

Strings and Tuples

CSE/IT 107L

NMT Department of Computer Science and Engineering

“All thought is a kind of computation.”

— D. Hobbes

“It [programming] is the only job I can think of where I get to be both an engineer and artist. There’s an incredible, rigorous, technical element to it, which I like because you have to do very precise thinking. On the other hand, it has a wildly creative side where the boundaries of imagination are the only real limitation.”

— A. Hertzfeld

Introduction

This lab will introduce two new Python values that are very similar to lists: tuples and strings. You have seen strings before, but now you will learn to manipulate them. Strings and tuples support indexing, slicing, membership testing with `in`, equality checks, and other similar features. The main difference is that they are immutable, meaning that the elements at each index cannot be changed.

The lab also introduces keyword arguments, which are useful for multi-purpose functions or for those with arguments that have default values. With a more detailed knowledge of strings, you will be able to write documentation for functions in the form of docstrings, which will be visible using the `help()` function.

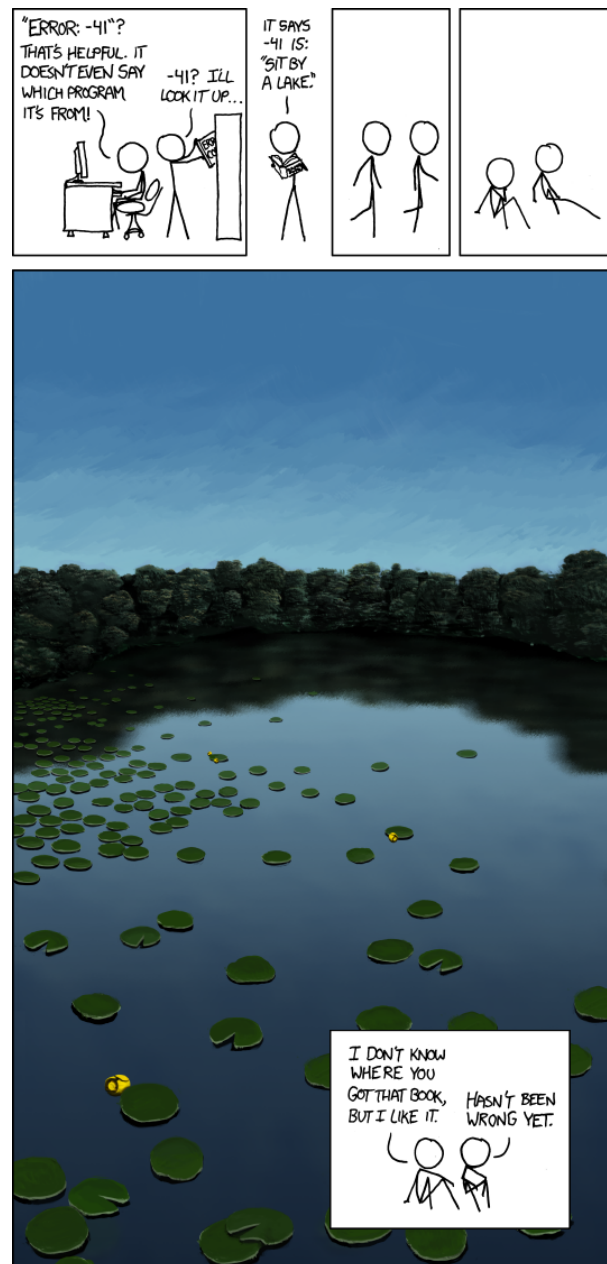


Figure 1: <http://xkcd.com/1024>

Contents

1	Tuples: Fixed-Length, Immutable Ordered Collections	1
2	Defining, Using, and Transforming Strings	2
2.1	Insert Special Characters by Escaping Them <code>"\", \', \\</code>	2
2.2	Multi-Line Strings Using <code>"""</code>	3
2.3	Checking String Properties Using <code>len</code> , <code>in</code> , <code>==</code> , <code>!=</code>	3
2.3.1	Checking String-Specific Properties	4
2.4	Indexing and Slicing Strings	5
2.5	Creating New Strings	6
2.5.1	Concatenation and Repetition Using <code>+</code> , <code>*</code>	6
2.5.2	Replacing Substrings With <code>string.replace(old, new)</code>	7
2.5.3	Creating Uppercase and Lowercase Strings	7
2.6	Splitting a String Into a List of Strings Using <code>str.split()</code>	7
3	Documenting Your Functions and Modules With Docstrings	8
4	Optional Function Arguments: Keyword Arguments	9
5	Exercises	10
	Submitting	13

1 Tuples: Fixed-Length, Immutable Ordered Collections

Tuples work a lot like lists, except they cannot be modified; they are *immutable*. Instead of brackets [], tuples are delimited by writing parentheses () around the elements. Tuples support slicing, just as lists do.

```

1 >>> food = ('eggs', 'bananas', 'lemonheads')
2 >>> food[1]
3 'bananas'
4 >>> food[1:3]
5 ('bananas', 'lemonheads')
6 >>> food[1] = 'steak' # tuples are immutable!
7 Traceback (most recent call last):
8   File "<stdin>", line 1, in <module>
9 TypeError: 'tuple' object does not support item assignment
10 Traceback (most recent call last):
11   File "<stdin>", line 1, in <module>
12 TypeError: 'tuple' object does not support item assignment

```

Notice Python raises a `TypeError` with message “‘tuple’ object does not support item assignment” if the code attempts to assign to an item within a tuple. If a tuple contains a mutable value, such as a list, then that element can be modified without restriction.

We can nest tuples and lists interchangeably:

```

1 >>> points = [(1, 5), (3, 5), (0, 2), (1, 1), # shift+enter for multi-line
2 ...          (2, 1), (3, 1), (4, 2)]
3 >>> points[0]
4 (1, 5)
5 >>> fubar = ([1, 2, 3], [4, 5, (7, 8, 9)], (10, 11))
6 >>> fubar
7 ([1, 2, 3], [4, 5, (7, 8, 9)], (10, 11))

```

Tuples, like lists, can also be empty. Use `len()` to find the length of a tuple:

```

1 >>> foo = ()
2 >>> len(foo)
3 0

```

We can also define a tuple by simply separating values by commas: Conversely, we can *unpack* tuples, which means assigning each element of a tuple to a variable. For example:

```

1 >>> food = 'trail mix', 'nothing'
2 >>> tylerfood, chrisfood = food
3 >>> # equivalent to, but shorter than: tylerfood = food[0]; chrisfood = food[1]
4 >>> tylerfood
5 'trail mix'
6 >>> chrisfood
7 'nothing'

```

Unpacking is useful for assigning a fixed number of return values from a function.

2 Defining, Using, and Transforming Strings

You have seen strings in Python before. They are sequences of characters enclosed by either double quotes or single quotes; for example:

```
1 >>> s = "I'm a string."
2 >>> print(s)
3 I'm a string.
4 >>> r = 'I am also a string.'
5 >>> print(r)
6 I am also a string.
```

2.1 Insert Special Characters by Escaping Them "\" , \' , \\"

Notice how we did not use a single quote in the second string because it was enclosed (*delimited*) by single quotes. The proper way to use a single quote in a single quoted string or a double quote in a double quoted string is to use escape the quote with a backslash:

```
1 >>> s = "Previously, we said \"I'm a string.\"."
2 >>> print(s)
3 Previously, we said "I'm a string.".
4 >>> r = 'I\'m also a string.'
5 >>> print(r)
6 I'm also a string.
```

This is called *escaping* a character. We *escaped* the double quotes and single quote respectively so that Python did not think it was the end of the string.

You do not have to escape a single quote in a double quoted string and vice versa:

```
1 >>> s = "I can use single quotes ' ' here"
2 >>> print(s)
3 I can use single quotes ' ' here
4 >>> r = 'I can use double quotes "" here'
5 >>> print(r)
6 I can use double quotes "" here
```

If you want a string to go to a new line when printed, you use `\n` for that:

```
1 >>> "Explicit is better than implicit.\nSimple is better than complex."
2 'Explicit is better than implicit.\nSimple is better than complex.'
3 >>> print("Explicit is better than implicit.\nSimple is better than complex.")
4 Explicit is better than implicit.
5 Simple is better than complex.
```

What, however, if you actually need a `\` in a string? You can type two backslashes to escape the backslash:

```
1 >>> print("C:\\Users\\hrivera")
2 C:\Users\hrivera
```

2.2 Multi-Line Strings Using `"""`

If you want strings to go on for two or more lines of Python code, you have to enclose the string in three double quotes. Everything between the two triple-quotes will be a part of the string.

```
1 weizsaecker = """We in the older generation owe to young people not the
2 fulfillment of dreams but honesty. We must help younger people to
3 understand why it is vital to keep memories alive. We want to help them
4 to accept historical truth soberly, not one-sidedly, without taking
5 refuge in Utopian doctrines, but also without moral arrogance. From our
6 own history we learn what man is capable of. For that reason we must not
7 imagine that we are quite different and have become better. There is no
8 ultimately achievable moral perfection. We have learned as human beings,
9 and as human beings we remain in danger. But we have the strength to
10 overcome such danger again and again."""
```

Listing 1: Excerpt of Richard von Weizsäcker’s speech in the Bundestag to commemorate the 40th anniversary of the end of World War II.

```
1 >>> r = """I hear America singing, the varied carols I hear,
2 ... Those of mechanics, each one singing his as it should be blithe and strong,
3 ... The carpenter singing his as he measures his plank or beam,"""
4 >>> print(r)
5 I hear America singing, the varied carols I hear,
6 Those of mechanics, each one singing his as it should be blithe and strong,
7 The carpenter singing his as he measures his plank or beam,
```

Listing 2: Excerpt of *I Hear America Singing* by Walt Whitman. Notice the triple-dots in the Python interactive shell indicate that this is a multi-line segment of code.

2.3 Checking String Properties Using `len`, `in`, `==`, `!=`

Lists, tuples, and strings are sequence data types — this means they support indexing, slicing, and many of the functions that were introduced in the section describing lists in lab 4. A lot of other operations that work on lists also work on strings. You can test membership of a string using `in`.

```
1 >>> s = 'abcde'
2 >>> print('c' in s)
3 True
4 >>> print('f' in s)
5 False
6 >>> print('f' not in s)
7 True
```

```
8 >>> print('ab' in s)
9 True
```

You can measure the length of a string by writing `len()`.

```
1 >>> s = 'abcDE'
2 >>> print(len(s))
3 5
4 >>> len("")
5 0
6 >>> len("Hello world!")
7 12
```

2.3.1 Checking String-Specific Properties

There are several functions that you can use to check if a string contains certain sets or sequences of characters. For example, you can check if a string is all uppercase using the `s.isalpha()` function. These functions all start with “is” and return a boolean value.

```
1 >>> s = 'abcDE'
2 >>> print(s.isnumeric())
3 False
4 >>> s.isalpha()
5 True
6 >>> s.islower()
7 False
8 >>> print(s.isupper())
9 False
```

Here is a complete list:

`s.isalnum()` Return True if all characters in S are alphanumeric and there is at least one character in S, False otherwise.

`s.isdecimal()` Return True if there are only decimal characters in S, False otherwise.

`s.isidentifier()` Return True if S is a valid identifier according to the language definition.

`s.isnumeric()` Return True if there are only numeric characters in S, False otherwise.

`s.isspace()` Return True if all characters in S are whitespace and there is at least one character in S, False otherwise.

`s.isupper()` Return True if all cased characters in S are uppercase and there is at least one cased character in S, False otherwise.

`s.isalpha()` Return True if all characters in S are alphabetic and there is at least one character in S, False otherwise.

`s.isdigit()` Return True if all characters in S are digits and there is at least one character in S, False otherwise.

`s.islower()` Return True if all cased characters in S are lowercase and there is at least one cased character in S, False otherwise.

`s.isprintable()` Return True if all characters in S are considered printable in `repr()` or S is empty, False otherwise.

`s.istitle()` Return True if S is a titlecased string and there is at least one character in S, i.e. upper- and titlecase characters may only follow uncased characters and lowercase characters only cased ones.

2.4 Indexing and Slicing Strings

A character is a single letter, digit, space, newline, or other symbol. Think of a string as a list of characters. This is because a string can be indexed like a list in order to pick out a specific *character*. For example:

```
1 >>> s = "99 Python Problems"
2 >>> s[0]
3 '9'
4 >>> s[2]
5 ' '
6 >>> s[3]
7 'P'
8 >>> s[-1]
9 's'
10 >>> s[100]
11 Traceback (most recent call last):
12   File "stdin", line 1, in <module>
13     s[100]
14 IndexError: string index out of range
15 Traceback (most recent call last):
16   File "stdin", line 1, in <module>
17     s[100]
18 IndexError: string index out of range
```

Slicing works on strings. Review lab 4 for a description of several types of slices.

```
1 >>> s = "The cat in the hat"
2 >>> print(s[2])
3 e
4 >>> print(s[4:7])
5 cat
6 >>> print(s[15:18])
7 hat
8 >>> print(s[14:18])
9 hat
10 >>> print(s[17:14:-1])
```

```

11 tah
12 >>> print(s[17::-1])
13 tah eht ni tac ehT

```

2.5 Creating New Strings

Unlike lists, strings are *immutable*. A string's components can be accessed but can't be modified.

```

1 >>> s = 'Python'
2 >>> s[0] = 'J'
3 Traceback (most recent call last):
4   File "<stdin>", line 1, in <module>
5   TypeError: 'str' object does not support item assignment
6 Traceback (most recent call last):
7   File "<stdin>", line 1, in <module>
8   TypeError: 'str' object does not support item assignment

```

In this case, Python raises a **TypeError**, just like when attempting to modify tuples. If you want to change a string, you must create a new string:

```

1 >>> s = 'Python'
2 >>> r = 'J' + s[1:]
3 >>> print(r)
4 Jython

```

2.5.1 Concatenation and Repetition Using +, *

You can concatenate, or join, two strings using the + operator; this operation creates a new string.

```

1 >>> r = "The cat"
2 >>> s = " in the hat"
3 >>> r + s
4 'The cat in the hat'
5 >>> s + r
6 ' in the hatThe cat'

```

In addition to that, we can repeat strings using the multiplication operator *:

```

1 >>> s = "Hi"
2 >>> r = 5 * s
3 >>> print(r)
4 HiHiHiHiHi
5 >>> n = 3
6 >>> print(r + 'cat' * n)
7 HiHiHiHiHicatcatcat

```


2.5.2 Replacing Substrings With `string.replace(old, new)`

The `string.replace(old, new)` replaces the substring `old` with the string `new`. For example:

```
1 >>> s = 'abcDE'
2 >>> r = s.replace('ab', 'more')
3 >>> print(r)
4 morecDE
5 >>> print(s)
6 abcDE
```

2.5.3 Creating Uppercase and Lowercase Strings

The functions `s.upper()`, `s.lower()`, and `s.capitalize()` create new strings that are entirely uppercase or lowercase, or capitalized.

```
1 >>> s = 'abcDE'
2 >>> print(s.upper())
3 ABCDE
4 >>> s.lower()
5 'abcde'
6 >>> s.capitalize()
7 'Abcde'
```

2.6 Splitting a String Into a List of Strings Using `str.split()`

How can a string of space separated words be transformed into a list of words? For example, the string `"The cat in the hat"` might be more useful as a list of strings: `["The", "cat", "in", "the", "hat"]`. Python's `str.split(delimiter)` function takes a string and returns a list of all substrings that were separated by a given delimiter.

To split words separated by spaces, the delimiter would be the string `" "`. Here are some examples:

```
1 >>> "The cat in the hat".split(" ")
2 ['The', 'cat', 'in', 'the', 'hat']
3 >>> "1,2,3,4,5,6,7".split(",")
4 ['1', '2', '3', '4', '5', '6', '7']
5 >>> "1, 2, 3, 4, 5, 6,7".split(", ") # not magic
6 ['1', '2', '3', '4', '5', ' 6,7']
7 >>> "none_here".split(" ")
8 ['none_here']
```

To split a string on any kind of whitespace, such as spaces or newlines, call the function without any arguments: `str.split()`.

```
1 >>> "The cat in\nthe hat".split(" ")
2 ['The', 'cat', 'in\nthe', ' ', ' ', ' ', ' ', 'hat']
```

```
3 >>> "The cat in\nthe hat".split()
4 ['The', 'cat', 'in', 'the', 'hat']
```

3 Documenting Your Functions and Modules With Docstrings

The `help()` function introduced in the previous labs prints the given function's documentation which is known as a docstring. You can write detailed documentation for your functions by placing a triple-quoted string immediately under the beginning of a function definition (after the `def f(...):`).

The docstring is a comment describing the use of a function: the documentation should include information about each argument, the return value, and the function's behavior.

Function comments must follow the Python Enhancement Proposal 257 (PEP 257), a document that describes Python docstring conventions. It is found at:

<https://www.python.org/dev/peps/pep-0257/>

The highlights of PEP 257 are listed below.

For short functions, the docstring may be a one-liner:

```
1 def midpoint(a, b):
2     """Find and return the midpoint of the given a and b."""
3     return (a+b)/2
```

For larger functions or for a longer explanation, you must write a short summary followed by as much text as needed:

```
1 def calculate_weekly_pay(pay_rate, hours, tax_rate):
2     """Find net weekly pay after taxes with overtime set to 1.5 the pay rate.
3
4     The net pay after taxes is calculated given the number of hours worked in a
5     week, a pay rate, and a flat tax rate. Takes into consideration overtime pay
6     at 1.5 times the pay rate.
7
8     Arguments:
9     pay_rate -- rate of pay
10    hours -- number of hours worked in one week
11    tax_rate -- flat tax rate (for example, 0.15 for 15%)
12    """
13    pay_before_taxes = hours * pay_rate
14
15    # Add overtime payment if necessary
16    if hours > 40:
17        pay_before_taxes += (hours - 40) * pay_rate * 0.5
18
19    pay_after_taxes = pay_before_taxes * (1 - tax_rate)
20    return pay_after_taxes
```

Docstrings are Required

From now on, you will be required to put a *docstring* at the beginning of every function that you write.

4 Optional Function Arguments: Keyword Arguments

Functions may take zero or more required arguments, these are called *positional arguments*. There is a special syntax that can be used to define *optional arguments*. For example, the following function takes either one or two arguments. The second argument has a default value of 5.

```
1 >>> def f(fst, snd = 5):
2 ...     return fst * snd
3 >>> f(10)
4 50
5 >>> f(10, 6)
6 60
7 >>> f()
8 Traceback (most recent call last):
9   File "<stdin>", line 1, in <module>
10 TypeError: f() missing 1 required positional argument: 'fst'
11 Traceback (most recent call last):
12   File "<stdin>", line 1, in <module>
13 TypeError: f() missing 1 required positional argument: 'fst'
```

An important feature is that keyword arguments may be labeled and listed in any order when calling the function. For example, the following function takes two keyword arguments. There are five valid ways to call this function. You may pass no arguments, then the default values are used. You may pass either argument by itself or you may pass both arguments in any order.

```
1 >>> def p(greeting='Hello', subject='world'):
2 ...     return '{} {}'.format(greeting, subject)
3 >>> p()
4 'Hello world'
5 >>> p(greeting='Goodbye')
6 'Goodbye world'
7 >>> p(subject='humans')
8 'Hello humans'
9 >>> p(greeting='Bonjour', subject='le monde')
10 'Bonjour le monde'
11 >>> p(subject='le monde', greeting='Bonjour')
12 'Bonjour le monde'
```

5 Exercises

Exercise 5.1 (stringfun.py).

This exercise is a modification of exercises in the Google Python class, licensed under the Apache License 2.0.

The following functions should not print anything. Create a user interface that uses the functions, the `input()` function, and the `print()` function.

`ends()` Write a function that takes in a string from the user and prints just the first two and the last two characters of the string. You may assume that any input will be at least 2 characters long.

```
1 == ends ==
2 Enter a string >>> spring
3 spng
```

`mix()` Write a function that takes two strings a and b and prints the two strings concatenated, but with the first two characters of each word swapped with the other word's first two characters. You may assume that any input will be at least two characters long. For example:

```
1 == mix ==
2 String a >>> german
3 String b >>> english
4 enrman geglish
```

```
1 == mix ==
2 String a >>> dog
3 String b >>> dinner
4 dig donner
```

Exercise 5.2 (luhns.py).

Luhn's algorithm provides a quick way to check if a credit card is valid or not. The algorithm consists of four steps.

1. We will use the Diners Club card number 38520000023237 as an example.

3 8 5 2 0 0 0 0 0 2 3 2 3 7

2. Starting with the second to last digit (ten's column digit), multiply every other digit by two.

6 8 10 2 0 0 0 0 0 2 6 2 6 7

- Sum all the digits of the resulting number. Note that for 10, you also sum its digits: $(1 + 0)$; for an 18, you would do $1 + 8$.

$$6 + 8 + (1 + 0) + 2 + 0 + 0 + 0 + 0 + 0 + 2 + 6 + 2 + 6 + 7 = 40$$

- If the total sum modulo by 10 is zero, then the card is valid; otherwise it is invalid. For this example, the last step is to check if 40 modulus 10 is equal to 0, which is true. So the card is valid.

Write a program that implements Luhn's Algorithm for validating credit cards. It should ask the user to enter a credit card number and tell the user whether it is valid or not.

There must be a separate function called `validate` that takes in a card number and validates it. It should not print anything nor accept user input.

Testing Script

You can test your code in `luhns.py` by running the test script `test_luhns.py`. This will automatically test your function on various card numbers. The output of the test script may look like:

```
1 $ ls
2 luhns.py test_luhns.py ...
3 $ python3 test_luhns.py
4 luhns.validate("49927398717") should return False, but returned True.
5 luhns.validate("1234567812345678") should return False, but returned True.
6 The function luhns.validate correctly verified 3 out of 5 card numbers.
```

Exercise 5.3 (`fractions.py`).

Any fraction can be written as the division of two integers. You could express this in Python as a tuple — (numerator, denominator).

For example, the fractions $\frac{1}{2}$, $\frac{10}{7}$, and $\frac{499}{10001}$ can be represented using the tuples `(1, 2)`, `(10, 7)`, and `(499, 10001)`.

Write the following functions:

`reduce(fraction)` This function takes a fraction, reduces it, and returns the result. For example, `reduce((8, 4))` should return `(2, 1)`. To reduce a fraction a/b , divide a and b by their GCD. The result is $(a/d)/(b/d)$. The `math` module comes with the `math.gcd` function.

`add(fraction1, fraction2)` Given two fractions as tuples, add them.

`multiply(fraction1, fraction2)` Given two fractions as tuples, multiply them.

`divide(fraction1, fraction2)` Given two fractions as tuples, divide them.

These functions should not use `input()` or `print()`.

Write a small command-line interface such that the user running your script sees something like this:

```
1 $ python3 fractions.py
2 Enter a fraction >>> 5/3
3 Enter a fraction >>> 10/3
4 Reduced fractions to 5/3 and 10/3.
5 Sum of the fractions: 3/1.
6 Multiplication of the fractions: 50/9.
7 Division of the first by the second: 1/2 .
8 $ python3 fractions.py
9 Enter a fraction >>> 3628800/479001600
10 Enter a fraction >>> 10/1000
11 Reduced fractions to 1/132 and 1/100
12 Sum of the fractions: 29/1650
13 Multiplication of the fractions: 1/13200
14 Division of the first by the second: 25/33
```

The `split` function may be useful when converting strings “xxx/yyy” into tuples (xxx, yyy). You should use the functions `add`, `multiply`, `reduce`, and `divide` from your main function.

Index of New Functions and Methods

<code>*</code> , 6	<code>s.capitalize()</code> , 7	<code>s.upper()</code> , 7
<code>+</code> , 6	<code>s.isalnum()</code> , 4	sequence data types, 3
escaping characters, 2	<code>s.isalpha()</code> , 4	slicing strings, 5
immutable, 1, 6	<code>s.isdecimal()</code> , 4	slicing tuples, 1
<code>in</code> , 3	<code>s.isdigit()</code> , 5	<code>str.split(delimiter)</code> , 7
<code>len()</code> , 1, 4	<code>s.isidentifier()</code> , 4	<code>string.replace(old,</code> <code>new)</code> , 7
multi-line strings, 3	<code>s.islower()</code> , 5	strings, 2
optional arguments, 9	<code>s.isnumeric()</code> , 4	tuples, 1
positional arguments, 9	<code>s.isprintable()</code> , 5	<code>TypeError</code> , 1, 6
	<code>s.isspace()</code> , 4	unpack tuples, 1
	<code>s.istitle()</code> , 5	
	<code>s.isupper()</code> , 4	
	<code>s.lower()</code> , 7	

Submitting

You should submit your code as a tarball that contains all the exercise files for this lab. The submitted file should be named

`cse107_firstname_lastname_lab5.tar.gz`

Upload your tarball to Canvas.

List of Files to Submit

5.1	Exercise (stringfun.py)	10
5.2	Exercise (luhns.py)	10
5.3	Exercise (fractions.py)	11

Exercises start on page 10.