr', 'dict', 'dir', 'divmod', 'enumerate', 'eval', 'exec', 'exit',
25 'filter', 'float', 'format', 'frozenset', 'getattr', 'globals', 'hasattr',
26 'hash', 'help', 'hex', 'id', 'input', 'int', 'isinstance', 'issubclass',
27 'iter', 'len', 'license', 'list', 'locals', 'map', 'max', 'memoryview',
28 'min', 'next', 'object', 'oct', 'open', 'ord', 'pow', 'print', 'property',
29 'quit', 'range', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
30 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'vars',
31 'zip']

```

5.1.4 Packages

Packages are a way of structuring Python’s module namespace by using “dotted module names”. For example, the module name **A.B** designates a submodule named **B** in a package named **A**. Just like the use of modules saves the authors of different modules from having to worry about each other’s global variable names, the use of dotted module names saves the authors of multi-module packages like NumPy or the Python Imaging Library from having to worry about each other’s module names.

Suppose you want to design a collection of modules (a “package”) for the uniform handling of sound files and sound data. There are many different sound file formats (usually recognized by their extension, for example: **.wav**, **.aiff**, **.au**), so you may need to create and maintain a growing collection of modules for the conversion between the various file formats. There are also many different operations you might want to perform on sound data (such as mixing, adding echo, applying an equalizer function, creating an artificial stereo effect), so in addition you will be writing a never-ending stream of modules to perform these operations. Here’s a possible structure for your package (expressed in terms of a hierarchical filesystem):

1	sound/	Top-level package
2	__init__.py	Initialize the sound package
3	formats/	Subpackage for file format conversions
4	__init__.py	
5	wavread.py	
6	wavwrite.py	
7	aiffread.py	
8	aiffwrite.py	
9	auread.py	
10	auwrite.py	

5. IMPORTING MODULES & FILE STRUCTURE

```
11     ...
12     effects/                Subpackage for sound effects
13         __init__.py
14         echo.py
15         surround.py
16         reverse.py
17     ...
18     filters/                Subpackage for filters
19         __init__.py
20         equalizer.py
21         vocoder.py
22         karaoke.py
23     ...
```

When importing the package, Python searches through the directories on `sys.path` looking for the package subdirectory.

The `__init__.py` files are required to make Python treat the directories as containing packages; this is done to prevent directories with a common name, such as `string`, from unintentionally hiding valid modules that occur later on the module search path. In the simplest case, `__init__.py` can just be an empty file, but it can also execute initialization code for the package or set the `__all__` variable, described later.

Users of the package can import individual modules from the package, for example:

```
1 import sound.effects.echo
```

This loads the submodule `sound.effects.echo`. It must be referenced with its full name.

```
1 sound.effects.echo.echofilter(input, output, delay=0.7, atten=4)
```

An alternative way of importing the submodule is:

```
1 from sound.effects import echo
```

This also loads the submodule `echo`, and makes it available without its package prefix, so it can be used as follows:

```
1 echo.echofilter(input, output, delay=0.7, atten=4)
```

Yet another variation is to import the desired function or variable directly:

```
1 from sound.effects.echo import echofilter
```

Again, this loads the submodule `echo`, but this makes its function `echofilter()` directly available:

```
1 echofilter(input, output, delay=0.7, atten=4)
```

Note that when using `from package import item`, the item can be either a submodule (or subpackage) of the package, or some other name defined in the package, like a function, class or variable. The `import` statement first tests whether the item is defined in the package; if not, it assumes it is a module and attempts to load it. If it fails to find it, an **ImportError** exception is raised.

Contrarily, when using syntax like `import item.subitem.subsubitem`, each item except for the last must be a package; the last item can be a module or a package but can't be a class or function or variable defined in the previous item.

5.1.4.1 Importing * From a Package

Now what happens when the user writes `from sound.effects import *`? Ideally, one would hope that this somehow goes out to the filesystem, finds which submodules are present in the package, and imports them all. This could take a long time and importing sub-modules might have unwanted side-effects that should only happen when the submodule is explicitly imported.

The only solution is for the package author to provide an explicit index of the package. The `import` statement uses the following convention: if a package's `__init__.py` code defines a list named `__all__`, it is taken to be the list of module names that should be imported when `from package import *` is encountered. It is up to the package author to keep this list up-to-date when a new version of the package is released. Package authors may also decide not to support it, if they don't see a use for importing `*` from their package. For example, the file `sound/effects/__init__.py` could contain the following code:

```
1 __all__ = ["echo", "surround", "reverse"]
```

This would mean that `from sound.effects import *` would import the three named submodules of the **sound** package.

If `__all__` is not defined, the statement `from sound.effects import *` does *not* import all submodules from the package **sound.effects** into the current namespace; it only ensures that the package **sound.effects** has been imported (possibly running any initialization code in `__init__.py`) and then imports whatever names are defined in the package. This includes any names defined (and submodules explicitly loaded) by `__init__.py`. It also includes any submodules of the package that were explicitly loaded by previous `import` statements. Consider this code:

```
1 import sound.effects.echo
2 import sound.effects.surround
3 from sound.effects import *
```

In this example, the **echo** and **surround** modules are imported in the current namespace because they are defined in the **sound.effects** package when the `from...import` statement is executed. (This also works when `__all__` is defined.)

Although certain modules are designed to export only names that follow certain patterns when you use `import *`, it is still considered bad practice in production code.

Remember, there is nothing wrong with using:

```
1 from Package import specific_submodule}
```

In fact, this is the recommended notation unless the importing module needs to use submodules with the same name from different packages.

5.1.4.2 Intra-package References

When packages are structured into subpackages (as with the **sound** package in the example), you can use absolute imports to refer to submodules of siblings packages. For example, if the module **sound.filters.vocoder** needs to use the **echo** module in the **sound.effects** package, it can use `from sound.effects import echo`.

You can also write relative imports, with the `from module import name` form of import statement. These imports use leading dots to indicate the current and parent packages involved in the relative import. From the **surround** module for example, you might use:

```
1 from . import echo
2 from .. import formats
3 from ..filters import equalizer
```

Note that relative imports are based on the name of the current module. Since the name of the main module is always `"__main__"`, modules intended for use as the main module of a Python application must always use absolute imports.

5.1.4.3 Packages in Multiple Directories

Packages support one more special attribute, `__path__`. This is initialized to be a list containing the name of the directory holding the package's `__init__.py` before the code in that file is executed. This variable can be modified; doing so affects future searches for modules and subpackages contained in the package.

While this feature is not often needed, it can be used to extend the set of modules found in a package.

5.2 A Word on Scope

5.2.1 What's in a Scope

Scope refers to the collection of names that Python can access at a given point in your code. Names refer to objects such as variables, functions and a few other things. To begin, let's look at the initial scope of an interactive session. The `dir()` function will return a list of all the names associated with the name passed in as the argument (`dir(name)`). Calling `dir()` with no argument (`dir()`) returns a list of the names defined within the currently accessible scope:

```
1 >>> dir()
2 ['__builtins__', '__doc__', '__name__', '__package__']
```

Just out of interest, the names here are ones that are common to most modules:

- `__builtins__`: the built in language features of Python (such as `int()`, `float()` and `dir()`);
- `__doc__`: the docstring describing the module;
- `__name__`: the name of module; and
- `__package__`: the name of the package the module was imported from.

As this is the interactive environment, `__doc__` and `__package__` are not assigned any meaningful value and thus both contain the value `None`. `__name__` does have a unique value: `'__main__'`. When modules are imported, `__name__` contains the name of the module. In the special cases where the file is executed as a script or we are using the Python interpreter, `__name__` takes on the special value of `'__main__'`.

By creating variables and importing modules, we add to the names defined in this scope.

```
1 >>> myvariable = 42
2 >>> dir()
3 ['__builtins__', '__doc__', '__name__', '__package__', 'myvariable']
4 >>> import math
5 ['__builtins__', '__doc__', '__name__', '__package__', 'math', 'myvariable']
```

5.2.2 Copious Scopes

You may now be wondering why it is possible to call inbuilt functions such as `max()`, `min()` and `range()` when they do not appear in the list provided by `dir()`. The reason is that there are at least three nested scopes in play at the any time during execution: local scopes, global (module) scope and the builtin scope. Each scope is searched for a match in turn, starting at the innermost (local) scope, followed by the global (also called the module) scope and finally the builtins scope. The builtins scope is where `max()`, `min()` and `range()` live. The following text will describe the bounds of each of these scopes.

It is import to understand that scopes have everything to do with the textual structure of your code; a function defined within a file can search the global scope of the file it was defined in, the builtins and nothing else. It is just as important to recognise that scopes have very little to do with how the code is executed; a function called from outside the module it was defined in (after importing it, for example) will search the global scope of its home module and not the scope of the module which called it.

Local Scope

Local scopes are the most nested scope in Python: the name search will progress from the innermost scope out towards the outer most scopes (module followed by builtins). A local scope is created every time a function is entered. Upon entering the function, the local scope contains the function's arguments and any names assigned to during the function. If a name not in the local scope is referenced, it will search the parent scopes until it finds a match. If no matches are found, it will raise an exception: `NameError`.

As the local scope contains the names of all variables assigned to during the function the moment the function is entered, it is not possible to assign directly to names stored in other scopes. Any assignment encountered within that function will take place within the local scope. Here is an example.

```
1 >>> # module level
2 >>> modulevar = 6
3 >>> def A():
4 ...     modulevar = 10
5 >>> A()      # call the function to change modulevar
6 >>> print(modulevar)    # print the value of modulevar in the module scope.
7 6
```

Notice that the variable outside the local scope (outside the function body) has remained unchanged. The assignment to the name `modulevar` in the function `A()` took place in the local scope and had no effect beyond the bounds of that function. It is possible to assign to names defined in the module scope by declaring the name as `global` within the function. In this case, all assignments to that name will take place in the module scope.

```
1 >>> # module level
2 >>> modulevar = 6
3 >>> def A():
4 ...     global modulevar
5 ...     global Avar      # doesn't exist in the module scope yet.
6 ...     modulevar = 10
7 ...     Avar = 42
8 >>> A()      # call change global variables.
9 >>> print(modulevar)    # print the value of modulevar in the module scope.
10 10
11 >>> print(Avar)      # Avar now exists in the module scope.
12 42
```

By restricting where names are allowed to be used, functions are able to be treated as black boxes: variables go in and the result comes out without much opportunity to cause side-effects in the program at large.

The exception to the black box analogy is lists and other mutable types. For example, it is possible to use the methods of a list to change its contents and these changes will be propagated beyond the local scope. That is, you can append new elements, remove others and the changes would reach beyond the local scope. However, trying to assign a new list to this name would result in a list object of the same name being created in the local name space and all changes to that name would be local.

```
1 >>> # module level
2 >>> mylist = [1,2]
3 >>> myotherlist = [1,2]
4 >>> def A():
5 ...     # 'mylist' won't be found in this local scope, so it will search the
6 ...     # level above and find the one defined there.
7 ...     mylist.extend([3,4])
8 ...     # All assignments occur locally so myotherlist will remain the same.
9 ...     myotherlist = [100,200]
10 >>> print(mylist)
11 [1,2]
12 >>> print(myotherlist)
13 [1,2]
14 >>> A()
15 >>> print(mylist)
16 [1,2,3,4]
17 >>> print(myotherlist) # no change here
18 [1,2]
19 >>> A()
20 >>> print(mylist)
21 [1,2,3,4,3,4]
```

Unintended alterations to variables from within a function is a recipe for confusion. For this reason, it is wise to avoid using global variables wherever possible.

Global (Module) Scope

The global scope and module scope are different names for the same scope. This scope consists of all names defined at the module level. Module level names do not reside within any context which would generate a local scope, such as a function. In the following example, the variable `answer` and the function `A()` are names in the module scope.

```
1 >>> # module or global scope
2 >>> answer = 42
3 >>> def A():
4 ...     # Local scope
5 ...     greeting = 'hello'
```

Every module has its own unique module scope. Importing a module, say the `math` module, into a module called `mymod` only gives the functions in the `mymod` module

access to routines in the `math` module. The `math` module is unaware the `mymod` module has imported it. This separation allows programmers to use the same variable and function names in different sections of code without interfering with each other.

Builtins Scope

The final scope to be searched is always the builtin scope. This, predictably, contains the builtin Python functionality. The contents of the builtin scope can be examined by passing the builtins module to `dir()`.

```
1 >>> dir(__builtins__)
2 ['ArithmeticError', 'AssertionError', 'AttributeError', 'BaseException',
3   ...snip...
4 'eval', 'execfile', 'exit', 'file', 'filter', 'float', 'format', 'frozenset',
5 'getattr', 'globals', 'hasattr', 'hash', 'help', 'hex', 'id', 'input', 'int',
6 'intern', 'isinstance', 'issubclass', 'iter', 'len', 'license', 'list',
7 'locals', 'long', 'map', 'max', 'memoryview', 'min', 'next', 'object', 'oct',
8 'open', 'ord', 'pow', 'print', 'property', 'quit', 'range', 'raw_input',
9 'reduce', 'reload', 'repr', 'reversed', 'round', 'set', 'setattr', 'slice',
10 'sorted', 'staticmethod', 'str', 'sum', 'super', 'tuple', 'type', 'unichr',
11 'unicode', 'vars', 'xrange', 'zip']
```

5.2.3 The Takeaway Concepts

The discussion of scopes has been somewhat more esoteric than the other chapters; focusing more on how Python works under the hood than how to make Python do great things. For those playing at home, here's what you should be taking away from this section:

- Scopes define the objects (names) that you have access to at any given place in your code;
- There are three main scopes and they are searched for names in this order:
 1. Local;
 2. Global and Module;
 3. Builtins.
- Every module has its own, completely independent set of these scopes. This still applies even if you “`from modname import func`”. Though it is possible to call `func` without prefixing it with the name of the module it came from, its search path will still use the scopes of the module it was defined in. (i.e. `func` will search the `modname` scope not the one it was imported into).
- A scope is completely defined by the textual structure of your module, not how it executes.
- Unless specified with `global`, all variable assignments occur in the local scope.

5.3 Structuring Code

There is a lot of advice on how to write good code. Frustratingly, for every person giving advice, each will have their own mantras and principles about how to compose good code. The frustrating part about this is that one author's principles may contradict another's, and both may be completely valid. The reason this predicament exists stems from the most important limitation that anyone who writes code shares: our brains.

Every principle that is expounded by experts is, at its heart, addressing the same problem: managing complexity. Broadly generalizing, there are two categories of techniques, both of which will be summarized here and discussed later in this section.

The first group of techniques focuses on removing unnecessary complexity. Unnecessary complexity relates to the difficulty of understanding a piece of code because of the way it is written rather than because the problem is itself complex. Most advice addressing unnecessary complexity consists of utilising consistent coding conventions, such as naming conventions for constant variables. Other advice consists of avoiding combining language features in ways which decrease the comprehensibility of the code. The purpose of these techniques is to reduce the burden on the reader (who may be the original author reviewing the code six months later) to comprehend the code in front of them.

The second group of techniques focuses on managing the complexity of the problem at hand. These techniques focus on breaking a problem which has a high degree of complexity into several smaller problems, each of a shallower complexity. This reduces the number specific details relating to the problem being solved which must be actively kept in mind while writing. In essence, these techniques recognise that the human mind can only handle a limited amount of information at a given time and, therefore, they attempt to encourage techniques which improve comprehension and manageability. It can be said that they encourage the author to recognise the limitations of the human brain and to approach programming in a humble manner.

5.3.1 Coding Style

Coding style is important because it can improve the readability of code. It is essential to understand that code is always read more than it is written. While it may take longer to write readable code, the extra effort will pay for itself the next time someone reads your code. Remember that the next person who reads your code could be you in six months time.

Refer to PEP 8 for many suggestions about what style to code in. This document has been prepared by the lead developers of the Python language.

Be consistent in the manner you select names for functions and variables. Regardless of the chosen conventions, as long as people reading the code can get to know the conventions used, the code will be easier to read. An example might be to use descriptive names for all of your functions and variables.

Example descriptive names:

```
1 nsw_bus_numbers = [20001, 20240, 23404]
2 def find_swing_bus():
3     """Returns the swingbus number for the currently loaded case."""
```

5.3.2 Documentation

At a very minimum you should include a comment at the start of every function describing the data that it takes in and the data that it returns.

Comments should be used to document difficult parts of the code. Be careful not to confuse complex code with bad code that should be rewritten. If a section of code requires an inordinate amount of comments to explain how it works, it is possible that it that section could be rewritten in a simpler manner.

Make sure comments are updated along with any code that is being rewritten. It saves everyone time when the comments match what the code is doing. It is better to have no comments than it is to have incorrect comments.

5.3.3 Constants and Configurations

Remove all ‘magic’ numbers from your code. ‘Magic’ numbers are the constants that are defined throughout code. Most often, they represent default values or an estimate of the most likely value of a parameter and, in most cases, are created at a whim. The problem with ‘magic’ numbers is that a new reader of the code may not know why these magic numbers exist or how their value was derived.

If ‘magic’ numbers are used within a small script, and they will only be used in the one file, collect them at the beginning of the file and make them stand out by using ALL.CAPS. Furthermore, if the constant’s purpose or the reasoning for the constant’s value is not immediately obvious, include a brief comment documenting these characteristics. That way a reader will understand what variables are constant and know where to look for more information about them.

In the following example, it isn’t immediately clear what the point of getting Load 1 is.

```
1 load = conn.GetObj('Load', 'L01')
```

Using a named constant variable near the top of the file, it becomes obvious that we are getting a load to switch off. Moreover, if the same value has been used in more than one location, this method provides a consistent and bug free way of altering this value.

```
1 #----- Constants
2 OFFLINE_TARGET = 'L01'
3
4 offline_load = conn.GetObj('Load', 'L01')
```

If the script is a bit longer or you have constants that are shared between scripts, consider creating a constants file, defining all the constants in this file and importing

it into the script when necessary. This makes it easy to share constants and find them when they need to be changed.

Similarly, if you have a configuration that you wish to define, instead of creating variables in your scripts which define the output or input files, make the changes in a settings or constants file so that they are easily accessible.

5.3.4 Functions

Try to make functions that do one simple task really well instead of a few big ones that do everything. This is also a good guide to managing complexity. By limiting a function to doing only a specific action, the number of specific details required for this routine can be minimised. Writing targeted functions also helps maintainability. If every function has one task to do, the thinking that the reader needs to do to understand it is reduced. Small functions are often easier to test than larger functions too.

There are many warning signs that suggest a routine may need to be refactored. A few of them are listed below.

- Many levels of indentation:

Levels of indentation refer to the number of nested blocks of code (nested `if` statements or loops) within a function. While there is no hard and fast rule about the maximum number of indentation levels for good code (although the fewer the better), you should aim to make your code so that it is arranged in a logical manner. If number of nested blocks is getting large, consider breaking some of the inner blocks into their own function.

- Very long functions:

Defining an optimal length for a function is a very thorny issue. Some pundits say that they should be no longer than a screen or two worth of text. Others (from peer reviewed journals) suggest that functions should aim to be no more than 200 lines of code (excluding whitespace and comments). These recommendations are just that: recommendations. Sometimes it won't be possible to achieve these lengths, but don't let this be an excuse not to split functions into smaller pieces. If it is possible (and only in the most rare cases is it not) if a function is getting large and unwieldy, break the large function into smaller ones.

Moving commonly used code into its own function: An example

The following is an example story that highlights the importance of using repeatable functions.

Consider a piece of code that writes the current status of a branch to file. The writing statement is simple, one or two lines at most, and so the code is written out in full in five different locations where the program is required to write to file.

After the work has been completed, it appears that the developer, who is usually quite good at spelling, entered a typo. Instead of writing `branch status: ###`, it reads, `brnach status: ###`. This bug is simple to fix. It could be done with three keystrokes. However, due to the way it was originally written, the developer will have read through the code to find the five occurrences and fix them individually ("or was it six...gee it has been a while since I wrote this code"). If, when the programmer

was about to write this line for the second time, instead decided to rewrite the original instance into a function, the bug could have been fixed in one location and the programmer would have been completely satisfied that every instance of this bug was fixed.

The moral of this example: even though it may seem like such a small piece of trivial code, if you need to write it out more than once, it should be in a function. Doing so, as we just saw, improves maintainability. For any section of code that is longer than a few lines, it also improves readability by condensing common code into easily recognised function calls.

5.3.5 Creating Libraries

During the process of breaking the code down into functions, it is common to find functionality that you have seen in another piece of code. Moving these into a file that the two files share is a good idea for the same reasons as before: it is only necessary to fix bugs in one location. Having a library of code that is well tested reduces the number of bugs in new code.

5.4 Sources

- [1] Python Foundation. Official Python Tutorial, Chapter 6, 2011. <http://docs.python.org/3.3/tutorial/modules.html>.

Chapter 6

Input / Output

You Will Learn

- Reading text from a file;
- Writing text to a file;
- Converting from text to Python numbers.

6.1 Input and Output¹

There are several ways to present the output of a program; data can be printed in a human-readable form, or written to a file for future use. This chapter will discuss some of the possibilities.

6.1.1 Fancier Output Formatting

So far we've encountered two ways of writing values: *expression statements* and the **print()** function. (A third way is using the **write()** method of file objects; the standard output file can be referenced as **sys.stdout**. See the Library Reference for more information on this.)

Often you'll want more control over the formatting of your output than simply printing space-separated values. There are two ways to format your output; the first way is to do all the string handling yourself; using string slicing and concatenation operations you can create any layout you can imagine. The string type has some methods that perform useful operations for padding strings to a given column width; these will be discussed shortly. The second way is to use the **str.format()** method.

The **string** module contains a **Template** class which offers yet another way to substitute values into strings.

One question remains, of course: how do you convert values to strings? Luckily, Python has ways to convert any value to a string: pass it to the **repr()** or **str()** functions.

The **str()** function is meant to return representations of values which are fairly human-readable, while **repr()** is meant to generate representations which can be read by the interpreter (or will force a **SyntaxError** if there is no equivalent syntax). For objects which don't have a particular representation for human consumption, **str()**

will return the same value as `repr()`. Many values, such as numbers or structures like lists and dictionaries, have the same representation using either function. Strings, in particular, have two distinct representations.

Some examples:

```
1 >>> s = 'Hello, world.'
2 >>> str(s)
3 'Hello, world.'
4 >>> repr(s)
5 "'Hello, world.'"
6 >>> str(1/7)
7 '0.14285714285714285'
8 >>> x = 10 * 3.25
9 >>> y = 200 * 200
10 >>> s = 'The value of x is ' + repr(x) + ', and y is ' + repr(y) + '...'
11 >>> print(s)
12 The value of x is 32.5, and y is 40000...
13 >>> # The repr() of a string adds string quotes and backslashes:
14 ... hello = 'hello, world\n'
15 >>> hellos = repr(hello)
16 >>> print(hellos)
17 'hello, world\n'
18 >>> # The argument to repr() may be any Python object:
19 ... repr((x, y, ('spam', 'eggs'))))
20 "(32.5, 40000, ('spam', 'eggs'))"
```

Here are two ways to write a table of squares and cubes:

```
1 >>> for x in range(1, 11):
2 ...     print(repr(x).rjust(2), repr(x*x).rjust(3), end=' ')
3 ...     # Note use of 'end' on previous line
4 ...     print(repr(x*x*x).rjust(4))
5 ...
6 1   1   1
7 2   4   8
8 3   9  27
9 4  16  64
10 5  25 125
11 6  36 216
12 7  49 343
13 8  64 512
14 9  81 729
15 10 100 1000
16
17 >>> for x in range(1, 11):
18 ...     print('{0:2d} {1:3d} {2:4d}'.format(x, x*x, x*x*x))
19 ...
20 1   1   1
21 2   4   8
22 3   9  27
23 4  16  64
24 5  25 125
25 6  36 216
26 7  49 343
27 8  64 512
```

```
28 9 81 729
29 10 100 1000
```

(Note that in the first example, one space between each column was added by the way `print()` works: it always adds spaces between its arguments.)

This example demonstrates the `str.rjust()` method of string objects, which right-justifies a string in a field of a given width by padding it with spaces on the left. There are similar methods `str.ljust()` and `str.center()`. These methods do not write anything, they just return a new string. If the input string is too long, they don't truncate it, but return it unchanged; this will mess up your column lay-out but that's usually better than the alternative, which would be lying about a value. (If you really want truncation you can always add a slice operation, as in `x.ljust(n)[:n]`.)

There is another method, `str.zfill()`, which pads a numeric string on the left with zeros. It understands about plus and minus signs:

```
1 >>> '12'.zfill(5)
2 '00012'
3 >>> '-3.14'.zfill(7)
4 '-003.14'
5 >>> '3.14159265359'.zfill(5)
6 '3.14159265359'
```

Basic usage of the `str.format()` method looks like this:

```
1 >>> print('We are the {} who say "{}!"'.format('knights', 'Ni'))
2 We are the knights who say "Ni!"
```

The brackets and characters within them (called format fields) are replaced with the objects passed into the `str.format()` method. A number in the brackets can be used to refer to the position of the object passed into the `str.format()` method.

```
1 >>> print('{0} and {1}'.format('spam', 'eggs'))
2 spam and eggs
3 >>> print('{1} and {0}'.format('spam', 'eggs'))
4 eggs and spam
```

If keyword arguments are used in the `str.format()` method, their values are referred to by using the name of the argument.

```
1 >>> print('This {food} is {adjective}.'.format(
2 ...     food='spam', adjective='absolutely horrible'))
3 This spam is absolutely horrible.
```

Positional and keyword arguments can be arbitrarily combined:

6. INPUT / OUTPUT

```
1 >>> print('The story of {0}, {1}, and {other}'.format('Bill', 'Manfred', other='Georg'))
2 The story of Bill, Manfred, and Georg.
```

'!a' (apply `ascii()`), '!s' (apply `str()`) and '!r' (apply `repr()`) can be used to convert the value before it is formatted:

```
1 >>> import math
2 >>> print('The value of PI is approximately {}'.format(math.pi))
3 The value of PI is approximately 3.14159265359.
4 >>> print('The value of PI is approximately {!r}'.format(math.pi))
5 The value of PI is approximately 3.141592653589793.
```

An optional ':' and format specifier can follow the field name. This allows greater control over how the value is formatted. The following example rounds Pi to three places after the decimal.

```
1 >>> import math
2 >>> print('The value of PI is approximately {:.3f}'.format(math.pi))
3 The value of PI is approximately 3.142.
```

Passing an integer after the ':' will cause that field to be a minimum number of characters wide. This is useful for making tables pretty.

```
1 >>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 7678}
2 >>> for name, phone in table.items():
3 ...     print('{0:10} ==> {1:10d}'.format(name, phone))
4 ...
5 Jack          ==>      4098
6 Dcab          ==>      7678
7 Sjoerd        ==>      4127
```

If you have a really long format string that you don't want to split up, it would be nice if you could reference the variables to be formatted by name instead of by position. This can be done by simply passing the dict and using square brackets '[]' to access the keys

```
1 >>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
2 >>> print('Jack: {0[Jack]:d}; Sjoerd: {0[Sjoerd]:d}; '
3 ...      'Dcab: {0[Dcab]:d}'.format(table))
4 Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This could also be done by passing the table as keyword arguments with the '**' notation.

```
1 >>> table = {'Sjoerd': 4127, 'Jack': 4098, 'Dcab': 8637678}
2 >>> print('Jack: {Jack:d}; Sjoerd: {Sjoerd:d}; Dcab: {Dcab:d}'.format(**table))
3 Jack: 4098; Sjoerd: 4127; Dcab: 8637678
```

This is particularly useful in combination with the built-in function `vars()`, which returns a dictionary containing all local variables.

For a complete overview of string formatting with `str.format()`, see *Format String Syntax*.

6.1.1.1 Old string formatting

The `%` operator can also be used for string formatting. It interprets the left argument much like a `sprintf()`-style format string to be applied to the right argument, and returns the string resulting from this formatting operation. For example:

```
1 >>> import math
2 >>> print('The value of PI is approximately %5.3f.' % math.pi)
3 The value of PI is approximately 3.142.
```

More information can be found in the *printf-style String Formatting* section.

6.1.2 Reading and Writing Files

`open()` returns a *file object*, and is most commonly used with two arguments: `open(filename, mode)`.

```
1 >>> f = open('workfile', 'w')
```

The first argument is a string containing the filename. The second argument is another string containing a few characters describing the way in which the file will be used. *mode* can be `'r'` when the file will only be read, `'w'` for only writing (an existing file with the same name will be erased), and `'a'` opens the file for appending; any data written to the file is automatically added to the end. `'r+'` opens the file for both reading and writing. The *mode* argument is optional; `'r'` will be assumed if it's omitted.

Normally, files are opened in *text mode*, that means, you read and write strings from and to the file, which are encoded in a specific encoding (the default being UTF-8). `'b'` appended to the mode opens the file in *binary mode*: now the data is read and written in the form of bytes objects. This mode should be used for all files that don't contain text.

In text mode, the default when reading is to convert platform-specific line endings (`\n` on Unix, `\r\n` on Windows) to just `\n`. When writing in text mode, the default is to convert occurrences of `\n` back to platform-specific line endings. This behind-the-scenes modification to file data is fine for text files, but will corrupt binary data like that in JPEG or EXE files. Be very careful to use binary mode when reading and writing such files.

6.1.2.1 Methods of File Objects

The rest of the examples in this section will assume that a file object called `f` has already been created.

To read a file's contents, call `f.read(size)`, which reads some quantity of data and returns it as a string or bytes object. *size* is an optional numeric argument. When *size* is omitted or negative, the entire contents of the file will be read and returned; it's your problem if the file is twice as large as your machine's memory. Otherwise, at most *size* bytes are read and returned. If the end of the file has been reached, `f.read()` will return an empty string ('').

```
1 >>> f.read()
2 'This is the entire file.\n'
3 >>> f.read()
4 ''
```

`f.readline()` reads a single line from the file; a newline character (`\n`) is left at the end of the string, and is only omitted on the last line of the file if the file doesn't end in a newline. This makes the return value unambiguous; if `f.readline()` returns an empty string, the end of the file has been reached, while a blank line is represented by `\n`, a string containing only a single newline.

```
1 >>> f.readline()
2 'This is the first line of the file.\n'
3 >>> f.readline()
4 'Second line of the file\n'
5 >>> f.readline()
6 ''
```

For reading lines from a file, you can loop over the file object. This is memory efficient, fast, and leads to simple code:

```
1 >>> for line in f:
2 ...     print(line, end='')
3 ...
4 This is the first line of the file.
5 Second line of the file
```

If you want to read all the lines of a file in a list you can also use `list(f)` or `f.readlines()`.

`f.write(string)` writes the contents of *string* to the file, returning the number of characters written.

```
1 >>> f.write('This is a test\n')
2 15
```

To write something other than a string, it needs to be converted to a string first:

```
1 >>> value = ('the answer', 42)
2 >>> s = str(value)
3 >>> f.write(s)
4 18
```

`f.tell()` returns an integer giving the file object's current position in the file represented as number of bytes from the beginning of the file when in binary mode and an opaque number when in text mode.

To change the file object's position, use `f.seek(offset, from_what)`. The position is computed from adding *offset* to a reference point; the reference point is selected by the *from_what* argument. A *from_what* value of 0 measures from the beginning of the file, 1 uses the current file position, and 2 uses the end of the file as the reference point. *from_what* can be omitted and defaults to 0, using the beginning of the file as the reference point.

```
1 >>> f = open('workfile', 'rb+')
2 >>> f.write(b'0123456789abcdef')
3 16
4 >>> f.seek(5)      # Go to the 6th byte in the file
5 5
6 >>> f.read(1)
7 b'5'
8 >>> f.seek(-3, 2) # Go to the 3rd byte before the end
9 13
10 >>> f.read(1)
11 b'd'
```

In text files (those opened without a `b` in the mode string), only seeks relative to the beginning of the file are allowed (the exception being seeking to the very file end with `seek(0, 2)`) and the only valid *offset* values are those returned from the `f.tell()`, or zero. Any other *offset* value produces undefined behaviour.

When you're done with a file, call `f.close()` to close it and free up any system resources taken up by the open file. After calling `f.close()`, attempts to use the file object will automatically fail.

```
1 >>> f.close()
2 >>> f.read()
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in ?
5 ValueError: I/O operation on closed file
```

It is good practice to use the **with** keyword when dealing with file objects. This has the advantage that the file is properly closed after its suite finishes, even if an exception is raised on the way. It is also much shorter than writing equivalent **try-finally** blocks:

```
1 >>> with open('workfile', 'r') as f:
2 ...     read_data = f.read()
3 >>> f.closed
4 True
```

File objects have some additional methods, such as `isatty()` and `truncate()` which are less frequently used; consult the Library Reference for a complete guide to file objects.

6.1.2.2 Saving structured data with json

Strings can easily be written to and read from a file. Numbers take a bit more effort, since the `read()` method only returns strings, which will have to be passed to a function like `int()`, which takes a string like `'123'` and returns its numeric value 123. When you want to save more complex data types like nested lists and dictionaries, parsing and serializing by hand becomes complicated.

Rather than having users constantly writing and debugging code to save complicated data types to files, Python allows you to use the popular data interchange format called JSON (JavaScript Object Notation). The standard module called `json` can take Python data hierarchies, and convert them to string representations; this process is called *serializing*. Reconstructing the data from the string representation is called *deserializing*. Between serializing and deserializing, the string representing the object may have been stored in a file or data, or sent over a network connection to some distant machine.

Note:

The JSON format is commonly used by modern applications to allow for data exchange. Many programmers are already familiar with it, which makes it a good choice for interoperability.

If you have an object `x`, you can view its JSON string representation with a simple line of code:

```
1 >>> json.dumps([1, 'simple', 'list'])
2 '[1, "simple", "list"]'
```

Another variant of the `dumps()` function, called `dump()`, simply serializes the object to a *text file*. So if `f` is a *text file* object opened for writing, we can do this:

```
1 json.dump(x, f)
```

To decode the object again, if `f` is a *text file* object which has been opened for reading:

```
1 x = json.load(f)
```

This simple serialization technique can handle lists and dictionaries, but serializing arbitrary class instances in JSON requires a bit of extra effort. The reference for the `json` module contains an explanation of this.

6.2 Example

6.2.1 Introduction

As part of an investigation into the high price period caused by the rogue truck driver, you are to produce a plot of the price history on the day of the event. Fortunately, one of your colleagues has provided you with a library that does the plotting for you. Your task will be to extract the times and prices from a CSV file and pass this data to the plotting function.

In this task you will be writing a function that processes a CSV file. Writing your own CSV reader is a good example to practice the ideas covered in this chapter. However, don't ever do it again. It is better to use the `csv` module (Chapter 9.2) to read and write CSV files. It is a powerful module with lots of flexibility for doing this type of work. We will cover the `csv` module later in the course. For now, enjoy writing your own CSV reader!

You will be given the following files:

- `price_reader.py`

You will edit this file to solve the following tasks.

- `nsw-price.csv`

This is a comma separated value file containing the price history for the NSW region. The first line of the file is the header row, containing the names of the columns while the second line onwards contains the time in the first column and the price per MWh in the second.

- `library.py`

This file contains routines for converting the times being read (which are initially strings) to a data type that is more useful for Python (a `datetime` object). This data type is discussed in Chapter 9.1), however for this task you will only need to use the conversion function supplied in this file. It also contains a function for creating a plot of the acquired data.

6.2.2 Task 1

Complete the `main()` function to:

- open the CSV file `nsw-price.csv`;
- pass the open file to the `read_csv()` function;
- close the file; and
- pass the date and price lists to the plotting function `plot_prices()` supplied in `library.py`. The date list is the x-axis and the price list should be the y-axis.

It is your job to make `read_csv`:

- skip the header row;

- loop over the remaining rows; and
- return columns of dates (converted to `datetime` objects using the supplied `todate()` function in `library.py`) and prices (as floats).

Function Signature

`read_csv(file_obj)`

Arguments

- `file_obj`: an already opened file.

Return Value

A list of two lists - the first list will be the date (as `datetime` objects - use the supplied `todate()` in `library.py` to do this conversion for you), the second, the price (as a float).

Example usage

```
1 >>> read_csv(prices_file)
2 [[datetime.datetime(2012, 3, 1),
3   datetime.datetime(2012, 3, 1, 3),
4   datetime.datetime(2012, 3, 2),
5   ...],
6   [34.03,
7     36.52,
8     ...]
9 ]
```

Hint: Use the `split()` method of string objects to split them up into pieces.

```
1 >>> my_str = 'SettlementDate,Price\n'
2 >>> my_str.split(',')
3 ['date', 'price\n']
```

Hint: Use the `strip()` to remove the newline character and whitespace from the start and end of a string.

```
1 >>> my_str = 'SettlementDate,Price\n'
2 >>> my_str.strip()
3 'SettlementDate,Price'
```

6.3 Sources

- [1] Python Foundation. Official Python Tutorial, Chapter 7, 2011. <http://docs.python.org/3.3/tutorial/inputoutput.html>.

Chapter 7

When Things Go Wrong

You Will Learn

- About what Python does when it finds an error;
- How to catch errors and deal with them.

7.1 Errors and Exceptions¹

Until now error messages haven't been more than mentioned, but if you have tried out the examples you have probably seen some. There are (at least) two distinguishable kinds of errors: *syntax errors* and *exceptions*.

7.1.1 Syntax Errors

Syntax errors, also known as parsing errors, are perhaps the most common kind of complaint you get while you are still learning Python:

```
1 >>> while True print('Hello world')
2     File "<stdin>", line 1, in ?
3         while True print('Hello world')
4             ^
5 SyntaxError: invalid syntax
```

The parser repeats the offending line and displays a little ‘arrow’ pointing at the earliest point in the line where the error was detected. The error is caused by (or at least detected at) the token *preceding* the arrow: in the example, the error is detected at the function **print()**, since a colon (':') is missing before it. File name and line number are printed so you know where to look in case the input came from a script.

7.1.2 Exceptions

Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. Errors detected during execution are called *exceptions* and are not unconditionally fatal: you will soon learn how to handle them in Python

programs. Most exceptions are not handled by programs, however, and result in error messages as shown here:

```
1 >>> 10 * (1/0)
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in ?
4 ZeroDivisionError: division by zero
5 >>> 4 + spam*3
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in ?
8 NameError: name 'spam' is not defined
9 >>> '2' + 2
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in ?
12 TypeError: Can't convert 'int' object to str implicitly
```

The last line of the error message indicates what happened. Exceptions come in different types, and the type is printed as part of the message: the types in the example are **ZeroDivisionError**, **NameError** and **TypeError**. The string printed as the exception type is the name of the built-in exception that occurred. This is true for all built-in exceptions, but need not be true for user-defined exceptions (although it is a useful convention). Standard exception names are built-in identifiers (not reserved keywords).

The rest of the line provides detail based on the type of exception and what caused it.

The preceding part of the error message shows the context where the exception happened, in the form of a stack traceback. In general it contains a stack traceback listing source lines; however, it will not display lines read from standard input.

Built-in Exceptions lists the built-in exceptions and their meanings.

7.1.3 Handling Exceptions

It is possible to write programs that handle selected exceptions. Look at the following example, which asks the user for input until a valid integer has been entered, but allows the user to interrupt the program (using **Control-C** or whatever the operating system supports); note that a user-generated interruption is signalled by raising the **KeyboardInterrupt** exception.

```
1 >>> while True:
2 ...     try:
3 ...         x = int(input("Please enter a number: "))
4 ...         break
5 ...     except ValueError:
6 ...         print("Oops! That was no valid number. Try again...")
7 ...
```

The **try** statement works as follows.

- First, the *try clause* (the statement(s) between the **try** and **except** keywords) is executed.

- If no exception occurs, the *except clause* is skipped and execution of the **try** statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped. Then if its type matches the exception named after the **except** keyword, the except clause is executed, and then execution continues after the **try** statement.
- If an exception occurs which does not match the exception named in the except clause, it is passed on to outer **try** statements; if no handler is found, it is an *unhandled exception* and execution stops with a message as shown above.

A **try** statement may have more than one except clause, to specify handlers for different exceptions. At most one handler will be executed. Handlers only handle exceptions that occur in the corresponding try clause, not in other handlers of the same **try** statement. An except clause may name multiple exceptions as a parenthesized tuple, for example:

```
1 ... except (RuntimeError, TypeError, NameError):
2 ...     pass
```

The last except clause may omit the exception name(s), to serve as a wildcard. Use this with extreme caution, since it is easy to mask a real programming error in this way! It can also be used to print an error message and then re-raise the exception (allowing a caller to handle the exception as well):

```
1 import sys
2
3 try:
4     f = open('myfile.txt')
5     s = f.readline()
6     i = int(s.strip())
7 except IOError as err:
8     print("I/O error: {0}".format(err))
9 except ValueError:
10    print("Could not convert data to an integer.")
11 except:
12    print("Unexpected error:", sys.exc_info()[0])
13    raise
```

The **try ... except** statement has an optional *else clause*, which, when present, must follow all except clauses. It is useful for code that must be executed if the try clause does not raise an exception. For example:

```
1 for arg in sys.argv[1:]:
2     try:
3         f = open(arg, 'r')
4     except IOError:
5         print('cannot open', arg)
6     else:
7         print(arg, 'has', len(f.readlines()), 'lines')
8         f.close()
```

The use of the **else** clause is better than adding additional code to the **try** clause because it avoids accidentally catching an exception that wasn't raised by the code being protected by the **try ... except** statement.

When an exception occurs, it may have an associated value, also known as the exception's *argument*. The presence and type of the argument depend on the exception type.

The **except** clause may specify a variable after the exception name. The variable is bound to an exception instance with the arguments stored in **instance.args**. For convenience, the exception instance defines **__str__()** so the arguments can be printed directly without having to reference **.args**. One may also instantiate an exception first before raising it and add any attributes to it as desired.

```
1 >>> try:
2 ...     raise Exception('spam', 'eggs')
3 ... except Exception as inst:
4 ...     print(type(inst))    # the exception instance
5 ...     print(inst.args)    # arguments stored in .args
6 ...     print(inst)         # __str__ allows args to be printed directly,
7 ...                         # but may be overridden in exception subclasses
8 ...     x, y = inst.args    # unpack args
9 ...     print('x =', x)
10 ...    print('y =', y)
11 ...
12 <class 'Exception'>
13 ('spam', 'eggs')
14 ('spam', 'eggs')
15 x = spam
16 y = eggs
```

If an exception has arguments, they are printed as the last part ('detail') of the message for unhandled exceptions.

Exception handlers don't just handle exceptions if they occur immediately in the **try** clause, but also if they occur inside functions that are called (even indirectly) in the **try** clause. For example:

```
1 >>> def this_fails():
2 ...     x = 1/0
3 ...
4 >>> try:
5 ...     this_fails()
6 ... except ZeroDivisionError as err:
7 ...     print('Handling run-time error:', err)
8 ...
9 Handling run-time error: int division or modulo by zero
```

7.1.4 Raising Exceptions

The **raise** statement allows the programmer to force a specified exception to occur. For example:

```
1 >>> raise NameError('HiThere')
2 Traceback (most recent call last):
3   File "<stdin>", line 1, in ?
4 NameError: HiThere
```

The sole argument to **raise** indicates the exception to be raised. This must be either an exception instance or an exception class (a class that derives from **Exception**).

If you need to determine whether an exception was raised but don't intend to handle it, a simpler form of the **raise** statement allows you to re-raise the exception:

```
1 >>> try:
2 ...     raise NameError('HiThere')
3 ... except NameError:
4 ...     print('An exception flew by!')
5 ...     raise
6 ...
7 An exception flew by!
8 Traceback (most recent call last):
9   File "<stdin>", line 2, in ?
10 NameError: HiThere
```

7.1.5 User-defined Exceptions

Programs may name their own exceptions by creating a new exception class (see *Classes* for more about Python classes). Exceptions should typically be derived from the **Exception** class, either directly or indirectly. For example:

```
1 >>> class MyError(Exception):
2 ...     def __init__(self, value):
3 ...         self.value = value
4 ...     def __str__(self):
5 ...         return repr(self.value)
6 ...
7 >>> try:
8 ...     raise MyError(2*2)
9 ... except MyError as e:
10 ...     print('My exception occurred, value:', e.value)
11 ...
12 My exception occurred, value: 4
13 >>> raise MyError('oops!')
14 Traceback (most recent call last):
15   File "<stdin>", line 1, in ?
16 __main__.MyError: 'oops!'
```

In this example, the default `__init__()` of **Exception** has been overridden. The new behavior simply creates the *value* attribute. This replaces the default behavior of creating the *args* attribute.

Exception classes can be defined which do anything any other class can do, but are usually kept simple, often only offering a number of attributes that allow information about the error to be extracted by handlers for the exception. When creating a module

that can raise several distinct errors, a common practice is to create a base class for exceptions defined by that module, and subclass that to create specific exception classes for different error conditions:

```
1 class Error(Exception):
2     """Base class for exceptions in this module."""
3     pass
4
5 class InputError(Error):
6     """Exception raised for errors in the input.
7
8     Attributes:
9         expression -- input expression in which the error occurred
10        message -- explanation of the error
11    """
12
13    def __init__(self, expression, message):
14        self.expression = expression
15        self.message = message
16
17 class TransitionError(Error):
18     """Raised when an operation attempts a state transition that's not
19    allowed.
20
21    Attributes:
22        previous -- state at beginning of transition
23        next -- attempted new state
24        message -- explanation of why the specific transition is not allowed
25    """
26
27    def __init__(self, previous, next, message):
28        self.previous = previous
29        self.next = next
30        self.message = message
```

Most exceptions are defined with names that end in “Error,” similar to the naming of the standard exceptions.

Many standard modules define their own exceptions to report errors that may occur in functions they define. More information on classes is presented in chapter *Classes*.

7.1.6 Defining Clean-up Actions

The **try** statement has another optional clause which is intended to define clean-up actions that must be executed under all circumstances. For example:

```
1 >>> try:
2 ...     raise KeyboardInterrupt
3 ... finally:
4 ...     print('Goodbye, world!')
5 ...
6 Goodbye, world!
7 KeyboardInterrupt
```

A *finally clause* is always executed before leaving the **try** statement, whether an exception has occurred or not. When an exception has occurred in the **try** clause and has not been handled by an **except** clause (or it has occurred in a **except** or **else** clause), it is re-raised after the **finally** clause has been executed. The **finally** clause is also executed “on the way out” when any other clause of the **try** statement is left via a **break**, **continue** or **return** statement. A more complicated example:

```
1 >>> def divide(x, y):
2 ...     try:
3 ...         result = x / y
4 ...     except ZeroDivisionError:
5 ...         print("division by zero!")
6 ...     else:
7 ...         print("result is", result)
8 ...     finally:
9 ...         print("executing finally clause")
10 ...
11 >>> divide(2, 1)
12 result is 2.0
13 executing finally clause
14 >>> divide(2, 0)
15 division by zero!
16 executing finally clause
17 >>> divide("2", "1")
18 executing finally clause
19 Traceback (most recent call last):
20   File "<stdin>", line 1, in ?
21   File "<stdin>", line 3, in divide
22 TypeError: unsupported operand type(s) for /: 'str' and 'str'
```

As you can see, the **finally** clause is executed in any event. The **TypeError** raised by dividing two strings is not handled by the **except** clause and therefore re-raised after the **finally** clause has been executed.

In real world applications, the **finally** clause is useful for releasing external resources (such as files or network connections), regardless of whether the use of the resource was successful.

7.1.7 Predefined Clean-up Actions

Some objects define standard clean-up actions to be undertaken when the object is no longer needed, regardless of whether or not the operation using the object succeeded or failed. Look at the following example, which tries to open a file and print its contents to the screen.

```
1 for line in open("myfile.txt"):
2     print(line, end="")
```

The problem with this code is that it leaves the file open for an indeterminate amount of time after this part of the code has finished executing. This is not an issue in simple scripts, but can be a problem for larger applications. The **with** statement

allows objects like files to be used in a way that ensures they are always cleaned up promptly and correctly.

```
1 with open("myfile.txt") as f:
2     for line in f:
3         print(line, end="")
```

After the statement is executed, the file *f* is always closed, even if a problem was encountered while processing the lines. Objects which, like files, provide predefined clean-up actions will indicate this in their documentation.

7.2 Example

You have recently written a small program to help you calculate the ramp rate between recorded power measurements from a diesel generator.

The program asks the user to type in any number of power measurements recorded at a regular interval and the time interval in seconds.

Some example operation of your program:

```
Please enter the recorded power measurements separated by a space:
> 4.3 2.4 1.0 0.1 0.1 0
Please enter the time between power measurements in seconds:
> 30
```

```
-----
Ramp Rates:

-0.0633   |-0.0467   |-0.0300   |0.0000       |-0.0033
-----
```

One of your co-workers loves using your program. He says that it saves him at least 20 minutes a week. Lately though, he mentioned that it has been ‘crashing’ and ‘not working’. He sent you the screen shot shown in Figure 7.1 on the following page.

This is a perfect example for you to practise your exception handling skills.

You will be given the following files:

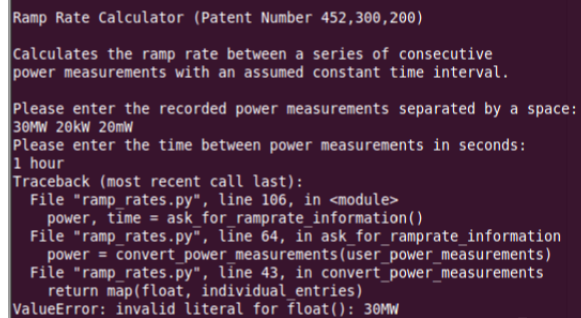
- `ramp_rates.py`

You will modify this file in order to complete this example.

- `test_ramp_rates.py`

The unit test case for this example. There is a separate test case in this file for each task. You can run the test case like any other Python file:

```
C:\Windows\py.exe -3.7-32 tests\test_ramp_rates.py
```



```
Ramp Rate Calculator (Patent Number 452,300,200)

Calculates the ramp rate between a series of consecutive
power measurements with an assumed constant time interval.

Please enter the recorded power measurements separated by a space:
30MW 20kW 20mW
Please enter the time between power measurements in seconds:
1 hour
Traceback (most recent call last):
  File "ramp_rates.py", line 106, in <module>
    power, time = ask_for_ramprate_information()
  File "ramp_rates.py", line 64, in ask_for_ramprate_information
    power = convert_power_measurements(user_power_measurements)
  File "ramp_rates.py", line 43, in convert_power_measurements
    return map(float, individual_entries)
ValueError: invalid literal for float(): 30MW
```

Figure 7.1: Screenshot of program crash.

7.2.1 Task 1

Under certain user input, the function `ask_for_ramprate_information()` may return an exception. The exception occurs in `ask_for_ramprate_information()` when `convert_power_measurements()` is called. Your job is to catch the `ValueError` exceptions that are thrown, print a useful error message to the user and then exit the program.

These types of errors occur when a user types in “10MW 9MW” instead of “10 9”.

An example solution:

```
1 try:
2     power = convert_power_measurements(user_power_measurements)
3 except ValueError:
4     print(power_measurement_error_message)
5     sys.exit(1)
```

7.2.2 Task 2

When the user enters “0” for the time interval, a `ZeroDivisionError` occurs in the program. Find and catch the `ZeroDivisionError`, print a helpful error message and exit the program when this exception occurs.

7.3 Sources

- [1] Python Foundation. Official Python Tutorial, Chapter 8, 2011. <http://docs.python.org/3.3/tutorial/errors.html>.

Chapter 8

PSS[®]E Macros and Custom Toolbar Buttons

You Will Learn

- Write your own macro file; and
- Add macro files to a PSS[®]E custom toolbar button.

8.1 What is a macro button

A Python script that runs when you click a button on the PSS[®]E custom toolbar. You can write a PSS[®]E macro to automate tasks that you do commonly. For example:

- Solve the case in a particular way;
- Create a backup file of your progress so far; and
- Show the total generation for a region.

8.2 The macro file structure

We have learned there are two ways to use a Python file.

- The first is to run it as a script by double clicking on it or by running from the command line; and
- The second is import the Python file and only use some of the functions inside of it. The Python file is used as a library in this case.

PSS[®]E treats the macro files as a script (the first method). When we click on the macro button, behind the scenes, PSS[®]E runs our Python script.

The macro file is easier to distribute to colleagues if it is self contained. Self contained means that they do not import library files you have written. The reasoning is that there is no guarantee that our colleagues have installed our hand-written library files in the same location that we have, and thus they will not be able to run our code. Only import the modules provided by PSS[®]E (like `psspy`), those in the Python

standard library (like the `csv` module) and those you know are already installed (like `matplotlib`).

Here we present to you a simple structure that should serve as a good starting point across a broad category of uses.

```
1 import psspy
2 psspy.setThrowPsseExceptions(True)
3
4 def macro():
5     psspy.fns1(options1=0)
6     psspy.fns1(options1=1)
7     supporting_function("an example supporting function")
8
9 def supporting_function(message):
10     print(message)
11
12 macro()
```

Our structure consists of the definition for the main `macro()` function. The `macro()` function is then called at the end of the file. The `macro()` function contains the main logic for our macro. Supporting functions can be placed below the definition and above where it is called.

8.3 Adding the macro button to PSS®E

Here is the procedure for adding new macro buttons to the custom toolbar:

Check that the custom toolbar is enabled within PSS®E

- Visit *Tools*
- Then *Customize Toolbars..*
- Then scroll down and make sure *Custom* is checked

Add your macro button to the custom toolbar

- Visit *Tools*
- Then *Configure Customize Toolbar Buttons*
- Near the bottom select the *Full Pathname of File*. Browse to find your Python macro file.
- Describe your button in the *Text to be Displayed* field. This description will pop up when you hover over the button in PSS®E
- Click *Update*. Do not click *Close*!
- Then finally click on the *right arrow* button to move your *Available Custom Toolbar Button* to the *Active Buttons* side.

8.4 Example

8.4.1 Introduction

You'd like to further investigate DieselCo's claims that adding small DieselCo generators throughout the region would have prevented voltage collapse. You will be required to add many little diesel generators to the network as part of your investigation. All of the little diesel generators will be of the same make. This is a good opportunity to create a macro button that adds a diesel generator to a bus.

The DieselCo website www.dieselco.co has a product statement for their flagship diesel generator; the DGFIFTY. It is a 5 MW (and 5.3 MVA) unit with maximum reactive power output of 5 MVar and no capacity to absorb reactive power. The reactive component of the source impedance is 0.2j. Leave the plant scheduled voltage and controlled bus at their default values.

You will be given the following files:

- `nsw.sav`

This is a model of the NSW network that you should be familiar with. We will be using this network throughout our examples as we continue to study the voltage collapse incident.

- `diesel_gen_macro.py`

This is an example macro file that already has helpful functions to ask the user for a bus number. Your job is to write functions to add a diesel generator. Then you must add the macro file to your PSS[®]E custom toolbar.

You will create a macro button

You will add the completed `diesel_gen_macro.py` file to the custom toolbar in PSS[®]E and add generators to the following buses:

- Armidale - bus number 20070
- Coffs Harbour - bus number 22142
- Lismore - bus number 20450

8.4.2 Task 1

Add the `diesel_gen_macro.py` file to the custom toolbar in PSS[®]E.

Run the macro. You will be prompted for a bus number to add a diesel generator to. After filling in a bus number you will see the following message printed to the PSS[®]E *progress* window:

```
1 WARNING could not add DGFIFTY at bus 20070 with id D1
```

8.4.3 Task 2

Create a function `add_diesel_gen_to_bus` that takes a bus number and a machine id. It should add a machine with the same specs as the DGFIFTY to this bus.

Function signature

`add_diesel_gen_to_bus(busnum, machine_id)`

Arguments

- *busnum* - bus number to add generator to
- *machine_id* - add generator with this machine id

Example usage

```
1 >>> add_diesel_gen_to_bus(20070, 'D1')
2 MACHINE "D1" AT BUS 20070 [ARM_330KV 330.00] ADDED.
3 POWER FLOW DATA ITEMS WITH NON-DEFAULT VALUES:
4 X--DEFAULT---X X---ACTUAL---X DATA ITEM
5 0.00000 0.100000 PG
6 0.00000 0.100000 QG
7 9999.00 0.100000 QT
8 -9999.00 0.00000 QB
9 9999.00 0.300000 PT
10 -9999.00 0.00000 PB
11 100.000 0.300000 MBASE
12 1.00000 0.200000 ZX
```

8.4.4 Task 3

Use your new macro button to add generators to these three places:

- Armidale - bus number 20070
- Coffs Harbour - bus number 22142
- Lismore - bus number 20450

Chapter 9

Standard Library Modules

You Will Learn

- CSV manipulation;
- How to wrangle the most mischievous of files;
- Confidently use time and date objects;

9.1 `datetime`

The `datetime` module provides a collection of classes that specialize in manipulating times and dates for formatting and output. The `time` module provides some similar functionality but requires all times to be converted to the number of seconds since epoch. Unless you are really good at estimating how many seconds have passed since January 1st, 1970, `datetime` provides a simpler method for defining times using units such as years, months, days, hours, minutes and microseconds.

`datetime` provides five key datatypes covering three specific areas relating to the representation of dates and times. Here they are, grouped by their specialty.

Representing dates and times:

- `time`: stores times only;
- `date`: stores dates only;
- `datetime`: stores both dates and times.

The difference between two times:

- `timedelta`

Specifying the timezone:

- `tzinfo`: the `datetime` and `time` functions can be made aware of timezones by attaching a `tzinfo` object to the `time` or `datetime` object. An object that has a `tzinfo` object attached to it can be converted to a different timezone by changing the `tzinfo` to a new timezone.

Timezones are not covered in this chapter. Working without a timezone is sufficient for most tasks. Refer to <http://docs.python.org/3.3/library/datetime.html#tzinfo-objects> if you require more information on this topic.

9.1.1 Creating **time**, **date** and **datetime** Objects

Each object can be created by calling the appropriate class instantiation method from the **datetime** module, as follows. Once an object has been created, it is immutable. Any operations which look like they change an attribute of one of these objects, is actually returning a new object of the same type.

All arguments may be defined using keywords. This is particularly useful when defining a few of the optional arguments.

9.1.1.1 **datetime** Objects³

`datetime.datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]])`

- The *year*, *month* and *day* arguments are required.
- *tzinfo* may be `None`, or an instance of a **tzinfo** subclass.
- The remaining arguments may be ints or longs, in the following ranges:
 - `MINYEAR <= year <= MAXYEAR`
 - (`MINYEAR` and `MAXYEAR` are constants defined in the **datetime** module. Type `datetime.MINYEAR` or `datetime.MAXYEAR` to return these values.)
 - `1 <= month <= 12`
 - `1 <= day <= number of days in the given month and year`
 - `0 <= hour < 24`
 - `0 <= minute < 60`
 - `0 <= second < 60`
 - `0 <= microsecond < 1000000`
- If an argument outside those ranges is given, **ValueError** is raised.

datetime objects can also be created using the following constructors. Only a selection of the more useful ones are given here. Refer to the Python Standard Library reference a complete listing.

- `datetime.datetime.today()`
Return the current local date and time, with `tzinfo = None`.
- `datetime.datetime.combine(date, time)`
Return a new **datetime** object whose date components are equal to the given *date* object's, and whose time components and `tzinfo` attributes are equal to the given *time* object's.

For any **datetime** object *d*, `d == datetime.combine(d.date(), d.timetz())`. If *date* is a **datetime** object, its time components and `tzinfo` attributes are ignored.

- `datetime.datetime.strptime(date_string, format)`

Return a `datetime` object corresponding to *date_string*, parsed according to formatting directive, *format*. This is equivalent to passing the output of the `time` module's `strptime()` function to the `datetime` specifier:

```
1 datetime(*(time.strptime(date_string, format)[0:6]))
```

`ValueError` is raised if *date_string* and *format* can't be parsed by `time.strptime()` or if it returns a value which isn't a time tuple. This method is an important function and will be covered in greater detail later in this Section.

9.1.1.2 time Objects³

`datetime.time([hour[, minute[, second[, microsecond[, tzinfo]]]])`

- All arguments are optional.
- *tzinfo* may be `None`, or an instance of a `tzinfo` subclass. The remaining arguments may be ints or longs, in the following ranges:
 - `0 <= hour < 24`
 - `0 <= minute < 60`
 - `0 <= second < 60`
 - `0 <= microsecond < 1000000`
- If an argument outside these ranges is given, `ValueError` is raised. All default to 0 except *tzinfo*, which defaults to `None`.

`time` objects can also be created using the following methods. Only a selection of the more useful ones are given here. Refer to the Python standard library reference a complete listing.

- `datetime.time.replace([hour[, minute[, second[, microsecond[, tzinfo]]]])` Given a `time` object, returns another `time` object with the original time augmented with the new data.

```
1 >>> aTime = datetime.time(hour=12,minute=6)
2 >>> aTime
3 datetime.time(12,6)
4 >>> aTime.replace(hour=5)
5 datetime.time(5,6)      # A new datetime.time object.
```

`time` objects can also be derived from a `datetime` object using `datetime`'s `time()` method:

- `datetime.datetime.time()`

Return only the time of the `datetime` object in a new time object.

```
1 >>> party_time = datetime.datetime(2000,1,1,0,1)
2 >>> party_time.time()      # return a datetime.time object.
3 datetime.time(0,1)
```

9.1.1.3 date Objects³

`datetime.date(year, month, day)`

- All arguments are required.
- Arguments may be ints or longs, in the following ranges:
 - `MINYEAR <= year <= MAXYEAR` (`MINYEAR` and `MAXYEAR` are constants defined in the `datetime` module. Type `datetime.MINYEAR` or `datetime.MAXYEAR` to return these values.)
 - `1 <= month <= 12`
 - `1 <= day <= number of days in the given month and year`
- If an argument outside those ranges is given, `ValueError` is raised.

`date` objects can also be created using the following methods. Only a selection of the more useful ones are given here. Refer to the Python standard library reference a complete listing.

`datetime.date.today()`

Return the current local date.

`datetime.date.replace([year[, month[, day]])`

Given a `date` object, returns another `date` object with the original date augmented with the new data.

```
1 >>> aDate = datetime.date(2001,12,6)
2 >>> aDate
3 datetime.date(2001,12,6)
4 >>> aDate.replace(day=26)
5 datetime.time(2001,12,26)      # A new datetime.date object.
```

`date` objects can also be derived from a `datetime` object using `datetime`'s `date()` method.

`datetime.datetime.date()`

Return only the date of the `datetime` object in a new `date` object.

```
1 >>> party_time = datetime.datetime(2000,1,1,0,1)
2 >>> party_time.date()      # return a datetime.date object.
3 datetime.date(2000,1,1)
```

Accessing the Attributes of the `time`, `date` and `datetime` Objects

Attributes of the `time`, `date` and `datetime` objects can be accessed using the familiar 'dot notation'. The available attributes are the same as the keyword arguments when creating the object: `year`, `month`, `day`, `hour`, `minute`, `second` and `microsecond`. All of the attributes are accessible when using a `datetime` object while only the relevant subset is available for the `date` and `time` objects. The following example shows how these attributes are accessed from a `datetime` object, `party_time`, and shows the range of values the attributes may hold.

```
1 >>> party_time = datetime.datetime(2000,1,1,0,12,36,1001)
2 >>> party_time.year      # Between 1 and 9999 inclusive.
3 2000
4 >>> party_time.month     # Between 1 and 12 inclusive.
5 1
6 >>> party_time.day       # Between 1 and the number of days in the given
7 ...                     # month of the given year.
8 1
9 >>> party_time.hour      # In range(24).
10 0
11 >>> party_time.minute    # In range(60).
12 12
13 >>> party_time.second     # In range(60).
14 36
15 >>> party_time.microsecond # In range(1000000).
16 1001
```

In addition to these raw attributes, each object type provides its own set of helper functions to return useful variants of these attributes. For instance, `date.weekday()` returns an integer corresponding to a day of the week (0 is Monday, 1 is Tuesday etc.). Refer to the `datetime` module, <http://docs.python.org/3.3/library/datetime.html>, in the Python Standard Library for other similar functions.

Creating `timedelta` Objects

A `timedelta`, as the name suggests, is an object for storing time differences. There are two ways of creating `timedelta` objects. The first is by defining a `timedelta` object by specifying the size of the time difference in days, seconds, microseconds, milliseconds, hours and weeks, as discussed later. The second way of creating a `timedelta` object

is by taking the difference between two `date` or `datetime` objects (date arithmetic is discussed in the following section).

`datetime.timedelta([days[, seconds[, microseconds[, minutes[, hours[, weeks]]]]])`

- All arguments are optional and default to 0. Arguments may be ints, longs, or floats, and may be positive or negative. Only days, seconds and microseconds are stored internally. Arguments are converted to those units:
 - A millisecond is converted to 1000 microseconds.
 - A minute is converted to 60 seconds.
 - An hour is converted to 3600 seconds.
 - A week is converted to 7 days.
- and days, seconds and microseconds are then normalized so that the representation is unique, with
 - `0 <= microseconds < 1000000`
 - `0 <= seconds < 3600*24` (the number of seconds in one day)
 - `-999999999 <= days <= 999999999`
- If any argument is a float and there are fractional microseconds, the fractional microseconds left over from all arguments are combined and their sum is rounded to the nearest microsecond. If no argument is a float, the conversion and normalization processes are exact (no information is lost). If the normalized value of `days` lies outside the indicated range, `OverflowError` is raised.
- Note that normalization of negative values may be surprising at first. For example,

```
1 >>> day_and_second = datetime.timedelta(days=1, seconds=1)
2 >>> day_and_second
3 datetime.timedelta(1, 1)
4 >>> -day_and_second # (-2 days + 86399 seconds) = -(1day, 1 second)
5 datetime.timedelta(-2, 86399)
```

9.1.2 Date Arithmetic

Of the three classes that represent dates and times, only `date` and `datetime` can be used in arithmetic expressions. An attempt to perform addition or subtraction on a `time` object will result in a `TypeError`.

The supported operations on `date` and `datetime` are as follows. Note that taking the difference between two `datetime` or `date` objects creates a `timedelta` object.

<code>date = date + timedelta</code>	<code>datetime = datetime + timedelta</code>
<code>date = date - timedelta</code>	<code>datetime = datetime - timedelta</code>
<code>timedelta = date - date</code>	<code>timedelta = datetime - datetime</code>

It is not possible to mix `datetimes` and `dates` in the same expression; attempting to do so will result in a `TypeError`. However, it is possible to use the `timedelta` produced by taking the difference between one type, two `datetimes` for instance, and add this `timedelta` to a type different to the one it was derived from, `date` in this case. Only the `days` attribute of the `timedelta` object is used in an addition or subtraction operation between a `timedelta` and a `date`. Seconds are not rounded, simply ignored.

As `timedelta` objects represent a time difference and not a fixed date, they possess a greater flexibility when it comes to arithmetic. Addition and subtraction between two `timedelta` objects is possible, just like `date` and `datetime`. However, unlike the other types, negative values are valid, as is multiplication and division by integers and, new as of Python 3.2, floats. Similarly new since Python 3.2 is the ability to divide one `timedelta` object by another `timedelta` object.

```
1 >>> -datetime.timedelta(days=1)
2 datetime.timedelta(-1)
3 >>> datetime.timedelta(days=1) * 2
4 datetime.timedelta(2)
5 >>> datetime.timedelta(days=1) / 2
6 datetime.timedelta(0, 43200)
7 >>> datetime.timedelta(days=1) / datetime.timedelta(days=5)
8 0.2
```

9.1.3 Comparing Times and Dates

Comparisons are possible for all the temporal objects but can only be conducted using like datatypes: `datetimes` can only be compared to other `datetimes`, `timedeltas` can only be compared to other `timedeltas`, and so on. Comparisons can be made with the greater than, less than or equal operators and work as expected: a date or time preceding another is considered smaller.

```
1 >>> now = datetime.datetime.now()
2 >>> now_plus_day = now + datetime.timedelta(day=1)
3 >>> now < now_plus_day
4 True
```

As `timedelta` objects can be negative, the most positive `timedelta` object is considered the greatest, just as in normal comparisons. Comparison of the magnitude of two `timedeltas` can be carried out in the same way as you would for numeric data: using the `abs()` function.

```
1 >>> day_and_second = datetime.timedelta(days=1, seconds=1)
2 >>> abs(-day_and_second) == day_and_second
3 True
```

9.1.4 Formatted Output of `time`, `date` and `datetime` Objects

By default, printing an object from the `datetime` module results in a stringified representation of the object.

```
1 >>> party_time = datetime.datetime(2000,1,1,0,1)
2 >>> print(party_time)
3 2000-01-01 00:01:00
4 >>> print(party_time.date())
5 2000-01-01
6 >>> print(party_time.time())
7 00:01:00
8 >>> print(datetime.timedelta(days=1))
9 1 day, 0:00:00
```

In addition to the automatic formatting, `date`, `time` and `datetime` objects have access to the formatting function `strftime()`. `strftime()` can be called as a method of one of the `datetime` objects, as follows:

```
1 datetime.datetime.strftime(format)
```

This method takes a format string, consisting of directives, which dictates the formatting of the returned date and time string. A list of directives and their effect can be found in Table 9.1 on page 114. If a directive which is not applicable to the type being written is issued (a year directive (`%Y`) is specified when printing a `time` object), the smallest value for that field is substituted in the string.

Example usage of the `strftime()` method using a `time` object:

```
1 >>> aTime = datetime.time(13,10)
2 >>> aTime.strftime('%I:%M %p')
3 '01:10 PM'
4 >>> aTime.replace(hour=10).strftime('%I:%M %p')
5 '10:10 AM'
6 >>> aTime.strftime('%Y')      # No year stored in a time object
7 '1900'
```

9.1.5 Reading Time From a String

The `strptime()` can be thought of performing the inverse function of the `strftime()` function. It uses the same format directives to mark the location of the relevant units of time within the string to be read and returns corresponding `datetime` object. This method is only available to `datetime` objects or as:

```
1 datetime.datetime.strptime(date_string,format)
```

Unspecified units of time default to the smallest value for that unit. A list of the format directives and their purpose can be found in Table 9.1 on page 114.

```
1 >>> datetime.datetime.strptime('10:10 AM', '%I:%M %p')
2 datetime.datetime(1900, 1, 1, 10, 10)
3 >>> datetime.datetime.strptime('01:10 PM', '%I:%M %p')
4 datetime.datetime(1900, 1, 1, 13, 10)
```

Table 9.1: `strptime()` and `strftime()` directives.

Directive	Meaning
%a	Locale's abbreviated weekday name.
%A	Locale's full weekday name.
%b	Locale's abbreviated month name.
%B	Locale's full month name.
%c	Locale's appropriate date and time representation.
%d	Day of the month as a decimal number [01,31].
%f	Microsecond as a decimal number [0,999999], zero-padded on the left.
%H	Hour (24-hour clock) as a decimal number [00,23].
%I	Hour (12-hour clock) as a decimal number [01,12].
%j	Day of the year as a decimal number [001,366].
%m	Month as a decimal number [01,12].
%M	Minute as a decimal number [00,59].
%p	Locale's equivalent of either AM or PM.
%S	Second as a decimal number [00,61].
%U	Week number of the year (Sunday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Sunday are considered to be in week 0.
%w	Weekday as a decimal number [0(Sunday),6].
%W	Week number of the year (Monday as the first day of the week) as a decimal number [00,53]. All days in a new year preceding the first Monday are considered to be in week 0.
%x	Locale's appropriate date representation.
%X	Locale's appropriate time representation.
%y	Year without century as a decimal number [00,99].
%Y	Year with century as a decimal number.
%z	UTC offset in the form +HHMM or -HHMM (empty string if the object is naive).
%Z	Time zone name (empty string if the object is naive).
%%	A literal '%' character.

9.2 csv

9.2.1 Introduction²

The so-called CSV (Comma Separated Values) format is the most common import and export format for spreadsheets and databases. There is no “CSV standard” so the format is operationally defined by the many applications which read and write it. The lack of a standard means that subtle differences often exist in the data produced and consumed by different applications. These differences can make it annoying to process CSV files from multiple sources. Still, while the delimiters and quoting characters vary, the overall format is similar enough that it is possible to write a single module which can efficiently manipulate such data, hiding the details of reading and writing the data from the programmer.

The CSV module implements classes to read and write tabular data in CSV format. It allows programmers to say, “write this data in the format preferred by Excel,” or “read data from this file which was generated by Excel,” without knowing the precise details of the CSV format used by Excel. Programmers can also describe the CSV formats understood by other applications or define their own special-purpose CSV formats.

9.2.2 reader Objects

As the name of the object suggests, the purpose of a **reader** object is to read CSV files. Once created, the **reader** object is used to easily access the data stored in the CSV file. The Python Standard Library outlines the way to create a reader object and its behaviour as follows.

`csv.reader(csvfile[, dialect='excel'][, fmtparam])`²

Return a **reader** object which will iterate over lines in the given *csvfile*. If *csvfile* is a file object, it must be opened with the `'newline=''` setting so that end of line characters do not get mangled¹. An optional *dialect* parameter can be given which is used to define a set of parameters specific to a particular CSV *dialect* (discussed later). The other optional *fmtparam* keyword arguments can be given to override individual formatting parameters in the current *dialect*. For full details about the *dialect* and formatting parameters, see Section 9.2.6 on page 122.

Here is a short example illustrating the simplest way of using **reader**. Each row read from the CSV file is returned as a list of strings. No automatic data type conversion is performed (so if you are expecting the values of one column to be integers, you will have to use `int()` on that value to convert it to an integer before attempting to use it as such).

```

1 >>> import csv
2 >>> spamReader = csv.reader(open('eggs.csv', 'r', newline=''))
3 >>> for row in spamReader:
4 ...     print(', '.join(row))

```

¹If `newline=''` is not specified, newlines embedded inside quoted fields will not be interpreted correctly, and on platforms that use `\r\n` line endings on write an extra `\r` will be added. It should always be safe to specify `newline=''`, since the csv module does its own (universal) newline handling.

```
5 Spam, Spam, Spam, Spam, Spam, Baked Beans
6 Spam, Lovely Spam, Wonderful Spam
```

9.2.2.1 Processing the Data

For CSV files with regular data in them (regular in this context means each row has the same number of columns), there are several methods to read the data into Python. The simplest methods result in a list of lists, similar to how the data was found in the file.

```
1 >>> contents = list(reader)
```

will produce the same result as:

```
1 >>> contents = [ ]
2 >>> for row in csvreader:
3 ...     contents.append(row)
```

A potential problem with this approach is that the data is still stored in rows. The data may be easier to work with when stored in columns.

The function `zip()` can be used as a quick one liner for transposing a list of lists. `zip()` takes the i^{th} element from each sequence that is passed to it, groups them in a tuple in the order they appeared in the `zip()` call and returns an iterator². The `*` symbol expands a list of lists, `contents` in this example, so that the individual lists (the rows) of `contents` are passed in as separate arguments to `zip()`. The net effect of these two processes is transposing a lists of rows in to a list of columns:

```
1 >>> contents_in_columns = zip(*contents)
2 >>> print(list(contents_in_columns))
```

It is also possible to combine it with multiple assignment to break each column into its own variable:

```
1 >>> firstnames, lastnames = list(zip(*contents)) # 'contents' only contains 2 columns
```

The previous examples have illustrated how to import the data as it appeared in the CSV files. The data was read into Python as strings and no processing of the data was conducted. By looping over the `reader` object, we can process the data a row at a time.

²An iterator is a sequence object that can be looped over. In this case, each time you loop over the `zip` object, it will return the next group of i^{th} elements. See Section 2.3 on page 27 for more information on the use of `zip()`.

```

1 >>> first_name_list = [ ]
2 >>> last_name_list = [ ]
3 >>> ages = [ ]
4 >>> for row in reader:
5 ...     first_name, last_name, age = row
6 ...     if (last_name in last_name_list) and (first_name in first_name_list):
7 ...         continue # name is a repeat, skip it and continue
8 ...     first_name_list.append(first_name)
9 ...     last_name_list.append(last_name)
10 ...     ages.append(int(age)) # convert the age to an integer

```

This example reads in a CSV file which stored some personal data about a small group of people. It checked for duplicates and skipped them if they were detected. Finally, it converted the third column, `age`, from the string it was read in as, to an integer.

An alternative to type converting the data as it is read in is to convert the data once the entire file has been read in. This method assumes that the data being converted resides in its own list, separate from the other columns. A column of data can be converted using either of the following one-liners, where `int()` should be replaced with an appropriate conversion depending on the data being read in. For instance, if a floating point number, '1.04', is being read in, it would be appropriate to use the `float()` function to convert the data to the correct data type.

```

1 >>> ages_str = ['1','2','3']
2 >>> ages1 = list(map(int, ages_str)) # method 1
3 >>> ages1
4 [1,2,3]
5 >>> ages2 = [int(age) for age in ages_str] # method 2
6 >>> ages2
7 [1,2,3]

```

9.2.3 writer Objects

As the name of the object suggests, the purpose of a **writer** object is to write data to CSV files. Once created, the **writer** object is used to write the data that is passed to it to the specified file. The Python Standard Library outlines the way to create a **writer** object and its behaviour as follows.

`csv.writer(csvfile[, dialect='excel'], fmtparam)`²

Return a **writer** object responsible for converting the user's data into delimited strings on the given file-like object. *csvfile* can be any object with a `write()` method. If *csvfile* is a file object, it must be opened with the `'newline=''` flag so that end of line characters do not get mangled³. An optional `dialect` parameter can be given which is used to define a set of parameters specific to a particular CSV dialect. The other optional *fmtparam* keyword arguments can be given to

³If `newline=''` is not specified, newlines embedded inside quoted fields will not be interpreted correctly, and on platforms that use `\r\n` lineendings on write an extra `\r` will be added. It should always be safe to specify `newline=''`, since the csv module does its own (universal) newline handling.

override individual formatting parameters in the current `dialect`. For full details about the `dialect` and formatting parameters, see Section 9.2.6 on page 122. All non-string data are stringified with `str()` before being written.

A short usage example which does the inverse of the `reader` example. This snippet of code could be used to generate the csv file which the reader example takes as an input:

```
1 >>> import csv
2 >>> spamWriter = csv.writer(open('eggs.csv', 'w', newline=''))
3 >>> spamWriter.writerow(['Spam'] * 5 + ['Baked Beans'])
4 >>> spamWriter.writerow(['Spam', 'Lovely Spam', 'Wonderful Spam'])
```

9.2.3.1 `writerow(row)`

Notice that we use the `writer` object's `writerow()` to write one row of data to the CSV file. `row` must be a sequence (list, tuple or another sequence type). The `writerow()` method takes this sequence, formats it according to the `dialect` of the `writer` object and then writes this formatted string to file.

A common misconception when using `writer` for the first time is that you can pass it a group of columns and the writer object will group the *i*th element from each column together and write the row to file. Unfortunately, `writer` does not work this way and requires more effort (but not much) to produce a CSV file. To write a row to file, the elements to be written must be collected into a sequence for the consumption of `writerow()`. The following example illustrates how to combine `writerow()` with a `for` loop to easily generate a CSV file of any length.

```
1 >>> import csv
2 >>> csv_outfile = open('powers.csv', 'w', newline='')
3 >>> power_writer = csv.writer(csv_outfile)
4 >>> for i in range(200):
5 ...     power_writer.writerow([i, i**2, i**3, i**4])
6 >>> csv_outfile.close()
```

Using `for` and `writerow()` together allows you to compose the rows as you go. In the above example, we were able to compute the powers and then write them out in the same step. This method can also be used to take several lists and form a row by taking one element from each list, as follows:

```
1 >>> import csv
2 >>> csv_outfile = open('powers.csv', 'w', newline='')
3 >>> number = range(200)
4 >>> number2 = [i**2 for i in number]
5 >>> number3 = [i**3 for i in number]
6 >>> number4 = [i**4 for i in number]
7 >>> power_writer = csv.writer(csv_outfile)
8 >>> for i, n in enumerate(number):
9 ...     power_writer.writerow([n, number2[i], number3[i], number4[i]])
10 >>> csv_outfile.close()
```

The same result can also be achieved more elegantly using the `zip()` function:

```

1 >>> for row in zip(number, number2, number3, number4):
2 ...     power_writer.writerow(row)
3 >>> csv_outfile.close()

```

`writerows()` is another method of the `writer` object which can be used to write out to a CSV file. As the name suggests this will write the block of rows which are passed to it, to a file. For example, the following produces the same output as the previous examples, but without the `for` statement.

```

1 >>> power_writer.writerows(zip(number, number2, number3, number4))
2 >>> csv_outfile.close()

```

9.2.4 Handling Headers

Headers are easily accommodated by reading or writing the header line before the data set is read from or written to the file. While the `csv` module does provide ways of automatically dealing with headers, in most cases, as the file format is generally known in advance, it is far easier to handle them explicitly.

To handle the reading of headers, the first line of the CSV file can be read by calling `next()`⁴ with the `reader` object as its argument: `next(reader)`.

```

1 >>> with open('mycsv.csv', 'r', newline='') as csv_in:
2 ...     reader = csv.reader(csv_in)
3 ...     # read the header
4 ...     header = next(reader)
5 ...     for row in reader:
6 ...         # read in the data in the file.

```

The reason for this behaviour is to do with the way Python handles files. In short, Python starts at the start and reads sequentially through the file. The only time it will read a section of the file again is if it is explicitly told to rewind and read that section again. Given that Python has not been told to rewind, `reader` will continue reading from its last position, which in this case will be the start of the second line. To emphasize this point, consider calling `list(reader)` on the `reader` object after the header has been read. This statement reads from the current position in the file to its end and returns a list of the contents.

```

1 >>> with open('mycsv.csv', 'r', newline='') as csv_in:
2 ...     reader = csv.reader(csv_in)
3 ...     headers = next(reader)
4 ...     data = list(reader)

```

Writing headers is relatively straight forward in comparison to reading headers. The header is just written before the data.

⁴The `next()` function returns the next value from an `iterator` object, or if the `iterator` is empty, a `StopIteration` exception. This is essentially how a `for` loop works behind the scenes.

```
1 >>> with open('mycsv_out.csv', 'w', newline='') as csv_out:
2 ...     writer = csv.writer(csv_out)
3 ...     writer.writerow(['heading 1', 'heading 2'])
4 ...     # write the data
5 ...     writer.writerows(data)
```

9.2.5 dialect Objects

CSV files are not a standardised format. Accordingly, people have taken it upon themselves to create many variants, only slightly different from each other, yet different enough to require specialised code to read. One of the great features of the `csv` module provided by the Python Standard Library is that it provides a mechanism for processing CSV files with different formatting, also known as dialects, with a minimum of fuss.

Dialects are a collection of the conventions that a particular CSV file uses to store its data. The `csv` module provides the `dialect` object for storing these properties. From the perspective of the `csv` module, changing only a single dialect property results in a completely new `dialect`. For instance, the delimiter used to separate the columns in most CSV files is (as the name suggests) a comma, `,`. However, despite the CSV name, other characters are used such as the tab character, `\t`. There are many points of difference between the CSV formats and accordingly, `dialects` have many properties covering a range of functionality. A list of the `dialect` properties available in the `csv` module is included at the end of this section.

It is possible to get a listing of the `dialects` that are currently registered with the `csv` module by issuing:

```
1 >>> csv.list_dialects()
2 ['excel-tab', 'excel']
```

The two `dialects` listed by the previous command are shipped with the `csv` module and are available to every Python installation. The process of registering new `dialects` with the `csv` module will be discussed later.

The `dialect` used by a `reader` or `writer` object is set when the object first defined and cannot be changed. By default, if no `dialect` is specified, the `'excel'` dialect is used. The `'excel'` dialect is arguably the most common CSV format and it is possible that you may never come across a file which requires a different `dialect`. If another `dialect` is required, it must be specified using the *dialect* keyword when the `reader` or `writer` object is being created.

```
1 >>> csv.reader(file_obj) # uses the default 'excel' dialect
2 >>> csv.reader(file_obj, dialect='excel-tab') # use the 'excel-tab' dialect
```

Passing a `dialect` to an object is the first of two ways of defining the `dialect` to be used for that object. The second method is useful when the required `dialect` to read a file differs from a registered `dialect` by only a few parameters. In this case, a base `dialect` is specified, using the *dialect* keyword and the `dialect` parameters which need to be overridden are explicitly specified when creating the `reader` or `writer`

object. The following two lines of code do the same thing: they start with the ‘`excel`’ dialect and change the delimiter from a comma to the tab character (which, perhaps unsurprisingly, makes the new dialect equivalent to the ‘`excel-tab`’ dialect). The first line takes advantage of the fact that the ‘`excel`’ dialect is the default dialect.

```
1 >>> csv.reader(file_obj, delimiter='\t')
2 >>> csv.reader(file_obj, dialect='excel', delimiter='\t')
```

Any number of dialect properties can be overridden in the one statement by specifying the keyword argument for that property and its new value.

If a dialect requires several parameters to be overridden and is used in a number of locations, it can be useful to register a new dialect with the `csv` module. This can be achieved using the `register_dialect()` method of the `csv` module. Registering a dialect shares many similarities with selecting a dialect when creating a reader or writer object. The major difference being the first argument is the name of the new dialect, in the following example, ‘`joes-special`’, rather than the CSV file being read or written. The other arguments are the same: keyword arguments specify the base dialect and the parameters which need to be overridden. In this example, ‘`excel`’ is specified as the base dialect (we didn’t really need to do this as, again, it is the default), the delimiter parameter is overridden with the ‘`|`’ character and the quoting behaviour of this object is defined as `QUOTE_MINIMAL` (see Section 9.2.6 on the following page, for a description of this parameter and others).

```
1 >>> csv.register_dialect(name [,base_dialect][,keyword_parameters_to_change])
2 >>> csv.register_dialect('joes-special', 'excel', delimiter='|', quoting=csv.QUOTE_MINIMAL)
```

Once it is registered, the new dialect appears in the list of registered dialects and can be selected the same way as the inbuilt ones.

```
1 >>> csv.list_dialects()
2 ['excel-tab', 'excel', 'joes-special']
3 >>> csv.reader(file_obj, dialect='joes-special')
```

For commonly used non-standard dialects with many unique parameters, it may be worthwhile defining the dialect once in a settings file so that it is accessible by name as soon as a settings file is imported.

```
settings.py
1 import csv
2
3 # Register a new dialect.
4 # This modifies the csv module directly, so once this file has been imported
5 # at least once, the new dialect will be accessible by any call to the csv
6 # module regardless of whether that module imported settings.py.
7 # For readability and maintainability, it is strongly suggested you do not
8 # rely on this behaviour and import settings.py in any module which makes use
9 # of the new dialect.
10
11 csv.register_dialect('joes-special', 'excel', delimiter='|',
12                    quoting=csv.QUOTE_MINIMAL)
```

9.2.6 dialect Properties

Dialect.delimiter

A one-character string used to separate fields. It defaults to ‘,’.

Dialect.doublequote

Controls how instances of `quotechar` appearing inside a field should be themselves be quoted. When `True`, the character is doubled. When `False`, the `escapechar` is used as a prefix to the `quotechar`. It defaults to `True`. On output, if `doublequote` is `False` and no `escapechar` is set, `Error` is raised if a `quotechar` is found in a field.

Dialect.escapechar

A one-character string used by the `writer` to escape the delimiter if quoting is set to `QUOTE.NONE` and the `quotechar` if `doublequote` is `False`. On reading, the `escapechar` removes any special meaning from the following character. It defaults to `None`, which disables escaping.

Dialect.lineterminator

The string used to terminate lines produced by the writer. It defaults to ‘\r\n’.

Note: The `reader` is hard-coded to recognise either ‘\r’ or ‘\n’ as end-of-line, and ignores `lineterminator`. This behavior may change in the future.

Dialect.quotechar

A one-character string used to quote fields containing special characters, such as the `delimiter` or `quotechar`, or which contain new-line characters. It defaults to ‘‘’.

Dialect.quoting

Controls when quotes should be generated by the `writer` and recognised by the `reader`. It can take on any of the `QUOTE_*` constants and defaults to `QUOTE_MINIMAL`.

csv.QUOTE_ALL

Instructs `writer` objects to quote all fields.

csv.QUOTE_MINIMAL

Instructs `writer` objects to only quote those fields which contain special characters such as `delimiter`, `quotechar` or any of the characters in `lineterminator`.

csv.QUOTE_NONNUMERIC

Instructs `writer` objects to quote all non-numeric fields.

Instructs the `reader` to convert all non-quoted fields to type `float`.

csv.QUOTE_NONE

Instructs `writer` objects to never quote fields. When the current `delimiter` occurs in output data it is preceded by the current `escapechar` character. If `escapechar` is not set, the `writer` will raise `Error` if any characters that require escaping are encountered.

Instructs `reader` to perform no special processing of quote characters.

Dialect.skipinitialspace

When True, whitespace immediately following the delimiter is ignored.

The default is False.

9.2.7 csv Errors²

An example of how to catch and informatively report errors when reading CSV files then stop execution.

```
1 import csv, sys
2 filename = 'some.csv'
3 with open(filename, 'r', newline='') as f:
4     reader = csv.reader(f)
5     try:
6         for row in reader:
7             print(row)
8     except csv.Error as e:
9         sys.exit('file {}, line {}: {}'.format(filename, reader.line_num, e))
```

9.3 `os.path`

Ever wanted to run wild through your filesystem creating and modifying files wherever you want? Or do you like to live a little more conservatively and prefer browsing through your filesystem finding only the files that you want and ignoring the rest? If you have ever wanted to do either of these things then `os.path` is the module for you!

`os.path` contains functions for common pathname manipulations. This is handy if, for instance, you want to get input from a user specified directory and write the output of your program to another. `os.path` contains functions which make implementing these file manipulations trivial.

Another key feature of `os.path` is its portability. With a little care, a program you wrote on Windows using the `os.path` module will still work on MacOS, regardless of the slight differences in how the filesystem works.

`os.path` manipulates only strings that represent plausible paths for the operating system it is running on, not actual directories and files on the system. Primarily, its role is as a convenience module which simplifies the joining and separating of strings that represent valid paths on the operating system that the program is running on.

9.3.1 Dissecting Paths

Breaking full paths down in to their constituents.

`os.split(path)`

Splits the filepath about the last directory separator (`os.sep`). Returns a tuple with the component of *path* before the last separator in the first element and the remainder in the second.

For windows, the path separator (`os.sep`) is the backslash character, `'\'`. As Python uses the backslash character for escape sequences in strings (remember the character for a new line `'\n'`), the backslash character must be escaped with another backslash, `'\\'`. Other OSes, such as Mac and Linux, use a forward slash as their path separator, `'/'`, which can be represented in a string as is.

```
1 >>> filepaths = ['c:\\stuff\\hello.py', 'c:\\stuff\\', 'c:\\stuff']
2 >>> for path in filepaths:
3 ...     print("%s" : "%s" %(path, os.path.splitext(path)))
4 ...
5 "c:\\stuff\\hello.py" : "('c:\\stuff\\hello', '.py')"
6 "c:\\stuff\\" : "('c:\\stuff\\', '')"
7 "c:\\stuff" : "('c:\\stuff', '')"
```

`os.path.basename(path)`

Returns only the component of *path* after the last directory separator. This is the same as the second element of the tuple `os.path.split()` returns.

```
1 >>> filepaths = ['c:\\stuff\\hello.py', 'c:\\stuff\\', 'c:\\stuff']
2 >>> for path in filepaths:
3 ...     print("%s" : "%s" %(path, os.path.basename(path)))
4 ...
```

```
5 "c:\stuff\hello.py" : "hello.py"
6 "c:\stuff\" : ""
7 "c:\stuff" : "stuff"
```

`os.path.dirname(path)`

Returns only the component of *path* before the last directory separator. This is the same as the first element of the tuple `os.path.split()` returns.

```
1 >>> filepaths = ['c:\\stuff\\hello.py', 'c:\\stuff\\', 'c:\\stuff']
2 >>> for path in filepaths:
3 ...     print('%s' : "%s" % (path, os.path.dirname(path)))
4 ...
5 "c:\stuff\hello.py" : "c:\stuff"
6 "c:\stuff\" : "c:\stuff"
7 "c:\stuff" : "c:\"
```

`os.path.splitext(path)`

Splits *path* about the final extension separator ('.'). Returns everything preceding the extension separator as the first element of the returned tuple, and the extension (with the extension separator) as the second.

```
1 >>> filepaths = ['hello.py', 'hello', 'c:\\stuff\\hello.py']
2 >>> for path in filepaths:
3 ...     print('%s' : "%s" % (path, os.path.splitext(path)))
4 ...
5 "hello.py" : "('hello', '.py')"
6 "hello" : "('hello', '')"
7 "c:\stuff\hello.py" : "('c:\\stuff\\hello', '.py')"
```

9.3.2 Building Paths

`os.path.join(pathstr1[, pathstr2[, ...]])`

Joins the supplied arguments using the directory separator (`os.sep`) appropriate for the host operating system. It will only add a separator between the strings if necessary. In other words, `os.path.join()` will check if the string begins or ends in a path separator and only add one if necessary.

```
1 >>> os.path.join('salutations', 'greetings', 'hello.py')
2 'salutations\\greetings\\hello.py'
3 >>> pathlist = ['salutations', 'greetings', 'hello.py']
4 >>> os.path.join(*pathlist) # expand a list containing path constituents
5 'salutations\\greetings\\hello.py'
```

`os.path.normpath(path)`

Normalise *path*. This function sanitises *path* by removing excessive path separators and redundant relative path movements.

```
1 >>> # Remove extra separator
2 ... os.path.normpath('hello\\yo.py')
3 'hello\yo.py'
4 >>> # Evaluate and remove relative path movements
5 ... os.path.normpath('hello\yo\..\goodbye\hi.py')
6 'hello\goodbye\hi.py'
```

9.3.3 Inspecting the Filesystem

In addition to path manipulations, `os.path` provides functions which can test various properties of files and directories.

`os.path.exists(path)`: return `True` if *path* actually exists.

`os.path.isdir(path)`: return `True` if *path* points to a directory.

`os.path.isfile(path)`: return `True` if *path* points to a file.

`os.path.getsize(path)`: return the size, in bytes, of *path*.

There are many other attributes, such as access time and modified time, that can be obtained using `os.path`. Refer to the Python Standard Library for a list of all the functions provided by this module.

9.4 glob¹

The `glob` module finds all the pathnames matching a specified pattern according to the rules used by the Unix shell. No tilde expansion is done, but `*`, `?`, and character ranges expressed with `[]` will be correctly matched. This is done by using the `os.listdir()` and `fnmatch.fnmatch()` functions in concert, and not by actually invoking a subshell. (For tilde and shell variable expansion, use `os.path.expanduser()` and `os.path.expandvars()`.)

`glob.glob(pathname)`

Return a possibly-empty list of path names that match *pathname*, which must be a string containing a path specification. *pathname* can be either absolute (like `/usr/src/Python-1.5/Makefile`) or relative (like `../..Tools/*/*.gif`), and can contain shell-style wildcards. Broken symlinks are included in the results (as in the shell).

`glob.iglob(pathname)`

Return an **iterator** which yields the same values as `glob()` without actually storing them all simultaneously.

For example, consider a directory containing only the following files: `1.gif`, `2.txt`, and `card.gif`. `glob()` will produce the following results. Notice how any leading components of the path are preserved.

```
1 >>> import glob
2 >>> glob.glob('./[0-9].*')
3 ['./1.gif', './2.txt']
4 >>> glob.glob('*.gif')
5 ['1.gif', 'card.gif']
6 >>> glob.glob('?.gif')
7 ['1.gif']
```

9.5 Example

9.5.1 Introduction

In testing the veracity of DieselCo's claims you will need to identify the best locations to place the generators.

You dust off Dr. Prabha Kundur's book. It seems that the ideal locations to add the generators to the network are where the voltage varies most with changes in reactive power $\frac{\delta Q}{\delta V}$.

It so happens that you have the Jacobian for the NSW network in CSV format. Find and rank the five best buses to add generators to, based on the magnitude of the $\frac{\delta Q}{\delta V}$ column. Keep in mind that lower values indicate a better location, so the best possible location would approach zero.

You will be given the following files:

- `diesel_locations.py`

You will edit this file so that it reads the provided Jacobian CSV file and then prints the best 5 bus locations in order.

- `jacobian_nsw.csv`

A kind researcher has given you a representation of the Jacobian matrix in CSV format to use in your study. The columns are as follows:

- *Bus Name* - a human readable bus location name.
- *Bus Number* - a bus number that corresponds to your `nsw.raw` base case.
- *dQ/dV* - the voltage sensitivity of this bus to changes in reactive power.

- `test_diesel_locations.py`

The unit test case for this example. There is a separate test case in this file for each task. You can run the test case like any other Python file:

```
C:\Windows\py.exe -3.7-32 tests\test_diesel_locations.py
```

9.5.2 Task 1

Edit the `find_best_locations` function so that it opens the given `jacobian_csv_filename` and creates a `csv.reader` object. Print the first five rows (do not print the header) of the CSV to the screen.

9.5.3 Task 2

Now extend the `find_best_locations` function to rank all of the locations by lowest dQ/dV to highest. Return this sorted list do not print it to the screen.

```
1 >>> # example output with invented buses - yours will differ to this.
2 >>> find_best_locations('jacobian_nsw.csv')
3 [('BAYWSATER 330KV', 22000, 0.03),
4  ('SYDWEST 500KV', 21000, 0.05),
5  ...
6  ('WOLLI CK 132KV', 25020, 1.85)]
```

9.5.4 Task 3

An extension question. Instead of returning all of the locations, return only the first five locations. Only return the bus numbers. Do not return the name and sensitivity columns.

```
1 >>> # example output with invented buses - yours will differ to this.
2 >>> find_best_locations('jacobian_nsw.csv')
3 [22000, 21000, 23340, 201001, 20200]
```

9.6 Sources

- [1] Python Foundation. Python Standard Library Documentation, Section 10.7 - glob, 2011. <http://docs.python.org/3.3/library/glob.html>.
- [2] Python Foundation. Python Standard Library Documentation, Section 13.1 - csv, 2011. <http://docs.python.org/3.3/library/csv.html>.
- [3] Python Foundation. Python Standard Library Documentation, Section 8.1 - date-time, 2011. <http://docs.python.org/3.3/library/datetime.html>.

Chapter 10

PSS[®]E and Exception Handling

You Will Learn

- Debugging difficult code; and
- Exception handling for the PSS[®]E API.

10.1 The Python Debugger — `pdb`

The `pdb` module allows you to halt your program at any point, and step through it one line at a time, inspect and even change variables.

`pdb` exists as its own module. To use it you must import it and then call one of its functions. The following example file uses the `pdb` function `set_trace()` about halfway down. `set_trace()` halts the program and allows us to step through the code one line at a time.

```
1 # usingpdb.py - a simple example with pdb.
2 buses = {20110: "swingbus", 20010: "genbus"}
3
4 import pdb
5 pdb.set_trace()
6
7 for bus in buses:
8     print(bus)
```

Here is an example of how the program output would look if `pdb` wasn't used:

```
1 20110
2 20010
```

Though, now that we have inserted `pdb`, we get:

```
1 -> for bus in buses:
2 (pdb)
```

and the program stops, with the cursor at the `(pdb)` prompt. From here we can check the contents of `buses`:

```
1 (pdb) buses
2 {20110: "swingbus", 20010: "genbus"}
```

We can try to check the contents of the variable `bus`:

```
1 (pdb) bus
2 *** NameError: name "bus" is not defined
```

The `NameError` arises because we haven't yet begun the `for` loop, so `bus` does not exist yet. Let's step to the next line of code. We use the `n` command to step to the next line of code.

```
1 (pdb) n
2 -> print(bus)
3 (pdb)
```

Now we have executed the first step of the `for` loop and are just about to print the contents of `bus`.

`pdb` has a whole range of commands that you can use:

n - to step over a function;

s - step into a function;

r - continue until the end of this function;

l - list the source code before and after the current line;

c - continue program execution until next break-point; and

q - quit immediately.

You can find more information about commands at the Python library reference <http://docs.python.org/library/pdb.html>.

10.2 PSS[®]E Exception Handling

All of the functions in the PSS[®]E API return an error code to indicate whether or not a function completed successfully. Returning error codes was the only method for checking if a function was successful in the Fortran and IPLAN languages on which PSS[®]E is based.

We learnt in Chapter 7 that the Python language was designed using another popular method for error checking called exception handling. The PSS[®]E API authors have included both the return code and exception handling methods for error handling, though the exception handling method is turned off by default.

It is a good idea to turn on the exception handling for the PSS[®]E API regardless of whether you use it in conjunction with the error handling capabilities Python provides. If an error is encountered in a PSS[®]E API call, it will act like any other exception in Python: if it is not handled, it will stop execution and inform you with some information (limited in the case of the PSS[®]E API) about the error. The key here is that it will inform you of the error immediately, not the next time you try to access the data that the function should have returned. This makes debugging code much easier.

To turn on exception handling, near the top of your files use the following code:

```
1 psspy.setThrowPsseExceptions(True)
```

Now running a function that has an error will raise an exception:

```
1 >>> ierr = psspy.cong()
2 >>> ierr = psspy.fns1()
3 Traceback (most recent call last):
4   File "converted_loads.py", line 17, in <module>
5   File "converted_loads.py", line 13, in main
6   File ".\psspy.py", line 5378, in fns1
7 psspy.Fns1Error: fns1 Error: ierr=2
```

Here we can see that in the file `converted_loads.py` on line 13, calling `psspy.fns1()` raised a `psspy.Fns1Error` with error code 2 (`ierr=2`). Looking up error code 2 in the PSS[®]E API documentation says that Generators are converted.

Alternately if you choose to check the return code, you must explicitly check all return codes for correctness:

```
1 >>> ierr = psspy.cong()
2 >>> ierr = psspy.fns1()
3 >>> if ierr != 0:
4     # do something here because you have detected an error.
```

All of the API functions will return `ierr` regardless of the error handling method you choose, even if you turn on exception handling for PSS[®]E.

10.3 Example: Find the Bug in the Haystack

10.3.1 Introduction

You are sitting in your luxurious corner office watching a boat slowly make its way past. Outside it is sunny and you can imagine yourself lying on the deck of the boat. The tall sail mast casts a shadow over your beer, keeping it cool. Without a single knock, in bounds your co-worker. His cherubic face is beaming. He has been assigned to investigate the voltage collapse near Armidale and whether or not the presence of diesel generators would have prevented the problem.

You are handed a USB stick, it contains a script that reproduces the conditions of the voltage collapse and a routine to add the new diesel generators.

Unfortunately for you the script has a difficult to track down bug in it which causes all of the results to be incorrect. Track down and fix the bug so that the results match the expected output file.

You will be given the following files:

- `armidale_non_cred_contingency.py`

The file given to you by your co-worker. The program prints the voltage at the Armidale 330kV bus before and after the non-credible contingency.

- `library.py`

A collection of useful utilities used by the `armidale_non_cred_contingency.py` file. The bug you are searching for is not inside one of these functions.

- `nsw.sav`

This is a model of the five state network that you should be familiar with. We are using it to study the voltage collapse condition near Armidale.

- `test_armidale_non_cred_contingency.py`

The unit test case for this example. There is only a single test that has been written at an extremely high level. You can run the test case like any other Python file:

```
c:\Windows\py.exe -3.7-32 tests\test_armidale_non_cred_contingency.py
```

10.3.2 Task 1

Find and fix the error in the `armidale_non_cred_contingency.py` program so that the test case passes.

You may use any debugging techniques that you like, some popular methods are:

- using `print` statements to show the value of key variables at certain stages in the program;
- using the Python debugger (`pdb`) to step through your code line by line and interactively find the error; and
- using exception handling to raise any errors and find the line in the program causing the problem.

Chapter 11

Logging

You Will Learn

- Logging program output to a file; and
- A faster way to debug your programs

11.1 A Simple Logging Example¹

A very simple example is:

```
1 import logging
2 logging.warning('Watch out!') # will print a message to the console
3 logging.info('I told you so') # will not print anything
```

If you type these lines into a script and run it, you'll see:

```
1 WARNING:root:Watch out!
```

printed on the console. The `INFO` message doesn't appear because the default logging level of the `logging` module is `WARNING`. The logging level sets the threshold for the severity of messages to be logged. Any less severe than the level set, are ignored. This feature will be discussed in greater detail in the next section. The printed message includes the indication of the level and the description of the event provided in the logging call, i.e. 'Watch out!'. Don't worry about the 'root' part for now: it will be explained later. The actual output can be formatted quite flexibly if you need that; formatting options will also be explained later.

11.1.1 Logging to a File¹

A very common situation is that of recording logging events in a file, so let's look at that next:

```
1 import logging
2 logging.basicConfig(filename='example.log',level=logging.DEBUG)
3 logging.debug('This message should go to the log file')
4 logging.info('So should this')
5 logging.warning('And this, too')
```

And now if we open the file and look at what we have, we should find the log messages:

```
1 DEBUG:root:This message should go to the log file
2 INFO:root:So should this
3 WARNING:root:And this, too
```

This example also shows how you can set the logging level which acts as the threshold for tracking. In this case, because we set the threshold to `DEBUG`, all of the messages were printed.

If you run the above script several times, the messages from successive runs are appended to the file `example.log`. If you want each run to start afresh, not remembering the messages from earlier runs, you can specify the *filemode* keyword argument, by changing the call in the above example to:

```
1 logging.basicConfig(filename='example.log', filemode='w', level=logging.DEBUG)
```

The output will be the same as before, but the log file is no longer appended to, so the messages from earlier runs are lost.

You can also specify a format that automatically writes the log time and other information.

```
1 FORMAT = '%(asctime)s %(message)s'
2 logging.basicConfig(format=FORMAT)
3 logging.warning('Protocol problem: connection reset')
```

would print something like

```
1 2006-02-08 22:20:02,165 Protocol problem: connection reset
```

11.2 Example

You are concerned that the Jacobian CSV file you were given yesterday didn't have the right number of buses in it. So you wrote a script that counts the number of buses in that file. Your manager thinks this is a great idea, and wants you to set up a routine log that reports on the number buses in that CSV file - in case it ever changes.

The log should have:

- time of check; and
- number of buses in the CSV.

Example of the log:

```
1 [INFO]      - 1998-01-01 00:00:00,000 - 10
2 [INFO]      - 1998-01-01 00:10:00,000 - 10
3 [WARNING]   - 1998-01-01 00:20:00,000 - File jacobian-nsw.csv is missing
```

Notice that it also reports any errors in the program, because your software is now considered mission critical.

Change your program so that instead of printing to the screen, the program output is logged to a file. Use Python's `logging` module for this task.

You will be given the following files:

- `csv_monitor.py`

You will modify this file in order to complete this example.

- `test_csv_monitor.py`

The unit test case for this example. There is a separate test case in this file for each task. You can run the test case like any other Python file:

```
C:\Windows\py.exe -3.7-32 tests\test_csv_monitor.py
```

11.2.1 Task 1

Use the `basicConfig()` function from the `logging` module to configure logging to a file called "`jacobian_monitor.log`", set the logging level to `INFO`, and the logging format to:

```
1 "[% (levelname)-10s] - %(asctime)-20s - %(msg)s"
```

Now when you create log messages, the output will look like:

```
1 [INFO]    ] - 1998-01-01 00:00:00,000 - 0
```

11.2.2 Task 2

Use the `logging` module to log the number of buses in the `jacobian_nsw.csv` file using the `INFO` level message. Issue a `WARNING` level message if the CSV file is missing:

```
1 [WARNING ] - 1998-01-01 00:00:00,00 - No buses in jacobian_nsw.csv.
```

or there are zero buses in the CSV file,

```
1 [WARNING] - 1998-01-01 00:20:00,000 - File jacobian_nsw.csv is missing
```

The CSV filename name in the warning message should change based on the CSV filename in the `settings` module. So if the `settings` module `CSV_FILE` variable changed to `alternate.csv` then the `WARNING` message should read:

```
1 [WARNING ] - 1998-01-01 00:00:00,00 - No buses in alternate.csv.
```

11.3 Sources

- [1] Vinay Sajip. Python logging Howto, 2011. <http://docs.python.org/dev/howto/logging.html>.

Chapter 12

Introducing the GUI

You Will Learn

To make pop up boxes to ask the user for:

- a filename;
- a save file location;
- text input; and
- the answer to a yes or no question.

12.1 Ask the User for Input

Every Python installation ships with a built in Graphical User Interface (GUI) toolkit called `tkinter`. With `tkinter` you can do simple things; like pop up a dialog box that asks for a filename. Or you can do more complex things; like creating an entire mp3 player application.

In the following sections we'll give you some simple examples you can use to make your program more user friendly. We'll then conclude with building our own pop up box from scratch.

12.1.1 Ask for Filename

Tkinter ships with a library of pre-built pop up boxes called `filedialog`. The functions in this library are all related to asking for filenames and directories.

In this example we create an *Open* file window, as shown in Figure 12.1 on the next page. You might use this to ask for which saved project file to use.

Function Signature

```
filedialog.askopenfilename(**options)
```

A list of the available *options* keyword arguments:

- *filetypes* sets the types of files that may be selected from. For example: Use this to restrict the user to `.xls` files only.

```
filetypes=[('Excel 95', '*.xls'), ('Excel 2010', '*.xlsx')]
```

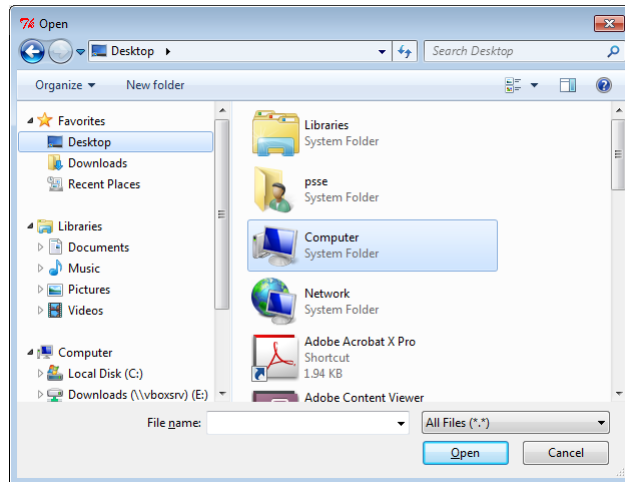


Figure 12.1: File open select box created by `filedialog.askopenfilename()`

- *initialdir* sets the starting directory that is shown when the dialog pops up. Otherwise the current working directory is shown.
`initialdir=r"c:\saved files"`
- *title* sets the window title, otherwise the default: *Open* is used.
`title="Please select a Project File"`

Example Usage

```
1 from tkinter import filedialog
2 filename = filedialog.askopenfilename()
```

12.1.2 Ask for Save File Location

`filedialog.asksaveasfilename()` is similar to the `filedialog.askopenfilename()` function but with one crucial difference: you may enter a new filename to save to. An example of this dialog box is shown in Figure 12.2 on the facing page. Contrast this with `filedialog.askopenfilename()` where you must select an already existing file.

Function Signature

The `filedialog.asksaveasfilename()` function has the same function options as the `filedialog.askopenfilename()` function.

Example Usage

```
1 from tkinter import filedialog
2 filename = filedialog.asksaveasfilename()
```

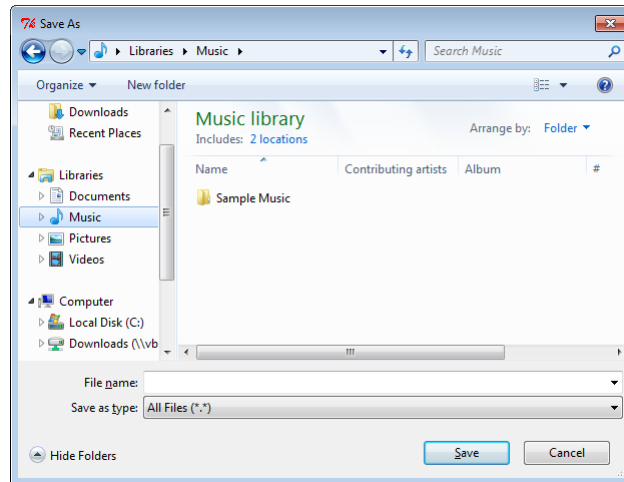


Figure 12.2: Save as dialog box created by `filedialog.asksaveasfilename()`

12.1.3 Ask for Directory

What if you want to know where a folder containing files is but not any particular filename? Use `filedialog.askdirectory()`, as shown in Figure 12.3 on the next page.

You might use this to ask where to keep log files.

Function Signature

```
filedialog.askdirectory([**options])
```

A list of the available *options* keyword arguments:

- *initialdir* sets the starting directory that is shown when the dialog pops up. Otherwise the current working directory is shown.
`initialdir=r"c:\saved files"`
- *title* sets the window title, otherwise the default: *Open* is used.
`title="Please select a Project File"`
- *mustexist* sets whether the user can create a new directory. Otherwise the directory must exist. The default value is `False`.
`mustexist=True`

Example Usage

```
1 from tkinter import filedialog
2 filename = filedialog.askdirectory()
```

12.1.4 Ask a Question

Tkinter comes with another library `messagebox` for asking yes or no type questions, as shown in Figure 12.4 on page 145. Example uses are “Save before quitting?” and “Are

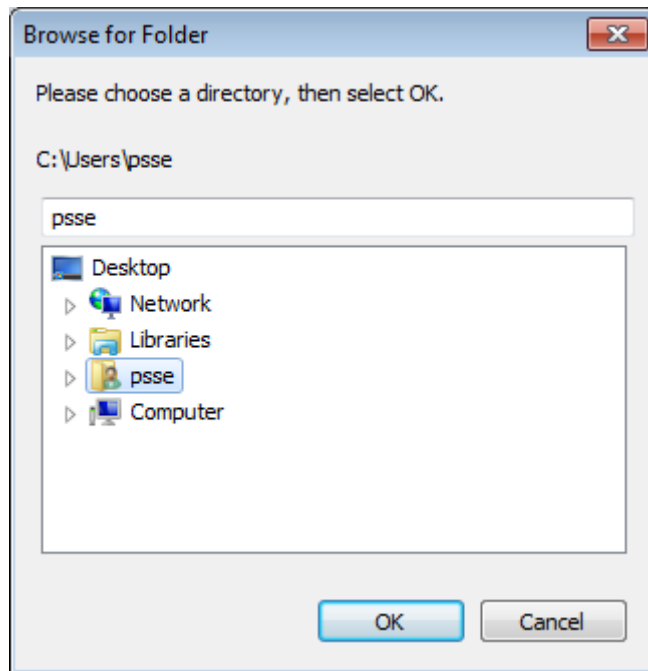


Figure 12.3: Select directory dialog box created by `filedialog.askdirectory()`

you sure you want to delete the contents of the c drive?".

Function Signature

`messagebox.askyesno(title, message)`

- *title* sets the window title.
- *message* sets the message shown above the “Yes” and “No” buttons.

Example Usage

```
1 >>> from tkinter import messagebox
2 >>> response = messagebox.askyesno(
3 ...     title="Before you quit",
4 ...     message="Would you like to save your work?")
5 >>> print(response)
6 False
```

12.1.5 Ask About Anything

Sometimes you want more information than just a filename, directory or a simple yes or no answer. You might want to know the terminal name to search for, the user’s email address or their preference from a list of choices.

This is where the ready made pop up boxes in `filedialog` and `messagebox` cannot help you. You will need to make your own custom application to ask for the information.

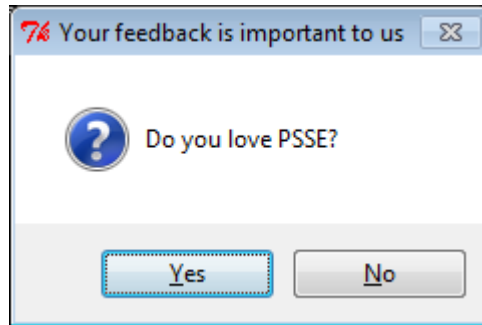


Figure 12.4: A message box posing a question with an obvious answer. Created by `messagebox.askyesno()`

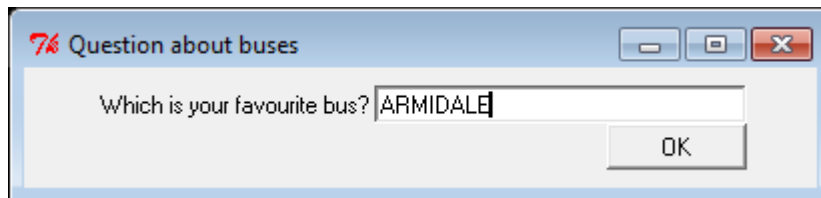


Figure 12.5: Custom dialog box created by the example code.

We'll show you how to make an application to ask for a new saved case title. But you can use it to ask for any text input. The dialog box created by this example is shown in Figure 12.5.

```

1 from tkinter import Tk, StringVar, Entry, Button, Label, W, E
2
3 def askquestion(title, message, default=None):
4     root = Tk()
5     root.title(title)
6
7     # calculate the approximate width to set the screen to.
8     # based on the number of characters in the title or message.
9     width = max(len(message), len(title))
10    text_width = width * 7 + 200
11
12    root.geometry(str(text_width) + "x60")
13    root.grid()
14
15    label = Label(root, text=message)
16    label.grid(sticky=W, row=0, column=1)
17
18    value = StringVar()
19    entry = Entry(root, width=30, textvariable=value)
20    entry.grid(row=0, column=2)
21
22    result = []

```

```
23 def get_value():
24     result.append(value.get())
25     root.destroy()
26
27 btn = Button(root, text="OK", command=get_value, width=10)
28 btn.grid(sticky=E, row=1, column=2)
29
30 root.mainloop()
31
32 try:
33     return result.pop()
34 except IndexError:
35     return default
```

Before we begin understanding how this was done, here is a quick example of how you would use the `askquestion()` function.

```
1 >>> askquestion("Question about buses", "Which is your favourite bus?")
2 'BAYS_11kV_G3'
3 >>> askquestion("Question about buses", "What do you like about it?")
4 'Perfect combination of strength and magic'
```

We have imported from `tkinter`. The `tkinter` module is where all the objects you need to build your own application are. Lets describe each of the objects we imported:

Tk is the core application class. This creates an empty window complete with title and exit box in the corner.

StringVar is a variable object that can let you know when its value has changed. The **Entry** object uses the **StringVar** to store the user input.

Entry is the text entry field that you can see in Figure 12.5. This is what your user will actually type into.

Button is a blank button. We'll write our own text onto it "OK" and specify which function to call when it is clicked (`get_value()`).

Label some static text we place next to the **Entry** field.

W and **E** are variables that mean West and East. We use **E** to set the button to stick to the east (right hand) side of the window.

Tkinter applications consist of a root application window that is populated by various **Labels**, **Buttons** and **Entry** objects. We create a base grid layout with columns and rows on the root window. Then into each cell of the grid we put individual **Label** and **Button** objects.

Looking at Figure 12.5 on the preceding page, can you see that the root application window is divided into two rows and two columns? The first column in row one contains the **Label** "Which is your favourite bus?". The second column in row one is the **Entry** field. Finally we have put the "OK" **Button** into column two of the second row (and aligned it to the east).

```
1 root = Tk()
2 root.title(title)
```

This creates the root application window. It is blank at this point, except for the window title which we set.

```
1 # calculate the approximate width to set the screen to.
2 # based on the number of characters in the title or message.
3 width = max(len(message), len(title))
4 text_width = width * 7 + 200
5
6 root.geometry(str(text_width) + "x60")
7 root.grid()
```

Now we have (crudely) estimated the width of the window, then set its width and height using the `root.geometry` function. Finally we set the application window to have a grid layout.

```
1 label = Label(root, text=message)
2 label.grid(sticky=W, row=0, column=1)
3
4 value = StringVar()
5 entry = Entry(root, width=30, textvariable=value)
6 entry.grid(row=0, column=2)
```

Creating a `Label` and inserting it into the first row and first column. The `StringVar` is given to the `Entry` so it can store the user input.

```
1 result = []
2 def get_value():
3     result.append(value.get())
4     root.destroy()
5
6 btn = Button(root, text="OK", command=get_value, width=10)
7 btn.grid(sticky=E, row=1, column=2)
```

The "OK" Button will be put into the second row and second column. When clicked the Button will call the function `get_value()`. `get_value()` stores whatever the user typed into the Entry into the list `result`. It then destroys the root application. Destroy just means exiting the `root.mainloop()` function and closing the window.

```
1 root.mainloop()
2
3 try:
4     return result.pop()
5 except IndexError:
6     return default
```

Once `root.mainloop()` is called the root application window is made visible. The program will wait inside `root.mainloop()` until:

- the user either clicks the "OK" button (because `get_value()` calls `root.destroy()`);
- the user clicks the red exit button in the top right.

Finally, we try to return the contents of the `result` list, or if it is empty we return the given `default` value given.

You can learn more about Tkinter and its wide array of options for creating your own GUIs at <http://effbot.org/tkinterbook/>. The link says that it was last updated in 2005, it is written in Python2 rather than Python3, but the content is still very good.

The main difference between Python2 and Python3 was that `tkinter` had a capital 'T' `Tkinter` in Python2. Also both `filedialog` and `messagebox` were called `tkFileDialog` and `tkMessageBox` in Python2.

12.2 Example

12.2.1 Introduction

You have written a QV curve application. A beautiful program that can chart a QV curve for any bus on any network. Not only that, your program will also show the QV curves before and then after adding DieselCo DGFIFTY generators.

The code is a piece of art, suitable for hanging in the Louvre, Paris or any other public museum. But when you show your co-workers they all complain. “How do I use it?” they ask, “Its too hard to pick which base case file and bus number”. You try to reply, “Its as easy as changing these variables here, here and in he..” but they cut you off. None of them want to open your Python files and change variables to run your program. You need a way to make it “user friendly” maybe even “engineer friendly” which is altogether less arduous than “Grandma and Grandpa friendly”.

You’ll write some simple graphical user interfaces for the QV curve program in this example.

You will be given the following files:

- `nsw.raw`

This is a model of the NSW network that you should be familiar with. We will be using this network throughout our examples as we continue to study the voltage collapse incident.

- `qv_curve_comparison.py`

This is the QV curve application that you have written. The easy parts are already done; it successfully runs a QV analysis for a bus for a particular project file before and after adding the DieselCo DGFIFTY generators. You will be doing the difficult part: writing the GUI.

You will create this file

The program output is a nice PNG format image of the before and after QV curve. These image files can be put into your next Microsoft Word report or even shared with mates on Facebook, LinkedIn and Lotus Notes.

12.2.2 Task 1

The output PNG image filename is hardcoded in `qv_curve_comparison.py`. Edit the script so that the user can select the filename to save to via a pop-up tkinter file save as box.

12.2.3 Task 2

This is an extension question. The project name is hardcoded in the `qv_curve_comparison.py` Python script. Write your own custom Tkinter application that can ask the user for the project name to run a study on. You must check to see if the project name the user gives you is:

- Not blank

- A valid project name (it actually loads).

before running the study.

Chapter 13

Send Your Data To Excel

You Will Learn

- To create new Excel spreadsheets from Python;
- To send columns and rows of data to Excel; and
- Formatting for your Excel files so they look beautiful.

Excel Modules in the PSS®E API

Two modules that interact with Microsoft Excel are provided by the PSS®E Python API: `pssexcel` and `excelpy`. These modules are covered in the PSS®E Python API docs in Chapter 9.2 and Chapter 9.3 respectively. This chapter has a bias towards the general purpose module, `excelpy`, which allows Python to interact with Excel.

A brief section on `pssexcel`, which is useful for populating spreadsheets with the results from ACCC, PV and QV analyses as well as data which is useful for creating IECS fault data input files, is included at the end of this chapter.

13.1 Introducing `excelpy`

With `excelpy` you can completely control Excel from your Python scripts. Though don't get caught up trying to create Excel charts with it, because you can't.

In the following sections we will explore some examples that use `excelpy` module to control Microsoft Excel. Due to the size of the `excelpy` API, it is not feasible to cover all of the functionality provided by this function. We have made an effort to cover the fundamental operations that a user would want from this module.

For further information about this module, refer to the Chapter 9.3 of the PSS®E API documentation. Interestingly, this API does not cover reading from Excel files. If you desire more information about `excelpy` than the documentation provides, consider using the inbuilt documentation from the Python interpreter, as shown in the following example.

```
1 >>> # Make sure the PSSPY37 directory is on your path.  
2 >>> import excelpy
```

```
3 >>> help(excelpy)
4 Help on module excelpy:
5
6 NAME
7     excelpy
8
9 FILE
10    c:\program files (x86)\pti\psse34\psspy37\excelpy.pyc
11
12 DESCRIPTION
13    Excel Interface Python Functions
14
15    ... A complete listing, with explanations, of all
16        the functions provided by excelpy...
```

13.1.1 Create a New Workbook or Worksheet

Open Existing Workbook

The `excelpy.workbook()` function creates and returns a link to an existing or new Excel Workbook.

Function Signature

```
excelpy.workbook([xlsfile=''], [sheet=''], [overwritesheet=False], [mode='w'])
```

Function Arguments:

- *xlsfile* filename of the Excel file to open. Leave this blank to create a new Workbook.
- *sheet* Create this worksheet if in *write* mode or open this worksheet if in *read* mode.
- *overwritesheet* only used in write mode. If `True`, delete then create the named *sheet*. And if `False`, first backup any existing sheet named *sheet* before creating a new empty one. Default `False`.
- *mode* 'w' - create a new sheet with name *sheet*, 'r' - use existing sheet with name *sheet*. Default 'w'.
- This brief list of parameters covers the most used options. Refer to the PSS®E API for a full list of parameters for this function.

```
1 import excelpy
2
3 PAYROLL_ADVICE = r'c:\company finances\employee_salaries.xls'
4 SALARIES_SHEETNAME = "salaries_2010"
5
6 # create new sheet and delete the existing sheet (if it exists)
7 xlsfile = excelpy.workbook(
8     PAYROLL_ADVICE,
9     sheet=SALARIES_SHEETNAME,
10    overwritesheet=True)
11
12 # create new sheet and rename the existing sheet (if it exists)
```

```
13 xlsfile = excelpy.workbook(  
14     PAYROLL_ADVICE,  
15     sheet=SALARIES_SHEETNAME,  
16     overwritesheet=False)  
17  
18 # just open an existing sheet, dont delete or rename it.  
19 xlsfile = excelpy.workbook(  
20     PAYROLL_ADVICE,  
21     sheet=SALARIES_SHEETNAME,  
22     mode='r')
```

A New Empty Excel File

If the *xlsfile* isn't given then a new empty Excel file will be created.

```
1 import excelpy  
2  
3 # a brand new file, not saved anywhere yet.  
4 xlsfile = excelpy.workbook()
```

13.1.2 Writing Data to Excel

Write to the Cell A1

```
set_cell(address, value[, sheet=''])
```

Function Options

- *address* set this cell equal to *value*. Can be a tuple (row,column), a string 'a1', or a combination (1, 'a').
- *value* set the cell equal to this. The value can be a string, float, boolean, None or integer.
- *sheet* write to the cell in the named *sheet*. By default write to the currently open sheet.
- This list of parameters covers the most used options. Refer to the PSS®E API for a full list of parameters for this function (mostly formatting options).

```
1 xlsfile = excelpy.workbook()  
2 xlsfile.set_cell(address='a1', value="Hello World!")
```

Writing A Formula

```
1 xlsfile = excelpy.workbook()  
2  
3 xlsfile.set_cell(address='a1', value="=TODAY()")
```

Writing a Whole Range at Once

```
set_range(topRow, leftCol, data [, sheet=''])
```

Function arguments:

- *topRow* Set data in Excel starting from this row. Must be a row number.
- *leftCol* Set data in Excel starting from this column. Must be an integer column number or a string. E.g. 'a' or 1
- *data* A list of data rows. Will be set in Excel with the top corner specified by *topRow* and *topCol*
- *sheet* write to the range in the named *sheet*. By default write to the currently open sheet.
- This list of parameters covers the most used options. Refer to the PSS®E API for a full list of parameters for this function (mostly formatting options).

```
1 xlsfile = excelpy.workbook()
2 rows = [['Price', 'Demand', 'SettlementDate'],
3         [5.6      , 10202    , '2010-05-03 00:00:00']]
4 xlsfile.set_range(
5     topRow=1,
6     leftCol=1,
7     data=rows)
```

13.1.3 Reading Data From Excel

The PSS®E API documentation does not include information about the functions used for reading data from an Excel spreadsheet. Refer to the example at the start of the chapter about accessing the complete module documentation from an interactive Python session.

As with writing data to a spreadsheet, there are two functions for reading data from a spreadsheet: `get_cell()` and `get_range()`.

Get value from one cell.

```
get_cell(address, sheet=None)
```

Function Options

- *address* read the cell at this *address*. Can be a tuple (row,column), a string 'a1', or a combination (1, 'a').
- *sheet* the sheet to be read from.

Get data from cells in a range.

```
get_range(address[, transpose=False][, sheet=None])
```

Function Options

- *address* - range tuple specified as (topRow, leftCol, bottomRow, rightCol)

- *transpose* - Logical

Excel methods read data row by row and return a Python list of lists.

`transpose = False`

Each returned list represents data in a spreadsheet row.

`transpose = True`

Each returned list represents data in a spreadsheet column.

```
1 xlsfile = excelpy.workbook()
2
3 # read from a1 (1, 1) to j10 (10, 10)
4 rows = xlsfile.get_range(address=(1, 1, 10, 10))
```

13.1.4 Formatting Your Excel Files

There are many formatting functions which can be applied to single cells or a range of cells. Borders, font, text alignment and a few other attributes can be formatted using the formatting functions provided by `pssexcel`. All of the formatting functions operate in the same way: they take an address argument, which may be a single cell or a range, and formatting options particular to the type of formatting being conducted and apply the formatting to the cells.

For more information about formatting Excel spreadsheets, refer to the PSS[®]E API documentation.

The following example will show how to change the font for a range of cells.

```
1 xlsfile = excelpy.workbook()
2
3 rows = [['Price', 'Demand', 'SettlementDate'],
4         [5.6, 10202, '2010-05-03 00:00:00']]
5 xlsfile.set_range(
6     topRow=1,
7     leftCol=1,
8     data=rows)
9
10 # set the header row (1,1) to (1,3) bold.
11 xlsfile.font(address=(1,1,1,3), fontStyle='bold')
12
13 # set the data row (2,1) to (2,3) italic.
14 xlsfile.font(address=(2,1,2,3), fontStyle='italic')
15
16 # change the heading font colour to red.
17 xlsfile.font_color(address=(1,1,1,3), color='red')
```

13.2 Introducing pssexcel

The more specialized module, `pssexcel`, is used to write the results of a small set of operations to a spreadsheet. In particular, this module can write various user-selectable aspects of the results of an ACCC, PV or QV analysis to a spreadsheet.

This module can also be used to aid in the construction of an IEC input data file from a PSS[®]E save case by exporting the data required to generate the IEC input file into a spreadsheet. While the spreadsheet is not the IEC input file, the documentation suggests that a fault data input file can be generated by manipulating the data in the spreadsheet to obtain the required scenario. After the scenario has been generated in the Excel file, the fault input file can be generated by exporting the data from Excel into a CSV file with an extension `.iec` for use as the input to the IECS API.

The discussion of this module will conclude with an example of using this module to create an Excel file with the results of an ACCC analysis. The `pssexcel.accc()` function has many more options than what is shown below. For a complete listing, check the documentation under Chapter 9.2 of the PSS[®]E API reference.

This example assumes that an ACCC analysis output file, `my_accfile.acc`, exists.

```
1 import pssexcel
2
3 # 's' or 'summary'      ACCC Analysis Summary
4 # 'e' or 'events'      Contingency Events Description
5 # 'b' or 'branch'      Monitored Branch Flow (MVA)
6 # 'i' or 'interface'   Monitored Interface Flow (MW)
7 # 'v' or 'voltage'     Monitored Bus Voltage
8 # 'l' or 'load'        Loads Shed (MW)
9 # 'g' or 'generator'   Generator Dispatch (MW)
10 # 'p' or 'phase shifter' Phase Shifter Angle
11 options = ['s','e','b']
12
13 pssexcel.accc(accfile='my_accfile.acc', string=options, xlsfile='my_accc.xls')
```

13.3 Example

13.3.1 Introduction

Having shown the difference before and after adding the DGFIFTY generators, you are something of a local hero at work. Your direct Manager and Senior Manager are so pleased with your findings that they have asked that you put together a spreadsheet for the company board.

The board want to see each generator location name and their size. The board do not care for bus numbers, but you can assume that they are capable of reading the bus names from your saved case.

You will be given the following files:

- `nsw.sav`

This is a model of the NSW network that you should be familiar with. We will be using this network throughout our examples as we continue to study the voltage collapse incident.

- `excel_board_report.py`

This is the program you need to edit and will create the board report.

You will create this file

You will write an Excel file with three columns:

- *Location Name* this is the bus name
- *Generator Size* this is the machine base of the generator
- *Cost* the generator cost in millions of dollars. Assume a cost of one million dollars per megawatt of installed capacity.

13.3.2 Task 1

The `get_gens_from` function now reads a list of diesel generators from a saved case. The format of the returned list is:

```
[('ARM 132KV', 5.3), ...]
```

where each pair consists of the bus name and machine base in MW.

Extend the `board_report` function so that it writes the three columns *Location Name*, *Generator Size* and *Cost* to the columns *A*, *B* and *C* in an Excel workbook.

The *Cost* value isn't returned by the `get_gens_from` function. You will have to create it yourself using the formula given in the section above.

13.3.3 Task 2

Extend the `board_report` function so that it bolds the heading row.

13.3.4 Task 3

This is an extension questions. Make the cost per megawatt column a formula in Excel. The formula for the *Cost* column *C2* should be "`=B2 * D1`". Changing cell D1 to two million should then double your installed cost.

Chapter 14

A QV Curve Generator

You Will Learn

- Multiple axes and sub plots on `matplotlib` charts;
- To use the `qv_engine` API to run a QV analysis with multiple contingencies; and
- The `pssarrays` library to read QV analysis result files.

14.1 Adding Multiple Axes and Subplots to `matplotlib` Charts¹

MATLAB[®], and `pyplot`, have the concept of the current figure and the current axes. All plotting commands apply to the current axes. The function `gca()` returns the current axes (a `matplotlib.axes.Axes` instance), and `gcf()` returns the current figure (`matplotlib.figure.Figure` instance). Normally, you don't have to worry about this, because it is all taken care of behind the scenes. Below is a script to create two subplots.

```
1 import numpy as np
2 import matplotlib.pyplot as plt
3
4 def f(t):
5     return np.exp(-t) * np.cos(2*np.pi*t)
6
7 t1 = np.arange(0.0, 5.0, 0.1)
8 t2 = np.arange(0.0, 5.0, 0.02)
9
10 plt.figure(1)
11 plt.subplot(211)
12 plt.plot(t1, f(t1), 'bo', t2, f(t2), 'k')
13
14 plt.subplot(212)
15 plt.plot(t2, np.cos(2*np.pi*t2), 'r--')
```

The output of this command is shown in Figure 14.1 on the next page.

The `figure()` command here is optional because `figure(1)` will be created by default, just as a `subplot(111)` will be created by default if you don't manually specify

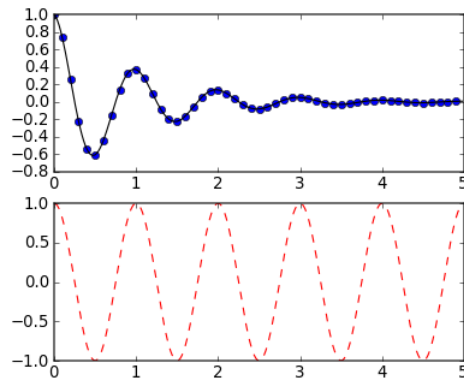


Figure 14.1: `subplot()` allows more than one plot to be included in a single figure.

an axes. The `subplot()` command specifies `numrows`, `numcols`, `fignum` where `fignum` ranges from 1 to `numrows` $\times$ `numcols`. The commas in the subplot command are optional if `numrows` $\times$ `numcols` < 10. So `subplot(211)` is identical to `subplot(2,1,1)`. You can create an arbitrary number of subplots and axes. If you want to place an axes manually, i.e. not on a rectangular grid, use the `axes()` command, which allows you to specify the location as `axes([left, bottom, width, height])` where all values are in fractional (0 to 1) coordinates.

You can create multiple figures by using multiple `figure()` calls with an increasing figure number. Of course, each figure can contain as many axes and subplots as your heart desires:

```
1 import matplotlib.pyplot as plt
2 plt.figure(1)           # the first figure
3 plt.subplot(211)        # the first subplot in the first figure
4 plt.plot([1,2,3])
5 plt.subplot(212)        # the second subplot in the first figure
6 plt.plot([4,5,6])
7
8
9 plt.figure(2)           # a second figure
10 plt.plot([4,5,6])       # creates a subplot(111) by default
11
12 plt.figure(1)           # figure 1 current; subplot(212) still current
13 plt.subplot(211)        # make subplot(211) in figure1 current
14 plt.title('Easy as 1,2,3') # subplot 211 title
```

You can clear the current figure with `clf()` and the current axes with `cla()`.

If you are making a long sequence of figures, you need to be aware of one more thing: the memory required for a figure is not completely released until the figure is explicitly closed with `close()`. Deleting all references to the figure, and using the window manager to kill the window in which the figure appears on the screen, is not enough, because `pyplot` maintains internal references until `close()` is called.

14.2 About the `qv_engine` API

Preparing for a QV analysis can be difficult. The process requires that you use the `dfax()` command to create a Distribution Factor Data File (dfx) file which you pass to the `qv_engine` that in turn writes its own output file.

Use the `psspy.dfax()` function to create the Distribution Factor Data File. You will require a subsystem data file, a monitored element file and a contingency description data file. The file formats for these files are complex, we refer you to the PSS®E Program Operation Manual for detailed information on each.

14.2.1 Subsystem Description Data File

Subsystems of the working case are listed in this file. Subsystems created using this file can be referred to in the Monitored Element Data file and Contingency Description Data file. Information about the file format is in Section 7.1.2 of the PSS®E Program Operation Manual.

14.2.2 Monitored Element Data File

Network elements that will be monitored for problems. Information about the file format is in Section 7.1.3 of the PSS®E Program Operation Manual

14.2.3 Contingency Description Data File

Contingencies (outages of elements) that will be tested as part of the QV analysis listed in this file. Information about the file format is in Section 7.1.4 of the PSS®E Program Operation Manual.

14.2.4 How to Setup `qv_engine()`

Create a dfx file using the `dfax()` command. *dfxfile* is the filename that PSS®E should write the dfx file to:

```
1 options = [1, 0] # calculate distribution factors, do not sort contents.
2 ierr = psspy.dfax(options, subfile, monfile, confile, dfxfile)
```

We will now run a QV analysis with the default solution settings for brevity. The solution will sweep the voltage range from 1.05p.u to 0.95p.u in 0.02p.u voltage step decrements.

```
1 settings = {
2     'values2': 1.05, # VHI
3     'values3': 0.95, # VLO
4     'values4': 0.02, # Delta V
5     'options10': studybus
6 }
7
8 ierr = psspy.qv_engine(dfxfile=dfxfile, accfile=accfile, **settings)
```

The results of the `qv_engine` command are now stored in the `accfile` output file. The next section will show you how to read the contents of an `accfile` to Python.

14.3 About the `pssarrays` Library

To read the output file created by `qv_engine()` you will need two functions: `qv_summary()` and `qv_solution()`

The `qv_summary()` file will list all the meta information about the QV analysis results:

- The string labels for contingencies run; and
- The number of monitored buses

The `qv_solution()` will give you the voltage and reactive power values that you need to plot on a chart, but only for a single contingency. You will need to first get a list of the contingency labels from the `qv_summary()` function and then request the voltage and reactive power values for charting.

So imagine the results of the last `qv_engine()` were written to the filename in variable `accfile`.

```
1 summary = pssarrays.qv_summary(accfile)
```

The list of contingencies we are after is in `summary.colabel`

```
1 contingency_names = summary.colabel
```

Now let's get the voltage and MVAR of the monitored plant for the first contingency:

```
1 rlst = pssarrays.qv_solution(accfile, contingency_names[0])
2 voltage = rlst.volts
3 mvar = rlst.mgenmvar
```

The `volts` object stores the monitored bus voltages in columns and the `mgenmvar` object stores the monitored generator bus MVAR contributions in columns. Both the `volts` and `mgenmvar` objects are differently sized arrays. We need to do some mildly complicated work to find out how to match the values in `mgenmvar` with their corresponding values in `volts`.

To get the voltage monitored bus labels:

```
1 volts_bus_labels = summary.mvbuslabel
```

To get the machine bus monitored bus labels:

```
1 mvar_bus_labels = summary.mgenbus
```

Bus labels are not the same thing as bus name or bus number. Instead of bus number or bus name, the `qv_summary()` function returns a combination of bus number, bus name and base kV voltage in a single string e.g:

```
1 " 20200 TAMW 132.0 "
```

We have a list of the voltage monitored bus labels and a 2D array of voltage measurements. In the next step we will pair each bus label with its recorded voltage measurements.

The voltage measurements are stored with one voltage step per row. So there will be 21 rows if the voltage measurements are between 1.1 and 0.9p.u at 0.01p.u per step. In this example we have only monitored 5 voltage buses, so each of the 21 rows will have 5 columns and so too the `volts_bus_labels` variable is 5 long.

First we need to transpose the 21 x 5 voltage measurement list to be 5 x 21:

```
1 transposed = zip(*voltage)
```

Now both `transposed` and `volts_bus_labels` have 5 elements and we can pair them together.

```
1 paired_labels_and_measurements = zip(volts_bus_labels, transposed)
```

The `paired_labels_and_measurements` is just a list of tuples. The first element is the bus label and the second is the 21 voltage measurements for that bus:

```
1 ((" 20200 TAMW 132.0 ", [1.1, 1.09, 1.08 ... ]),  
2 (" 20201 TAMW 132.0 ", [1.1, 1.092, 1.08 ...]))
```

As is, the paired labels isn't all that useful. But we can create a dictionary from it, and benefit from key lookups.

```
1 >>> voltage_measurements = dict(paired_labels_and_measurements)  
2 >>> voltage_measurements[" 20200 TAMW 132.0"]  
3 [1.1, 1.09, 1.08, ...]
```

If we do the same for the monitored generator buses, we can pair the voltage and reactive power measurements for all of the generator buses.

```
1 tamw_reactive = reactive_measurements[" 20200 TAMW 132.0"]  
2 tamw_voltage = voltage_measurements[" 20200 TAMW 132.0"]
```

Now it should be easy to create a chart with `matplotlib` with the reactive measurements on the Y axis and voltage measurements on the X axis.

```
1 plt.plot(tamw_voltage, tamw_reactive)  
2 plt.show()
```

14.4 Example: Voltage Stability and Dealing With a Difficult API

Before you submit all of your results to the company board, we'll revisit the QV curve example to write our own QV curve program (this time without the GUI).

Use the PSS®E `qv_engine` API to run a QV analysis for a number of monitored buses which will be given to you in a CSV file. The structure of the CSV file is quite simple, it contains a single column of bus numbers without a heading.

Then use the `pssarrays` library to read the `qv_engine` output file into Python. You will need to think of a way to transform the `pssarrays` output to a form that the `qv_curve()` function provided in your library can accept and plot.

A word of warning though, the QV analysis APIs are not for the faint of heart. This is an example of a difficult to use API, one where you must jump through many hoops to accomplish a task. Specifically you need to create many input files to run the `qv_engine` and then use multiple function calls to parse the resulting `qv_engine` output file. If you can't finish this example don't worry. It is such a poorly written API that even experienced Python programmers would have a hard time writing code with it. We hope that you learn from PSS®E's mistakes.

You will be given these files:

- `qv_curve.py`

Your code for this example should be written into this file. This program will produce a line chart showing a QV curve for each of the monitored buses and their contingencies.

- `library.py`

You should already have this file from previous examples. It contains a function `qv_chart` that can be used to create a line chart figure for the QV analysis results.

- `nsw.sav`

This is a model of the NSW network that you should be familiar with. We will be using this network throughout our examples as we continue to study the voltage collapse incident.

- `test_qv_curve.py`

The unit test case for this example. There is only a single test that has been written at an extremely high level. You can run the test case like any other Python file:

```
c:\Windows\py.exe -3.7-32 tests\test_qv_curve.py
```

The output on the command line will show where the tests are failing and give you some helpful debugging information.

You will create this file:

- The goal of this example is to save a PNG format image of a QV curve that you generate for the monitored buses and contingencies.

14.4.1 Task 1

We will extend the `qv_for_subsystem()` function to write the required con and mon files.

- Write the list of buses to a subfile using the `write_buses_in_subfile_format()`. This subfile is a temporary file, and doesn't need to exist beyond the end of the function. You should use the provided `make_temp_file()` function to create a temporary subfile.
- Write a monitored element file using the `write_monfile` function. Again the monitored element file is temporary, we don't need it after the function is complete so you should again use the `make_temp_file` function to create it.

Example `make_temp_file` usage:

```
1 >>> monfilename = make_temp_file('.mon')
2 >>> print monfilename
3 '89sgl12.mon'
4 >>> with open(monfilename, 'w') as opened:
5 ...     opened.write("just testing.")
6 ...
```

14.4.2 Task 2

We will now extend the `qv_for_subsystem()` function to use the PSS[®]E API.

Use the `psspy.dfax()` function to write the dfax file that is required to run the `qv_engine()`. Use the subfile and monfile you created earlier, and write to a new temporary dfax file. You should again use the `make_temp_file()` function to create a dfax filename, because we don't need it beyond the end of the `qv_for_subsystem()` function.

14.4.3 Task 3

We enter the final phase for editing the `qv_for_subsystem()` function.

Use the `psspy.qv_engine()` function to run the QV analysis and store the output accfile (which is the `qv_results_filename` argument).

Finally use the `try_delete_files()` function to remove the subfile monfile and dfax file which are temporary and no longer required.

With the conclusion of this task, the provided example framework will use your `qv_for_subsystem()` function to write the QV analysis result to a PNG chart in the same directory.

14.5 Sources

- [1] John Hunter, Darren Dale, and Michael Droettboom. Matplotlib User Guide - pyplot, 2011. http://matplotlib.sourceforge.net/users/pyplot_tutorial.html.

Chapter 15

Automating your Dynamic Studies

You Will Learn

- How to use the PSS®E Dynamic Study API; and
- To extract data from a PSS®E Dynamic Study.

15.1 About the Dynamics API

With the PSS®E Dynamics API you can run your dynamic studies automatically, while you have a coffee or spend more time studying the results.

Here are some dynamics studies that you could automate:

- Critical Clearing Time;
- Response to voltage disturbances; and
- Partial Load Rejection.

15.2 A Short Note About Preparing Dynamics Data

Preparing dynamics data for a study is involved and manual. You may find it difficult to automate, and we won't be covering preparation of dynamics data files in this course. PSS®E have a guide on manually preparing cases for dynamic study in Chapter 13 of the Program Application Manual II.

15.3 Loading a Dynamic Case and Snapshot

Given a saved case and a snapshot file that have been created earlier for dynamic studies. Here is how you would load them into PSS®E

```
1 SAVED_CASE = "savcnv.sav"  
2 SNAP_FILE = "savnw.snp"
```

```
3
4 psspy.case(SAVED_CASE)
5 psspy.rstr(SNAP_FILE)
```

This loaded the network data using `psspy.case` and the dynamic snapshot using `psspy.rstr`. Remember these two functions. You can call them again to reset a case after you've run a simulation.

15.4 Running a dynamic study

With the network and dynamics data already loaded into memory, the next step is to run a dynamics simulation. This example will apply a bus fault, clear it and return the generator voltage angles as a Python list.

PSS®E has a recommended pattern for dynamic studies

- STRT - initialise dynamic simulation at $t=-2*DELT$
- RUN - system runs to $t=0-$ just before the fault
- ALTR - apply a fault to the system
- RUN - run the system for the duration of the fault e.g. $t=0.1$
- ALTR - remove the fault from the system
- RUN - simulate the result for another few seconds e.g. $t=5.0$

We'll map these manual PSS®E commands to Python functions. You'll find more about the recommended pattern in Chapter 13 of the Program Application Guide II.

STRT

Function Signature

```
psspy.strt([option=0][, outfile=''])
```

Function Arguments:

- *option* monitor network convergence.
 - 1** automatically print the convergence monitor
 - 0** print only if it is enabled via the **CM** interrupt control code. Default
- *outfile* name of the channel output file, leave blank to bypass recording output.

The following code will save the recording output in a temporary file ending in `.out`.

```
1 import tempfile
2
3 outfile = tempfile.mktemp(suffix='.out')
4 psspy.strt(outfile=outfile)
```

RUN

Function Signature

```
psspy.run([option=0], tpause[, nprt][, nplt][, crtplt])
```

Function Arguments:

- *option* monitor network convergence.
 - 1** automatically print the convergence monitor
 - 0** print only if it is enabled via the CM interrupt control code. Default
- *tpause* run the simulation and pause at this time
- *nprt* print output channel values every *nprt*-th timestep.
- *nplt* write output channel to file every *nplt*-th timestep.
- *crtplot* plot output channel values every *crtplot*-th timestep.

The following will run the study until $t=1.05$.

```
1 psspy.run(tpause=1.05)
```

CHAN

Function Signature

```
psspy.machine_array_channel(status, id[, ident])
```

Function Arguments:

- *status* A tuple consisting of *channel index number*, *machine variable recorded* and *machine bus number* See Section 4.2.2.2 in the API manual for a list of the available machine variables.
 - status1** *channel index number*, default is -1, next available index
 - status2** *machine variable recorded* default 1 (machine angle), See Section 4.2.2.2 of the API manual.
 - status3** *machine bus number* no default.
- *id* Machine Id
- *ident* Channel label, if blank PSS®E will create one for you.

The following will record machine angle on the next available channel.

```
1 psspy.machine_array_channel(  
2     status2=1,      # ANGLE  
3     status3=101,    # BUS No.  
4     id='1')
```

ALTR

Function Signature

```
psspy.dist_bus_fault(ibus[, units=1][, basekv=0.0][, values=(0.0, -2.0E11)])
```

Function Arguments:

- *ibus* Bus number to place the fault at
- *units* units the fault admittance is specified in
 - 1 admittance in MVA Default.
 - 2 admittance in mhos
 - 3 admittance in ohms
- *basekv* Use this base voltage kV to calculate per unit fault admittance. If 0.0 use the base voltage of *ibus*
- *values* Real and reactive fault admittance (0.0, -2.0E11)

This code will apply a bus fault at bus 101, and run the study until $t=1.05$

```
1 psspy.dist_bus_fault(101)
2 psspy.run(tpause=1.05)
```

Function Signature

```
psspy.dist_clear_fault([ifault=1])
```

Function Arguments:

- *ifault* Index of the fault to clear. Faults are stored in memory from least recent to most recent. Least recent fault has the index 1.

We apply two bus faults, and then clear first and later clear the second.

```
1 psspy.dist_bus_fault(101)
2 psspy.run(tpause=1.01)
3 psspy.dist_bus_fault(102)
4 psspy.run(tpause=1.05)
5 psspy.dist_clear_fault(1)
6 psspy.run(tpause=1.07)
7 psspy.dist_clear_fault(1)
```

15.4.1 Getting Dynamics Results

The PSS[®]E dyntools has an easy class to read the data directly from the channel output file `dyntools.CHNF()`.

Class Signature

```
dyntools.CHNF([*outfiles])
```

Class Arguments:

- *outfile* A comma separated list of filenames for channel output files

Class Methods:

- `get_data([channels=''], outfile='')`
 - *channels* channel numbers to extract specified as a single number or list of numbers. Leave blank and PSS®E will extract all channels.
 - *outfile* extract channels from a specific outfile if more than one was given to CHNF when it was initialised. Otherwise returns channels from the first outfile.

Return Values:

- (*short_title*, *chanid* and *chandata*)

This code will read machine angle from the first channel in a channel output file and put the result into a Python list.

```
1 outfile = 'machine-angles-output.out'
2 channel = dyntools.CHNF(outfile)
3 title, columns, data = channel.get_data()
4
5 # first channel in this example has machine angle.
6 machine_angles = data[1]
```

15.4.2 Putting it together

In this section, we describe a short script that applies a bus fault with three different clearing times, 0.05, 0.1 and 0.15. The machine angles are plotted on a chart and saved as an image file named like `angles-clearing-time-0.05.png` for a report.

```
1 import psse34
2 import psspy
3 psspy.setThrowPsseExceptions(True)
4 import dyntools
5
6 import pylab
7
8 clearing_times = [0.05, 0.1, 0.15]
9
10 for clearing in clearing_times:
11
12     # reset the network and dynamic data.
13     psspy.case(SAV_FILENAME)
14     psspy.rstr(SAV_FILENAME)
15
16     # record machine angle
17     psspy.machine_array_channel(
18         status2=1,      # ANGLE
19         status3=101,    # bus.
20         id='1')
```

```
21
22     psspy.strt(outfile='output.out')
23     psspy.run(tpause=1.0)           # run to 1 second.
24     psspy.dist_bus_fault(101)      # apply fault
25     psspy.run(tpause=1 + clearing)  # run a short time.
26     psspy.dist_clear_fault(1)      # clear fault
27     psspy.run(tpause=5.0)          # run to 5 seconds.
28
29     # get the recorded machine angle.
30     channel = dyntools.CHNF(['output.out'])
31     title, columns, data = channel.get_data()
32     os.remove('output.out')
33
34     # channel 1 is machine angle.
35     angles = data[1]
36
37     pylab.figure()
38     pylab.plot(angles)
39     pylab.savefig('angles-clearing-time-%0.2f.png' % clearing)
```

By way of short description, the script loops three times. Each time the network and dynamics data are reset using `psspy.case` and `psspy.rstr`.

We set the channel to monitor machine angle. Then run through the actual dynamics portion. Which has the following scenario:

initial run to time 1 second

disturbance apply a bus fault for a short period of time

clear clear the fault and run the rest of the simulation to 5 seconds.

Finally, we take the machine angle data in a Python list and plot on a graph ready for a written report.

15.5 Example

15.5.1 Introduction

Great work so far! Your work studying the effect of the DGFIFTY machines has caught the attention of the *lead engineer*. She comes over to your desk and hands you a PSS®E dongle.

“This dongle is worth \$40,000 because it can run dynamic studies. I trust that you can calculate if the critical clearing time for these DGFIFTY machines meets the grid code. ”

Looking at the dongle, you marvel at how something that is just a fancy USB stick could cost as much as a car.

“Yes Ma’am, I’d love to run these dynamic studies to determine the critical clearance time. I’ll have the results to you within the hour.”

You will be given the following files:

- `savcnv.sav` This is the base case for these studies. We are using a different example network that has dynamic models built in.
- `savnw.snp` This is the dynamics data snapshot for the network you loaded above. Normally it takes some time to develop and tune the models for an entire network. So you get one that was prepared earlier.
- `fault.py` This is a fault script you had written while you were still a student. You’ll work on this script so that it records both machine angles and bus voltages.
- `settings.py` This file has your script variables. It includes the clearing time that you’ll study as well as listing the input `sav` and `snp` files.

You’ll create this file

The fault study will write a suite of charts showing machine angle and bus voltage. Two in total for the clearing time you study.

15.5.2 Task 1

Right now the script makes graphs that are unlabelled. Update the code so that the graph has a title “Machine Angle for <bus> and clearing time <time>” You should replace the <bus> and <time> with the actual values you use in the script.

15.5.3 Task 2

Begin recording bus voltage at the faulted machine bus. Plot that on it’s own chart too. So you’ll be making two charts for the clearing time:

- machine angle; and
- bus voltage in pu.

15.5.4 Task 3

Running only one clearing time in the script seems a bit underpowered. Re-write the code so that you can run 5 different clearing times instead of 1.

The script should create a total of 10 output graphs, two (angle and voltage) for each of the 5 different clearing times.

15.5.5 Task 4

Extension question. Don't worry if you can't finish this one. Either develop a program, or describe how you'd write a program to automatically detect if a simulation was stable or not after the fault is cleared. Could you use this program to automatically calculate the critical clearing time?