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

Solutions by: Lachlan Marnham

Last updated: March 20, 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

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 a 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) explainations 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 explaination 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

```
import sys

# Initialise the digits 0-9, and store them in a list called digits
ZERO = [' *** ', '*   *', '*   *', '*   *', '*   *', '*   *', ' *** ']
ONE = [' * ', '** ', ' * ', ' * ', ' * ', ' * ', '***']
TWO = [' *** ', '*   *', '    *', '   * ', '  *  ', ' *   ', '*****']
THREE = [' *** ', '*   *', '    *', '  ** ', '    *', '*   *', ' *** ']
FOUR = ['   *  ', '  **  ', ' * *  ', '*  *  ', '******', '   *  ', '   *  ']
FIVE = ['*****', '*    ', '*    ', '**** ', '    *', '*   *', ' *** ']
SIX = [' *** ', '*    ', '*    ', '**** ', '*   *', '*   *', ' *** ']
SEVEN = ['*****', '    *', '   * ', '  *  ', ' *   ', '*    ', '*    ']
EIGHT = [' *** ', '*   *', '*   *', ' *** ', '*   *', '*   *', ' *** ']
NINE = [' ****', '*   *', '*   *', ' ****', '    *', '    *', '    *']

digits = [ZERO, ONE, TWO, THREE, FOUR, FIVE, SIX, SEVEN, EIGHT, NINE]

try:
    # Read in the argument from the command line, store as a string
    number_string = sys.argv[1]

    row = 0
    while row < 7:
        line = ''
        for digit in number_string:
            dig_int = int(digit)
            # a = digits[dig_int] locates the right number in the list
            # digits. b = a[row] locates the right row of a.
            # b.replace('*', digit) replaces each astrix in b with the number
            # corresponding to digit. digits[dig_int][row].replace('*', digit)
            # does this all in one step.
            line += digits[dig_int][row].replace('*', digit) + '  '
        print(line)
        row += 1

# Don't raise if the user forgets to enter an argument at the command line
except IndexError:
    print("Error, try: python Ch01Ex01.py <integer>.")

# Don't raise if the argument entered at the command line is not an int
except ValueError:
    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

```
# initialise relevant objects
numbers = []
count = 0
total = 0
lowest = None
highest = None
TEMPLATE = 'count = {0} sum = {1} lowest = {2} highest = {3} mean = {4}'

# while loop will continue to seek new inputs until the user enters Enter, at
# which point loop terminates
while True:
    new_input = input('enter a number or Enter to finish: ')
    if new_input != '':
        # attempt to convert user input to a number. If input is not
        # convertible to a number, catch the exception and warn the user
        try:
            new_number = float(new_input)
        except ValueError:
            print('invalid input!')
        numbers.append(new_number)
        count += 1
        total += new_number
        if lowest is None:
            lowest = new_number
            highest = new_number
        elif new_number < lowest:
            lowest = new_number
        elif new_number > highest:
            highest = new_number
    else:
        break

# calculate the mean of the numbers and print results
try:
    mean = total / count
    print('numbers: ', numbers)
    print(TEMPLATE.format(count, total, lowest, highest, mean))
except ZeroDivisionError:
    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

```
from random import choice, randint

ARTICLES = ('the', 'a', 'an', 'one')
SUBJECTS = ('cat', 'dog', 'man', 'woman', 'aligator', 'emu', 'iguana',
            'octopus', 'urial')
VERBS = ('sang', 'ran', 'jumped')
ADVERBS = ('loudly', 'quietly', 'well', 'badly')
VOWELS = ('a', 'e', 'i', 'o', 'u')


def get_words():
    words = {'subject': choice(SUBJECTS),
             'verb': choice(VERBS),
             'adverb': choice(ADVERBS)
             }
    article = choice(ARTICLES)
    # subjects starting with a vowel/consonant should not follow 'a'/'an'
    if article == 'a' and words['subject'].startswith(VOWELS):
        article = 'an'
    elif article == 'an' and not words['subject'].startswith(VOWELS):
        article = 'a'
    words['article'] = article
    # Remove adverb if randint() returns 1
    if randint(0, 1):
        words.pop('adverb')
    return words


def make_line(words):
    line = words['article'] + ' ' + words['subject'] + ' ' + words['verb']
    if 'adverb' in words:
        line += ' ' + words['adverb']
    return line


def make_poem(number_of_lines):
    lines = []
    for i in range(number_of_lines):
        words_in_line = get_words()
        # The final line should end with a full stop, and all other lines
        # end in commas.
        if i == number_of_lines - 1:
            lines.append(make_line(words_in_line) + '.')
        else:
            lines.append(make_line(words_in_line) + ',')
    # The poem should begin with a capital letter
    return '\n'.join(lines).capitalize()


while True:
    user_input = input('Enter poem length between 1 and 10 lines: ')
    try:
        poem_length = int(user_input)
        if not 0 < poem_length <= 10:
            raise ValueError
    except ValueError:
        if user_input == '':
            poem_length = 5
        else:
            print('Invalid input. Please try again.')
            continue
    print(make_poem(poem_length))
    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

```
from math import floor


def find_average(number_list, with_extras=False):
    running_sum = 0
    for number in number_list:
        running_sum += number
    number_count = len(number_list)
    average = running_sum / number_count
    # with_extras is an optional flag which will return only average if set
    # to False (which it is by default) and will also return the sum and
    # sample size of the numbers if set to True
    if with_extras:
        return average, running_sum, number_count
    else:
        return average


def bubble_sort(number_list):
    swapped_flag = True
    while swapped_flag:
        n = 0
        swapped_flag = False
        for i in range(len(number_list) - 1 - n):
            if number_list[i] > number_list[i + 1]:
                number_list[i], number_list[i + 1] = number_list[i + 1], number_list[i]
                swapped_flag = True
        n += 1


def find_median(my_list):
    bubble_sort(my_list)
    max_index = len(my_list) - 1
    # If my_list contains an even number of elements, find the average of the
    # two center-most elements. Otherwise, return the centre-most element
```

```
    if len(my_list) % 2 == 0:
        i = floor((max_index / 2))
        # The with_extras flag takes its default value (False) here
        return find_average(my_list[i:i+2])
    else:
        median_index = max_index // 2
        return my_list[median_index]


# initialise relevant objects
numbers = []
count = 0
total = 0
lowest = None
highest = None
TEMPLATE = 'count = {0} sum = {1} lowest = {2} highest = {3} ' \
           'mean = {4} median = {5}'

# while loop will continue to seek new inputs until the user enters Enter,
# at which point loop terminates
while True:
    new_input = input('enter a number or Enter to finish: ')
    if new_input != '':
        # attempt to convert user input to a number. If input is not
        # convertible to a number, catch the exception and warn the user
        try:
            new_number = float(new_input)
        except ValueError:
            print('invalid input!')
        numbers.append(new_number)
        if lowest is None:
            lowest = new_number
            highest = new_number
        elif new_number < lowest:
            lowest = new_number
        elif new_number > highest:
            highest = new_number
    else:
        break


# calculate the mean of the numbers and print results
try:
    mean, total, count = find_average(numbers[:], with_extras=True)
    median = find_median(numbers[:])
    print('numbers: ', numbers)
    print(TEMPLATE.format(count, total, lowest, highest, mean, median))
except ZeroDivisionError:
    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.

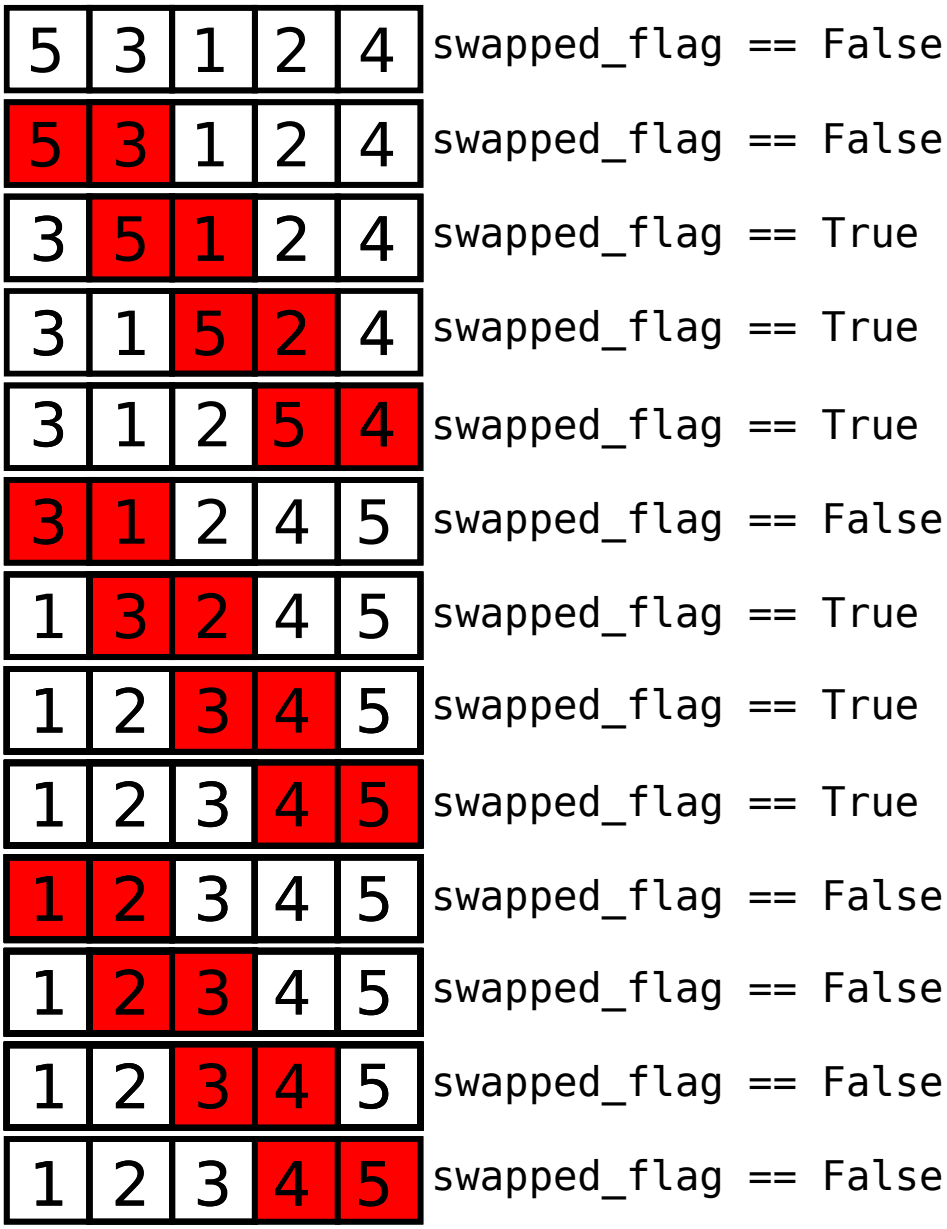

Figure 1: Schematic of Bubble Sort on a list.

As described above, we will end up with an algorithm which has worst–case complexity $(n-1)^2 = \mathcal{O}\left(n^2\right)$ (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 $\mathcal{O}\left(n^2\right)$, 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