

Solutions to “Programming in Python 3: A Complete Introduction to the Python Language”, by Mark Summerfield

Solutions by: Lachlan Marnham

Last updated: April 2, 2018

Contents

Preface	2
Chapter 1	3
Exercise 1	3
Exercise 2	4
Exercise 3	6
Exercise 4	7
Exercise 5	8
Chapter 2	12
Exercise 1	12
Exercise 2	12
Exercise 3	14
Exercise 4	15

Preface

I like to solve textbook problems for fun. I am providing them for public consumption under the assumption that they will be used by people to complement self-study. They are not intended for use by school/university students who were assigned problems from the textbook by an instructor. If you are a school/university student who was assigned problems from the textbook by an instructor, and have found your way here with a mind to copy my solutions: 1) please don't do that, and 2) consider asking your instructor to assign their own problems so your peers can't do it either. If you're struggling with an exercise, try to think of a specific question the answer to which would solve your problem, and ask it in one of the forums I've linked to below.

Format

Python3Solutions.pdf contains all of the code, with example test runs. These examples are usually chosen to show the full behaviour of the code as much as possible (successful run, user enters incorrect input, etc). There are also (sometimes lengthy) explanations of what's going on in the code. If Python isn't your first programming language, you can likely skip over those. But hopefully someone will find them useful. If you do want to try running my code, copy and pasting from the .pdf probably isn't the way to go. You can find all of the solutions in .py form [on my GitHub page](#).

Packages

I've made an effort to, given a certain exercise, only use the bits of Python which are covered in the textbook prior to that exercise. This leads to some examples of very verbose code which would be made much more elegant with a simple function available elsewhere, but I don't think that's in the spirit of the text. If I made a mistake somewhere, and you see a package imported somewhere which you don't have installed, you can [install it with pip](#).

Errata

If you find an error, typo or poor explanation anywhere, please do let me know: lachlan.marnham@gmail.com. There is bound to be at least one bug arising from my absent-mindedly writing python 2.x code and checking it with the corresponding interpreter.

Resources

- [Stack Overflow](#)
- [Python Subreddit](#)
- [Quora](#)

Chapter 1

Exercise 1

```
1 import sys
2
3 # Initialise the digits 0-9, and store them in a list called digits
4 ZERO = [ ' *** ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ' ]
5
6 ONE = [ ' * ', ' * ', ' * ', ' * ', ' * ', ' * ', ' * ', ' * ', ' * ', ' * ' ]
7 TWO = [ ' *** ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ' ]
8 THREE = [ ' *** ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ' ]
9 FOUR = [ ' * ', ' * ', ' * ', ' * ', ' * ', ' * ', ' * ', ' * ', ' * ', ' * ' ]
10 FIVE = [ ' *** ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ' ]
11 SIX = [ ' *** ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ' ]
12 SEVEN = [ ' *** ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ' ]
13 EIGHT = [ ' *** ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ' ]
14 NINE = [ ' *** ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ', ' *  * ' ]
15
16 digits = [ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE]
17
18 try:
19     # Read in the argument from the command line, store as a string
20     number_string = sys.argv[1]
21
22     row = 0
23     while row < 7:
24         line = ''
25         for digit in number_string:
26             dig_int = int(digit)
27             # a = digits[dig_int] locates the right number in the list
28             # digits. b = a[row] locates the right row of a.
29             # b.replace('*', digit) replaces each astrix in b with the number
30             # corresponding to digit. digits[dig_int][row].replace('*', digit)
31             # does this all in one step.
32             line += digits[dig_int][row].replace('*', digit) + ' '
33         print(line)
34         row += 1
35
36 # Don't raise if the user forgets to enter an argument at the command line
37 except IndexError:
38     print("Error, try: python Ch01Ex01.py <integer>.")
39
40 # Don't raise if the argument entered at the command line is not an int
41 except ValueError:
42     print("Error: argument must be an int.")
```

This is what we get on the terminal if we enter a valid integer (in this case I just entered 1234567890 to show what each digit looks like):

```
[lachy@workstation]$ python3 Ch01Ex01.py 1234567890
1      222      333      4      55555      666      77777      888      9999      000
11     2  2  3  3      44      5      6      7  8  8  9  9  0  0
1     2  2      3  4  4      5      6      7  8  8  9  9  0  0
1      2      33  4  4      5555      6666      7      888      9999  0  0
1      2      3  444444      5  6  6  7      8  8  9  0  0
1     2      3  3      4      5  5  6  6  7      8  8  9  0  0
111   22222      333      4      555      666      7      888      9      000
```

This is what happens if we enter invalid input. In the first case, I fail to give an argument to the call. This results in an `IndexError` being raised. In the second case I remember, but the program can't decide how to convert the argument into an integer. This results in a `ValueError` being raised.

```
[lachy@workstation]$ python3 Ch01Ex01.py
At the command line, use the format: Ch01Ex01.py <integer>.
[lachy@workstation]$ python3 Ch01Ex01.py onetwothreefour
Please enter a valid integer as the argument at the command line.
```

Exercise 2

```
1 # initialise relevant objects
2 numbers = []
3 count = 0
4 total = 0
5 lowest = None
6 highest = None
7 TEMPLATE = 'count = {0} sum = {1} lowest = {2} highest = {3} mean = {4}'
8
9 # while loop will continue to seek new inputs until the user enters Enter, at
10 # which point loop terminates
11 while True:
12     new_input = input('enter a number or Enter to finish: ')
13     if new_input != '':
14         # attempt to convert user input to a number. If input is not
15         # convertible to a number, catch the exception and warn the user
16         try:
17             new_number = float(new_input)
18         except ValueError:
19             print('invalid input!')
20         numbers.append(new_number)
21         count += 1
22         total += new_number
23         if lowest is None:
24             lowest = new_number
25             highest = new_number
26         elif new_number < lowest:
27             lowest = new_number
28         elif new_number > highest:
29             highest = new_number
30     else:
31         break
32
33 # calculate the mean of the numbers and print results
34 try:
35     mean = total / count
36     print('numbers: ', numbers)
37     print(TEMPLATE.format(count, total, lowest, highest, mean))
38 except ZeroDivisionError:
39     print('no numbers were entered')
```

The first exception handler catches the `ValueError` which would occur when the user enters a string which doesn't correspond to a number:

```
[lachy@workstation]$ python3 Ch01Ex02.py
enter a number or Enter to finish: 1
enter a number or Enter to finish: 5.5
enter a number or Enter to finish: -2
enter a number or Enter to finish: hello
invalid input!
enter a number or Enter to finish: 3.14159
enter a number or Enter to finish:
numbers: [1.0, 5.5, -2.0, -2.0, 3.14159]
count = 5 sum = 5.64159 lowest = -2.0 highest = 5.5 mean = 1.128318
```

The second catches the `ZeroDivisionError` which would result from trying to calculate the mean when the user decided to not enter any numbers at all:

```
[lachy@workstation]$ python3 Ch01Ex02.py
enter a number or Enter to finish:
no numbers were entered
```

The problem statement says the user should be able to enter numbers, but the only example usage given seems to assume these numbers will be of type `int` (check out the output in the example, where numbers are rendered like 3 instead of 3.0). There are a number of ways to make the output like that of the author. One can:

- explicitly assume that all numbers are integers (a bizarre assumption) along the lines of:

```
user_input = input('enter a number or Enter to finish: ')
try:
    user_number = int(user_input)
except ValueError:
    print("you didn't enter a number")
```

Note that we hit the `except` suite here if the user enters, e.g., 3.1,

- employ more complicated string formatting when printing results to the screen. Let's not do that because it hasn't been covered in the textbook when one arrives at these exercises,
- convert the input to `float` and then convert *that* to `int` later, if acceptable:

```
try:
    user_input = input('enter a number: ')
    user_number = float(user_input)
except ValueError:
    print('not a number')
if user_number == int(user_number):
    user_number = int(user_number)
```

this works in principle, but *why?*,

- Nest the exception handling. In this way, if an exception is hit because the input was not an `int`, the code will try to convert to a `float` instead. This method is ugly, potentially confusing and computationally suboptimal (exception handling can be expensive):

```
try:
    user_input = input('enter a number: ')
    user_number = int(user_input)
except ValueError:
    try:
        user_number = float(user_input)
    except ValueError:
        print('not a number')
```

I've assumed that the author didn't implement any of these, and instead made a silly mistake in assuming that all inputs would be of type `int`. The code I've given above relaxes this assumption.

One might complain that this has only relaxed the problem enough to take account of the full set of real numbers. True, but the author wanted us to print the largest and smallest of the set of numbers entered, and it's not possible to even define the concept of 'greater than' or 'less than' across the rest of the complex plane. I think it's safe to assume we can stick to the real numbers.

Exercise 3

```
1 from random import choice, randint
2
3 ARTICLES = ('the', 'a', 'an', 'one')
4 SUBJECTS = ('cat', 'dog', 'man', 'woman', 'aligator', 'emu', 'iguana',
5             'octopus', 'urial')
6 VERBS = ('sang', 'ran', 'jumped')
7 ADVERBS = ('loudly', 'quietly', 'well', 'badly')
8 VOWELS = ('a', 'e', 'i', 'o', 'u')
9
10
11 def get_words():
12     words = {'subject': choice(SUBJECTS),
13             'verb': choice(VERBS),
14             'adverb': choice(ADVERBS)}
15
16     article = choice(ARTICLES)
17     # subjects starting with a vowel/consonant should not follow 'a'/'an'
18     if article == 'a' and words['subject'].startswith(VOWELS):
19         article = 'an'
20     elif article == 'an' and not words['subject'].startswith(VOWELS):
21         article = 'a'
22     words['article'] = article
23     # Remove adverb if randint() returns 1
24     if randint(0, 1):
25         words.pop('adverb')
26     return words
27
28
29 # Vocabulary-to-phrase maker
30 def make_line(words):
31     line = words['article'] + ' ' + words['subject'] + ' ' + words['verb']
32     if 'adverb' in words:
33         line += ' ' + words['adverb']
34     return line
35
36
37 def make_poem():
38     lines = []
39     for i in range(5):
40         words_in_line = get_words()
41         # The final line should end with a full stop, and all other lines
42         # end in commas.
43         if i == 4:
44             lines.append(make_line(words_in_line) + '.')
45         else:
46             lines.append(make_line(words_in_line) + ',')
47     # The poem should begin with a capital letter
48     return '\n'.join(lines).capitalize()
49
50
51 print(make_poem())
```

There are a couple of features in the code above which aren't required by the problem statement (mostly a simple implementation of grammar). Firstly, when a noun begins with a vowel, the article preceding it is (usually) not 'a'. Similarly, words starting with a consonant usually do not follow 'an'. This is implemented on lines 18-21. We also put a comma after all but the last line, which is in turn terminated by a full-stop. Finally, the first line of the poem is capitalised.

An example run follows:

```
[lachy@workstation]$ python3 Ch01Ex03.py
An urial jumped loudly,
one man jumped well,
an emu jumped loudly,
a woman sang quietly,
a dog sang badly.
```

Exercise 4

```
1 from random import choice, randint
2
3 ARTICLES = ('the', 'a', 'an', 'one')
4 SUBJECTS = ('cat', 'dog', 'man', 'woman', 'aligator', 'emu', 'iguana',
5             'octopus', 'urial')
6 VERBS = ('sang', 'ran', 'jumped')
7 ADVERBS = ('loudly', 'quietly', 'well', 'badly')
8 VOWELS = ('a', 'e', 'i', 'o', 'u')
9
10
11 def get_words():
12     words = {'subject': choice(SUBJECTS),
13             'verb': choice(VERBS),
14             'adverb': choice(ADVERBS)}
15
16     article = choice(ARTICLES)
17     # subjects starting with a vowel/consonant should not follow 'a'/'an'
18     if article == 'a' and words['subject'].startswith(VOWELS):
19         article = 'an'
20     elif article == 'an' and not words['subject'].startswith(VOWELS):
21         article = 'a'
22     words['article'] = article
23     # Remove adverb if randint() returns 1
24     if randint(0, 1):
25         words.pop('adverb')
26     return words
27
28
29 def make_line(words):
30     line = words['article'] + ' ' + words['subject'] + ' ' + words['verb']
31     if 'adverb' in words:
32         line += ' ' + words['adverb']
33     return line
34
35
36 def make_poem(number_of_lines):
37     lines = []
38     for i in range(number_of_lines):
39         words_in_line = get_words()
40         # The final line should end with a full stop, and all other lines
41         # end in commas.
42         if i == number_of_lines - 1:
43             lines.append(make_line(words_in_line) + '.')
44         else:
45             lines.append(make_line(words_in_line) + ',')
46     # The poem should begin with a capital letter
47     return '\n'.join(lines).capitalize()
48
49
50 while True:
51     user_input = input('Enter poem length between 1 and 10 lines: ')
52     try:
53         poem_length = int(user_input)
54         if not 0 < poem_length <= 10:
55             raise ValueError
56     except ValueError:
57         if user_input == '':
58             poem_length = 5
59         else:
60             print('Invalid input. Please try again.')
61             continue
62     print(make_poem(poem_length))
63     break
```

The main difference here compared to the previous exercise, is that we now allow the user to define the poem length. This manifests itself in some basic exception handling and a `while` loop which will continue until valid input is given.

If the user enters an input which Python doesn't know how to cast to an `int` (e.g. a `str` of alphabetical characters, see below) a `ValueError` will be raised. We catch this exception, tell the user to try again,

and return control to the beginning of the loop. Entering an integer which is not in the allowed range ($0 < N_{\text{lines}} \leq 10$) won't automatically hit an exception, however, so we **raise** the `ValueError` explicitly so that it will be handled in the same way.

Here's an example output:

```
[lachy@workstation]$ python3 Ch01Ex04.py
Enter poem length between 1 and 10 lines: 7
The aligator sang,
one octopus ran quietly,
an aligator ran,
an iguana jumped badly,
one urial ran quietly,
a man ran,
a woman sang.
```

Here's a series of other possible ways the code can run. In the very final example, the user has entered **Enter**, and in this case the code defaults to $N_{\text{lines}} = 5$.

```
[lachy@workstation]$ python3 Ch01Ex04.py
Enter poem length between 1 and 10 lines: six
Invalid input. Please try again.
Enter poem length between 1 and 10 lines: 2.2
Invalid input. Please try again.
Enter poem length between 1 and 10 lines: -1
Invalid input. Please try again.
Enter poem length between 1 and 10 lines: 11
Invalid input. Please try again.
Enter poem length between 1 and 10 lines:
The aligator ran well,
an octopus ran quietly,
a cat ran quietly,
a dog ran badly,
the octopus jumped well.
```

Exercise 5

```
1 from math import floor
2
3
4 def find_average(number_list, with_extras=False):
5     running_sum = 0
6     for number in number_list:
7         running_sum += number
8     number_count = len(number_list)
9     average = running_sum / number_count
10    # with_extras is an optional flag which will return only average if set
11    # to False (which it is by default) and will also return the sum and
12    # sample size of the numbers if set to True
13    if with_extras:
14        return average, running_sum, number_count
15    else:
16        return average
17
18
19 def bubble_sort(number_list):
20     swapped_flag = True
21     while swapped_flag:
22         n = 0
23         swapped_flag = False
24         for i in range(len(number_list) - 1 - n):
25             if number_list[i] > number_list[i + 1]:
26                 number_list[i], number_list[i + 1] = number_list[i + 1], number_list[i]
27                 swapped_flag = True
28         n += 1
29
30
31 def find_median(my_list):
32     bubble_sort(my_list)
33     max_index = len(my_list) - 1
34     # If my_list contains an even number of elements, find the average of the
35     # two center-most elements. Otherwise, return the centre-most element
```

```

36     if len(my_list) % 2 == 0:
37         i = floor((max_index / 2))
38         # The with_extras flag takes its default value (False) here
39         return find_average(my_list[i:i+2])
40     else:
41         median_index = max_index // 2
42         return my_list[median_index]
43
44
45 # initialise relevant objects
46 numbers = []
47 count = 0
48 total = 0
49 lowest = None
50 highest = None
51 TEMPLATE = 'count = {0} sum = {1} lowest = {2} highest = {3} ' \
52            'mean = {4} median = {5}'
53
54 # while loop will continue to seek new inputs until the user enters Enter,
55 # at which point loop terminates
56 while True:
57     new_input = input('enter a number or Enter to finish: ')
58     if new_input != '':
59         # attempt to convert user input to a number. If input is not
60         # convertible to a number, catch the exception and warn the user
61         try:
62             new_number = float(new_input)
63         except ValueError:
64             print('invalid input!')
65         numbers.append(new_number)
66         if lowest is None:
67             lowest = new_number
68             highest = new_number
69         elif new_number < lowest:
70             lowest = new_number
71         elif new_number > highest:
72             highest = new_number
73     else:
74         break
75
76
77 # calculate the mean of the numbers and print results
78 try:
79     mean, total, count = find_average(numbers[:], with_extras=True)
80     median = find_median(numbers[:])
81     print('numbers: ', numbers)
82     print(TEMPLATE.format(count, total, lowest, highest, mean, median))
83 except ZeroDivisionError:
84     print('no numbers were entered')

```

A few things work differently here when compared to Exercise 2. Firstly the `find_median(...)` function will calculate the average of two values when the quantity of numbers provided is even. We are now calculating the average of some list of numbers in two different places, so we have included a dedicated function to handle this, which wasn't necessary before. Secondly, there is the matter of implementing a sorting algorithm in order to find the median.

There is no shortage of algorithms which could sort our numbers, but Bubble Sort strikes a nice balance between being very intuitive and not being *too* brute-forcey. Here's how it works:

1. Before looping through the numbers, initialize `swapped_flag` which is `False` until two numbers have been swapped on the current loop. It is then set to `True` until the end of that loop, when it becomes `False` again.
2. Look at the elements of the list with indices 0 and 1. If the latter is smaller than the former, swap them and set `swapped_flag==True`. Otherwise do nothing.
3. Repeat step 2 with elements 1 and 2, then 2 and 3, and so on for the whole list.
4. If, on the current run through the loop, none of the neighbouring elements were swapped, then at this stage `swapped_flag==False` and the list is sorted. Otherwise, set `swapped_flag==True` and return control to the beginning of the loop.

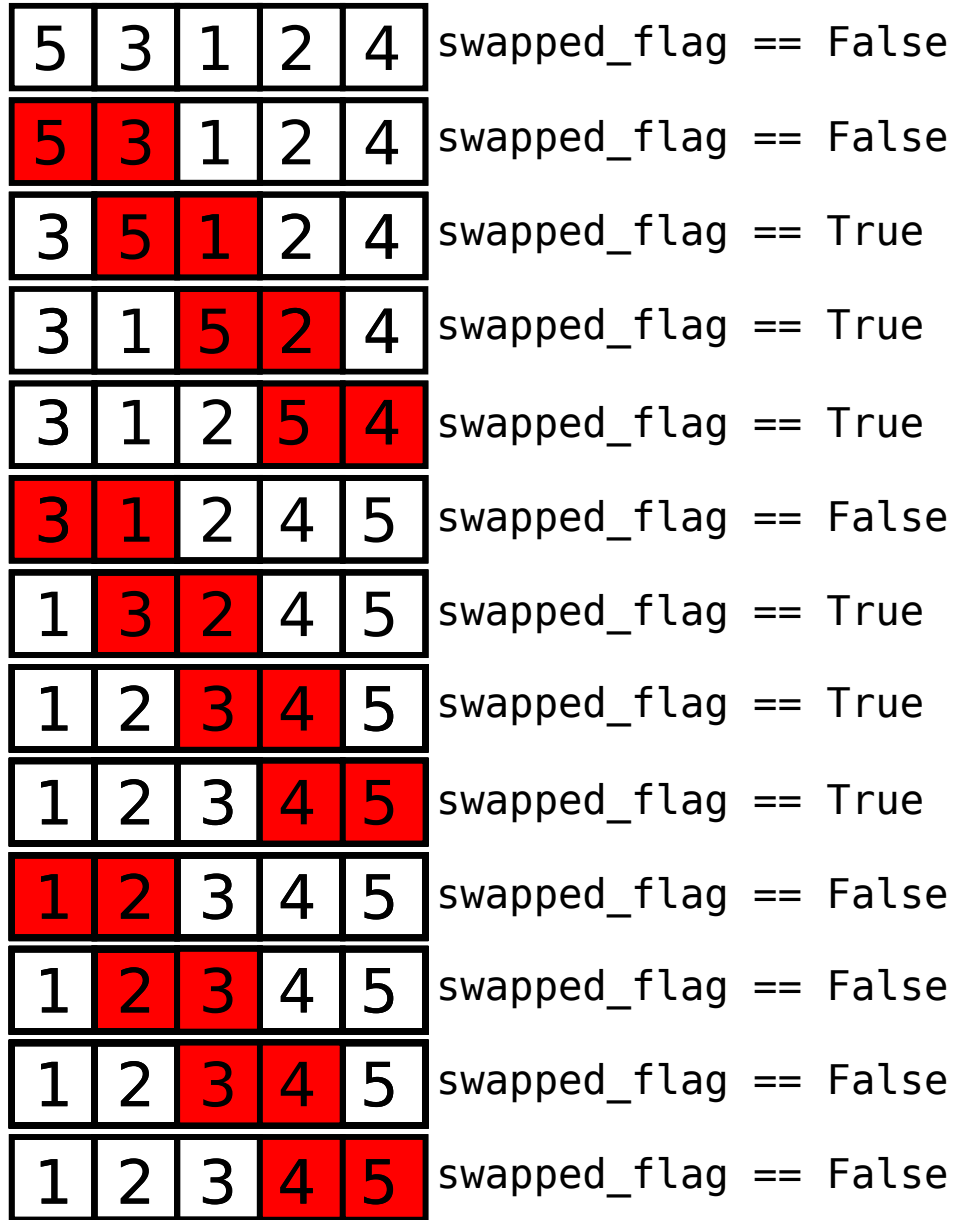


Figure 1: Schematic of Bubble Sort on a list.

As described above, we will end up with an algorithm which has worst-case complexity $(n - 1)^2 = \mathcal{O}(n^2)$ (where n is the length of the `list`). To see this, note that each time we run through the `while` loop we will check $n - 1$ pairs (and, perhaps, swap them). But in the worst-case scenario, we would have to run through that loop $n - 1$ times.

But it is never necessary to do so many steps, because on the first run through the loop, the largest number will definitely be in its final place. On the second run through the loop, the second-last number will be in place, and so on. In general, on the k^{th} run through the loop, we only need to check the $(n - k)^{\text{th}}$ pair of neighbours. The worst-case total number of steps is therefore:

$$\begin{aligned}
 \sum_{k=1}^{n-1} (n - k) &= n \sum_{k=1}^{n-1} 1 - \sum_{k=1}^{n-1} k \\
 &= n^2 - n - \frac{n^2 - n}{2} \\
 &= \frac{n^2 - n}{2}.
 \end{aligned} \tag{1}$$

Still $\mathcal{O}(n^2)$, but definitely faster. Therefore, this optimisation was implemented on line 24 of the code in the instantiation of the `for` loop.

Chapter 2

Exercise 1

```
1 import sys
2 import unicodedata
3
4
5 def print_unicode_table(word_list):
6     print("decimal    hex    chr  {0:~40}".format("name"))
7     print("-----  -")
8
9     code = ord(" ")
10    end = min(0xD800, sys.maxunicode) # Stop at surrogate pairs
11
12    while code < end:
13        c = chr(code)
14        name = unicodedata.name(c, "*** unknown ***")
15        for word in word_list:
16            if word is None or word in name.lower():
17                continue
18            else:
19                break
20        else:
21            print("{0:7} {0:5X} {0:3c} {1}".format(
22                code, name.title()))
23        code += 1
24
25
26 words = [None]
27 if len(sys.argv) > 1:
28     if sys.argv[1] in ("-h", "--help"):
29         print("usage: {0} [string_1] [string_2] ...".format(sys.argv[0]))
30         words = []
31     else:
32         words = [word.lower() for word in sys.argv[1:]]
33
34 if words:
35     print_unicode_table(words)
```

This exercise required only minimal changes to implement searching for multiple words. The program will read in the list of optional string arguments, should they be provided by the user. We then loop through all of the code points between `ord(" ")`==32 and `min(0xD800, sys.maxunicode)`. Here, the choice to start at 32 is because the `unicodedata` module does not store names for unicode characters with codepoints below 32. This includes some control characters (e.g. `code('\n')==10`). The maximum code point is chosen to:

- Always be less than the maximum unicode character available to the system, and
- Avoid being so large that surrogate pairs (where large code points are represented by two hex numbers instead of one) are excluded.

The code then loops through all of the code points in that range. For each one, it finds the corresponding name of the character. If all of the strings provided by the user at runtime are in the character name, it is printed (along with the code point in decimal and hex notation, and the character itself).

Exercise 2

```
1 import cmath
2 import math
3 import sys
4
5
6 def get_float(msg, allow_zero):
7     x = None
8     while x is None:
9         try:
10             x = float(input(msg))
11             if not allow_zero and abs(x) < sys.float_info.epsilon:
12                 print("zero is not allowed")
13                 x = None
14         except ValueError as err:
15             print(err)
16     return x
17
18
19 # Fix formatting of numbers like (-1-2j)-->-(1+2j) and (-1+2j)-->-(1-2j)
20 def fix_complex_sign(number):
21     if number.real < 0:
22         if number.imag < 0:
23             new_number = complex(abs(number.real), abs(number.imag))
24             changed_flag = True
25         else:
26             new_number = complex(abs(number.real), -abs(number.imag))
27             changed_flag = True
28     else:
29         new_number = number
30         changed_flag = True
31     if abs(new_number.real) < sys.float_info.epsilon:
32         new_number = complex(abs(new_number.real), new_number.imag)
33     return new_number, changed_flag
34
35
36 def make_equation(a, b, c, x_1, x_2):
37     a_template = "{a}x\N{SUPERSCRIPT TWO}"
38     b_template = "{sign_b} {b}x"
39     c_template = "{sign_c} {c}"
40     template = a_template
41     if abs(b) >= sys.float_info.epsilon:
42         template += b_template
43     if abs(c) >= sys.float_info.epsilon:
44         template += c_template
45     template += " = 0"
46
47     sign_x_1 = ''
48     sign_x_2 = ''
49     sign_b = ' +'
50     sign_c = ' +'
51     if type(x_1) == complex:
52         x_1, sign_changed_1 = fix_complex_sign(x_1)
53         if sign_changed_1:
54             sign_x_1 = '- '
55     if type(x_2) == complex:
56         x_2, sign_changed_2 = fix_complex_sign(x_2)
57         if sign_changed_2:
58             sign_x_2 = '- '
59     if b < 0:
60         sign_b = ''
61     if c < 0:
62         sign_c = ''
63     equation = (template + " \N{RIGHTWARDS ARROW} "
64                 "x = {sign_x_1}{x1}").format(**locals())
65     if x_2 is not None:
66         equation += " or x = {sign_x_2}{x2}".format(**locals())
67     return equation
68
69
```

```

70 print("ax\N{SUPERScript TWO} + bx + c = 0")
71 a = get_float("enter a: ", False)
72 b = get_float("enter b: ", True)
73 c = get_float("enter c: ", True)
74
75 x_1 = None
76 x_2 = None
77 discriminant = (b ** 2) - (4 * a * c)
78 if discriminant == 0:
79     x_1 = -(b / (2 * a))
80     if abs(x_1) < sys.float_info.epsilon:
81         x_1 = 0.0
82 else:
83     if discriminant > 0:
84         root = math.sqrt(discriminant)
85     else:
86         root = cmath.sqrt(discriminant)
87     x_1 = (-b + root) / (2 * a)
88     x_2 = (-b - root) / (2 * a)
89
90 print(make_equation(a, b, c, x_1, x_2))

```

Exercise 3

```

1 import sys
2 from xml.sax.saxutils import escape
3
4
5 def main():
6     # Get files from command line.
7     # Usage: python3 Ch02Ex03.py <input_filename> <output_filename>
8     input_filename = sys.argv[1]
9     output_filename = sys.argv[2]
10    max_width = 100
11    table = ""
12    table += print_start()
13    count = 0
14
15    # Read in data from file instead, instead of with call to input()
16    file_object = open(input_filename, "r")
17    data = file_object.read()
18    file_object.close()
19
20    # Convert data to html
21    lines = data.splitlines()
22    for line in lines:
23        try:
24            if count == 0:
25                color = "lightgreen"
26            elif count % 2 == 0:
27                color = "white"
28            else:
29                color = "lightyellow"
30            table = print_line(table, line, color, max_width)
31            count += 1
32        except EOFError:
33            break
34    table += print_end()
35
36    # Dump html into output file
37    output_file_object = open(output_filename, "w")
38    output_file_object.write(table)
39    output_file_object.close()
40
41
42 def print_start():
43     return "<table border='1'>\n"
44
45
46 def print_end():
47     return "</table>\n"

```

```

48
49
50 def print_line(table, line, color, max_width):
51     table += "<tr bgcolor='{0}'>\n".format(color)
52     fields = extract_fields(line)
53     for field in fields:
54         if not field:
55             table += "<td></td>\n"
56         else:
57             number = field.replace(",", "")
58             try:
59                 x = float(number)
60                 table += "<td align='right'>{0:d}</td>\n".format(round(x))
61             except ValueError:
62                 field = field.title()
63                 field = field.replace(" And ", " and ")
64                 if len(field) <= max_width:
65                     field = escape(field)
66                 else:
67                     field = "{0} ...".format(escape(field[:max_width]))
68                 table += "<td>{0}</td>\n".format(field)
69     table += "</tr>\n"
70     return table
71
72
73 def extract_fields(line):
74     fields = []
75     field = ""
76     quote = None
77     for c in line:
78         if c in "\"'":
79             if quote is None:           # start of quoted string
80                 quote = c
81             elif quote == c:           # end of quoted string
82                 quote = None
83             else:
84                 field += c              # other quote inside quoted string
85                 continue
86         if quote is None and c == ",": # end of a field
87             fields.append(field)
88             field = ""
89         else:
90             field += c                  # accumulating a field
91     if field:
92         fields.append(field)           # adding the last field
93     return fields
94
95
96 main()

```

Exercise 4