PYTHON PROGRAMMING COOKBOOK

Hot Recipes for Python Development

SEBASTIAN VINCI

WEB CODE GEEKS
WEB DEVELOPERS RESOURCE CENTER

Python Programming Cookbook

Contents

1 CSV Reader / Writer Example **1**

1.1 The Basics . . . 1

1.2 Reading and writing dictionaries . . . 5

1.3 Download the Code Project . . . 6

2 Decorator Tutorial **7**

2.1 Understanding Functions . . . 7

2.2 Jumping into decorators . . . 8

2.3 The Practice . . . 9

2.4 Download the Code Project . . . 14

3 Threading / Concurrency Example **15**

3.1 Python _thread module . . . 15

3.2 Python threading module . . . 16

3.2.1 Extending Thread . . . 16

3.2.2 Getting Current Thread Information . . . 17

3.2.3 Daemon Threads . . . 18

3.2.4 Joining Threads . . . 20

3.2.5 Time Threads . . . 22

3.2.6 Events: Communication Between Threads . . . 23

3.2.7 Locking Resources . . . 24

3.2.8 Limiting Concurrent Access to Resources . . . 29

3.2.9 Thread-Specific Data . . . 29

4 Logging Example **31**

4.1 The Theory . . . 31

4.1.1 Log Levels . . . 31

4.1.2 Handlers . . . 32

4.1.3 Format . . . 32

4.2 The Practice . . . 33

4.3 Download the Code Project . . . 39

5 Django Tutorial **40**
5.1 Creating the project . . . 40
5.2 Creating our application . . . 40
5.3 Database Setup . . . 41
5.4 The Public Page . . . 45
5.5 Style Sheets . . . 47
5.6 Download the Code Project . . . 49

6 Dictionary Example **50**
6.1 Define . . . 50
6.2 Read . . . 50
6.3 Write . . . 52
6.4 Useful operations . . . 52
6.4.1 In (keyword) . . . 53
6.4.2 Len (built-in function) . . . 53
6.4.3 keys() and values() . . . 54
6.4.4 items() . . . 55
6.4.5 update() . . . 56
6.4.6 copy() . . . 56
6.5 Download the Code Project . . . 57

7 Sockets Example **58**
7.1 Creating a Socket . . . 58
7.2 Using a Socket . . . 59
7.3 Disconnecting . . . 60
7.4 A Little Example Here . . . 60
7.5 Non-blocking Sockets . . . 62
7.6 Download the Code Project . . . 63

8 Map Example **64**
8.1 Map Implementation . . . 64
8.2 Python's Map . . . 65
8.3 Map Object . . . 68
8.4 Download the Code Project . . . 69

9 Subprocess Example **70**
9.1 Convenience Functions . . . 70
9.1.1 subprocess.call . . . 70
9.1.2 subprocess.check_call . . . 71
9.1.3 subprocess.check_output . . . 72
9.2 Popen . . . 72
9.3 Download the Code Project . . . 74

10 Send Email Example **75**

10.1 The Basics of smtplib 75

10.2 SSL and Authentication 76

10.3 Sending HTML 77

10.4 Sending Attachments 78

10.5 Download the Code Project 80

Preface

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

Python supports multiple programming paradigms, including object-oriented, imperative and functional programming or procedural styles. It features a dynamic type system and automatic memory management and has a large and comprehensive standard library. (Source: https://en.wikipedia.org/wiki/Python_%28programming_language%29)

In this ebook, we provide a compilation of Python examples that will help you kick-start your own projects. We cover a wide range of topics, from multi-threaded programming to web development with Django. With our straightforward tutorials, you will be able to get your own projects up and running in minimum time.

About the Author

Sebastian is a full stack programmer, who has strong experience in Java and Scala enterprise web applications.

He is currently studying Computers Science in UBA (University of Buenos Aires) and working a full time job at a .com company as a Semi-Senior developer, involving architectural design, implementation and monitoring.

He also worked in automating processes (such as data base backups, building, deploying and monitoring applications).

Chapter 1

CSV Reader / Writer Example

CSV (comma-separated values) is a standard for exporting and importing data across multiple formats, such as MySQL and excel.

It stores numbers and text in plain text. Each row of the file is a data record and every record consists of one or more fields which values are separated by commas. The use of the comma to separate every record's fields is the source of the name given to this standard.

Even with this **very** explicit name, there is no official standard CSVs, and it may denote some very similar delimiter-separated values, which use a variety of field delimiters (such as spaces and tabs, which are both very popular), and are given the .csv extension anyway. Such lack of strict terminology makes data exchange very difficult some times.

RFC 4180 provides some rules to this format:

- It's plain text
- Consists of records
- Every record consists of fields separated by a single character delimiter
- Every record has the same sequence of fields

But unless there is additional information about the provided file (such as if the rules provided by RFC were followed), data exchange through this format can be pretty annoying.

1.1 The Basics

Python has native support for CSV readers, and it's configurable (which, as we've seen, is necessary). There is a module csv which holds everything you need to make a CSV reader/writer, and it follows RFC standards (unless your configuration overrides them), so by default it should read and write valid CSV files. So, let's see how it works:

csv-reader.py

```
import csv
with open('my.csv', 'r+', newline='') as csv_file:
    reader = csv.reader(csv_file)
    for row in reader:
        print(str(row))
```

Here, we are importing csv and opening a file called *my.csv*, then we call csv.reader passing our file as a parameter and then we print each row in our reader.

If *my.csv* looks like this:

my.csv

```
my first column,my second column,my third column
my first column 2,my second column 2,my third column 2
```

Then, when you run this script, you will see the following output:

```
['my first column', 'my second column', 'my third column']
['my first column 2', 'my second column 2', 'my third column 2']
```

And writing is just as simple as reading:

csv-reader.py

```
import csv
rows = [['1', '2', '3'], ['4', '5', '6']]
with open('my.csv', 'w+', newline='') as csv_file:
    writer = csv.writer(csv_file)
    for row in rows:
        writer.writerow(row)

with open('my.csv', 'r+', newline='') as csv_file:
    reader = csv.reader(csv_file)
    for row in reader:
        print(str(row))
```

Then, in your csv file you'll see:

my.csv

```
1,2,3
4,5,6
```

And in your output:

```
['1', '2', '3']
['4', '5', '6']
```

It's pretty easy to see what is going on in here. We are opening a file in write mode, getting our writer from csv giving our file to it, and writing each row with it. Making it a little smarter:

csv-reader.py

```
import csv


def read(file_location):
    with open(file_location, 'r+', newline='') as csv_file:
        reader = csv.reader(csv_file)
        return [row for row in reader]


def write(file_location, rows):
    with open(file_location, 'w+', newline='') as csv_file:
        writer = csv.writer(csv_file)
        for row in rows:
            writer.writerow(row)


def raw_test():
    columns = int(input("How many columns do you want to write? "))
    input_rows = []
    keep_going = True
    while keep_going:
```

```
        input_rows.append([input("column {}: ".format(i + 1)) for i in range(0, columns)])
        ui_keep_going = input("continue? (y/N): ")
        if ui_keep_going != "y":
            keep_going = False

    print(str(input_rows))

    write('raw.csv', input_rows)
    written_value = read('raw.csv')
    print(str(written_value))

raw_test()
```

We ask the user how many columns does he want to write for each row and then ask him for a row as long as he wants to continue, then we print our raw input and write it to a file called raw.csv, then we read it again and print the data. When we run our script, the output will look like this:

```
How many columns do you want to write? 3
column 1: row 1, column 1
column 2: row 1, column 2
column 3: row 1, column 3
continue? (y/N): y
column 1: row 2, column 1
column 2: row 2, column 2
column 3: row 3, column 3
continue? (y/N):
[['row 1, column 1', 'row 1, column 2', 'row 1, column 3'], ['row 2, column 1', 'row 2,  ↩
    column 2', 'row 3, column 3']]
[['row 1, column 1', 'row 1, column 2', 'row 1, column 3'], ['row 2, column 1', 'row 2,  ↩
    column 2', 'row 3, column 3']]

Process finished with exit code 0
```

And, of course, our raw.csv looks like this:

raw.csv

```
"row 1, column 1","row 1, column 2","row 1, column 3"
"row 2, column 1","row 2, column 2","row 3, column 3"
```

Another rule the CSV format has, is the quote character. As you see, every input has a comma, which is our separator character, so the writer puts them between quoting marks (the default of the standard) to know that commas between them are not separators, but part of the column instead.

Now, although I would recommend leaving the configuration with its defaults, there are some cases where you need to change them, as you don't always have control over the csv's your data providers give you. So, I have to teach you how to do it (beware, great powers come with great responsibilities).

You can configure the delimiter and the quote character through `delimiter` and `quotechar` parameters, like this:

csv-reader.py

```
import csv


def read(file_location):
    with open(file_location, 'r+', newline='') as csv_file:
        reader = csv.reader(csv_file, delimiter=' ', quotechar='|')
        return [row for row in reader]


def write(file_location, rows):
    with open(file_location, 'w+', newline='') as csv_file:
```

```
        writer = csv.writer(csv_file, delimiter=' ', quotechar='|')
        for row in rows:
            writer.writerow(row)


def raw_test():
    columns = int(input("How many columns do you want to write? "))
    input_rows = []
    keep_going = True
    while keep_going:
        input_rows.append([input("column {}: ".format(i + 1)) for i in range(0, columns)])
        ui_keep_going = input("continue? (y/N): ")
        if ui_keep_going != "y":
            keep_going = False

    print(str(input_rows))

    write('raw.csv', input_rows)
    written_value = read('raw.csv')
    print(str(written_value))

raw_test()
```

So, now, having this console output:

```
How many columns do you want to write? 3
column 1: row 1 column 1
column 2: row 1 column 2
column 3: row 1 column 3
continue? (y/N): y
column 1: row 2 column 1
column 2: row 2 column 2
column 3: row 2 column 3
continue? (y/N):
[['row 1 column 1', 'row 1 column 2', 'row 1 column 3'], ['row 2 column 1', 'row 2 column 2 ↩
    ', 'row 2 column 3']]
[['row 1 column 1', 'row 1 column 2', 'row 1 column 3'], ['row 2 column 1', 'row 2 column 2 ↩
    ', 'row 2 column 3']]
```

Our raw.csv will like like this:

raw.csv

```
|row 1 column 1| |row 1 column 2| |row 1 column 3|
|row 2 column 1| |row 2 column 2| |row 2 column 3|
```

As you see, our new separator is the space character, and our quote character is pipe, which our writer is forced to use always as the space character is pretty common in almost every text data.

The writer's quoting strategy is also configurable, the values available are:

- `csv.QUOTE_ALL`: quotes every column, it doesn't matter if they contain a delimiter character or not.
- `csv.QUOTE_MINIMAL`: quotes only the columns which contains a delimiter character.
- `csv.QUOTE_NONNUMERIC`: quotes all non numeric columns.
- `csv.QUOTE_NONE`: quotes nothing. It forces you to check whether or not the user inputs a delimiter character in a column, if you don't, you will read an unexpected number of columns.

1.2 Reading and writing dictionaries

We've seen a very basic example of how to read and write data from a CSV file, but in real life, we don't want our CSV's to be so chaotic, we need them to give us information about what meaning has each of the columns.

Also, en real life we don't usually have our data in arrays, we have business models and we need them to be very descriptive. We usually use dictionaries for this purpose, and python gives us the tools to write and read dictionaries from CSV files.

It looks like this:

```
import csv

dictionaries = [{'age': '30', 'name': 'John', 'last_name': 'Doe'}, {'age': '30', 'name': ' ↩
    Jane', 'last_name': 'Doe'}]
with open('my.csv', 'w+') as csv_file:
    headers = [k for k in dictionaries[0]]
    writer = csv.DictWriter(csv_file, fieldnames=headers)
    writer.writeheader()
    for dictionary in dictionaries:
        writer.writerow(dictionary)

with open('my.csv', 'r+') as csv_file:
    reader = csv.DictReader(csv_file)
    print(str([row for row in reader]))
```

We are initializing a variable called `dictionaries` with an array of test data, then we open a file in write mode, we collect the keys of our dictionary and get a writer of our file with the headers. The first thing we do is write our headers, and then write a row for every dictionary in our array.

Then we open the same file in read mode, get a reader of that file and print the array of data. You will see an output like:

```
[{'name': 'John', 'age': '30', 'last_name': 'Doe'}, {'name': 'Jane', 'age': '30', ' ↩
    last_name': 'Doe'}]
```

And our csv file will look like:

my.csv

```
name,last_name,age
John,Doe,30
Jane,Doe,30
```

Now it looks better. The CSV file has our headers information, and each row has the ordered sequence of our data. Notice that we give the file names to our writer, as dictionaries in python are not ordered so the writer needs that information to write each row with the same order.

The same parameters apply for the delimiter and the quote character as the default reader and writer.

So, then again, we make it a little smarter:

csv-reader.py

```
import csv

def read_dict(file_location):
    with open(file_location, 'r+') as csv_file:
        reader = csv.DictReader(csv_file)
        print(str([row for row in reader]))
        return [row for row in reader]


def write_dict(file_location, dictionaries):
    with open(file_location, 'w+') as csv_file:
```

```
        headers = [k for k in dictionaries[0]]
        writer = csv.DictWriter(csv_file, fieldnames=headers)
        writer.writeheader()
        for dictionary in dictionaries:
            writer.writerow(dictionary)


def dict_test():
    input_rows = []
    keep_going = True
    while keep_going:
        name = input("Name: ")
        last_name = input("Last Name: ")
        age = input("Age: ")
        input_rows.append({"name": name, "last_name": last_name, "age": age})
        ui_keep_going = input("continue? (y/N): ")
        if ui_keep_going != "y":
            keep_going = False

    print(str(input_rows))

    write_dict('dict.csv', input_rows)
    written_value = read_dict('dict.csv')
    print(str(written_value))

dict_test()
```

And now when we run this script, we'll see the output:

```
Name: John
Last Name: Doe
Age: 30
continue? (y/N): y
Name: Jane
Last Name: Doe
Age: 40
continue? (y/N):
[{'age': '30', 'last_name': 'Doe', 'name': 'John'}, {'age': '40', 'last_name': 'Doe', 'name ↩
    ': 'Jane'}]
[{'age': '30', 'last_name': 'Doe', 'name': 'John'}, {'age': '40', 'last_name': 'Doe', 'name ↩
    ': 'Jane'}]
```

And our dict.csv will look like:

csv-reader.py

```
age,last_name,name
30,Doe,John
40,Doe,Jane
```

A little side note: As I said before, dictionaries in python are not ordered, so when you extract the keys from one to write its data to a CSV file you should order them to have your columns ordered always the same way, as you do not know which technology will your client use to read them, and, of course, in real life CSV files are incremental, so you are always adding lines to them, not overriding them. Avoid any trouble making sure your CSV will always look the same.

1.3 Download the Code Project

This was an example on how to read and write data from/to a CSV file.

Download You can download the full source code of this example here: **python-csv-reader**

Chapter 2

Decorator Tutorial

Sometimes, we encounter problems that require us to extend the behavior of a function, but we don't want to change its implementation.Some of those problems could be: logging spent time, caching, validating parameters, etc. All these solutions are often needed in more than one function: you often need to log the spent time of every http connection; you often need to cache more than one data base entity; you often need validation in more than one function. Today we will solve these 3 mentioned problems with Python decorators:

- Spent Time Logging: We'll decorate a couple functions to tell us how much time do they take to execute.
- Caching: We'll add cache to prevent a function from executing when its called several times with the same parameters.
- Validating: We'll validate a function's input to prevent run time errors.

2.1 Understanding Functions

Before we can jump into decorators, we need to understand how functions actually work. In essence, functions are procedures that return a value based on some given arguments.

```
def my_function(my_arg):
    return my_arg + 1
```

In Python, functions are first-class objects. This means that functions can be assigned to a variable:

```
def my_function(foo):
    return foo + 1
my_var = my_function
print(str(my_var(1))) # prints "2"
```

They can be defined inside another functions:

```
def my_function(foo):
    def my_inner_function():
        return 1
    return foo + my_inner_function()
print(str(my_function(1))) # still prints "2"
```

They can be passed as parameters (higher-order functions):

```
def my_function(foo, my_parameter_function):
    return foo + my_parameter_function()
def parameter_function(): return 1
print(str(my_function(1, parameter_function))) # still prints "2"
```

And they can return other functions (also, higher-order functions):

```
def my_function(constant):
    def inner(foo):
        return foo + constant
    return inner
plus_one = my_function(1)
print(str(plus_one(1))) # still prints "2"
```

Another thing to notice, is that inner functions have access to the outer scope, that's why we can use the parameter `constant` in the inner function of the last example. Also, this access is read-only, we can not modify variables from the outer scope within an inner function.

2.2 Jumping into decorators

Python decorators provide a nice and simple syntax to call higher-order functions. By definition, a decorator takes a function as a parameter, and returns a wrapper of that given function to extend its behavior without actually modifying it. Given this definition we can write somthing like:

```
def decorator(function_to_decorate):
    def wrapper(value):
        print("you are calling {} with '{}' as parameter".format(function_to_decorate. ↩
            __name__, value))
        return function_to_decorate(value)
    return wrapper


def replace_commas_with_spaces(value):
    return value.replace(",", " ")

function_to_use = decorator(replace_commas_with_spaces)
print(function_to_use("my,commas,will,be,replaces,with,spaces"))
```

And after execution, the output will look like:

```
you are calling replace_commas_with_spaces with 'my,commas,will,be,replaces,with,spaces' as ↩
     parameter
my commas will be replaces with spaces
```

So, what is actually happening here? We are defining a higher-order function called `decorator` that receives a function as a parameter and returns a wrapper of that function. The wrapper just prints to the console the name of the called function and the given parameters before executing the wrapped function. And the wrapped functions just replaces the commas with spaces. Now we have a decorator written here. But it's kind of annoying to define the decorator, the function and then assigning the wrapper to another variable to finally be able to use it. Python provides some sugar syntax to make it easier to write and read, and if we re-write this decorator using it:

```
def decorator(function_to_decorate):
    def wrapper(value):
        print("you are calling {} with '{}' as parameter".format(function_to_decorate. ↩
            __name__, value))
        return function_to_decorate(value)
    return wrapper


@decorator
def replace_commas_with_spaces(value):
    return value.replace(",", " ")

print(replace_commas_with_spaces.__name__)
```

```
print(replace_commas_with_spaces.__module__)
print(replace_commas_with_spaces.__doc__)
print(replace_commas_with_spaces("my,commas,will,be,replaces,with,spaces"))
```

We just annotate the function we want to wrap with the decorator function and that's it. That function will be decorated, and the output will look the same.

```
wrapper
__main__
None
you are calling replace_commas_with_spaces with 'my,commas,will,be,replaces,with,spaces' as ↩
     parameter
my commas will be replaces with spaces
```

Now, debugging this can be a real pain, as the `replace_commas_with_spaces` function is overridden with the wrapper, so its `__name__`, `__doc__` and `__module__` will also be overridden (as seen in the output). To avoid this behavior we will use `functools.wraps`, that prevents a wrapper from overriding its inner function properties.

```
from functools import wraps


def decorator(function_to_decorate):
    @wraps(function_to_decorate)
    def wrapper(value):
        print("you are calling {} with '{}' as parameter".format(function_to_decorate. ↩
            __name__, value))
        return function_to_decorate(value)
    return wrapper


@decorator
def replace_commas_with_spaces(value):
    return value.replace(",", " ")

print(replace_commas_with_spaces.__name__)
print(replace_commas_with_spaces.__module__)
print(replace_commas_with_spaces.__doc__)
print(replace_commas_with_spaces("my,commas,will,be,replaces,with,spaces"))
```

And now the output will be:

```
replace_commas_with_spaces
__main__
None
you are calling replace_commas_with_spaces with 'my,commas,will,be,replaces,with,spaces' as ↩
     parameter
my commas will be replaces with spaces
```

So, now we know how decorators work in python. Let's solve our mentioned problems.

2.3 The Practice

So, we need to implement cache, spent time logging and validations. Let's combine them all by solving a bigger problem: **palindromes**. Let's make an algorithm that, given a word, will check if it's a palindrome. If it isn't, it will convert it to palindrome. palindrome.py

```
def is_palindrome(string_value):
    char_array = list(string_value)
    size = len(char_array)
```

```
    half_size = int(size / 2)
    for i in range(0, half_size):
        if char_array[i] != char_array[size - i - 1]:
            return False
    return True


def convert_to_palindrome(v):
    def action(string_value, chars):
        chars_to_append = list(string_value)[0:chars]
        chars_to_append.reverse()
        new_value = string_value + "".join(chars_to_append)
        if not is_palindrome(new_value):
            new_value = action(string_value, chars + 1)
        return new_value
    return action(v, 0)

user_input = input("string to convert to palindrome (exit to terminate program): ")
while user_input != "exit":
    print(str(convert_to_palindrome(user_input)))
    user_input = input("string to check (exit to terminate program): ")
```

Here, we have a function called `is_palindrome`, which given an input, returns `True` if its palindrome, or `False` otherwise. Then, there is a function called `convert_to_palindrome` which, given an input, will add just as many characters (reversed, from the beginning) as necessary to make it palindrome. Also, there is a while that reads the user input until he inputs "exit". The output looks like:

```
string to convert to palindrome (exit to terminate program): anita lava la tina
anita lava la tinanit al aval atina
string to check (exit to terminate program): anitalavalatina
anitalavalatina
string to check (exit to terminate program): menem
menem
string to check (exit to terminate program): mene
menem
string to check (exit to terminate program): casa
casac
string to check (exit to terminate program): casaca
casacasac
string to check (exit to terminate program): exit
```

As you can see, it works just fine. But we have a couple problems:

- I don't know how long does it take it to process the input, or if its related to the length of it. (spent time logging)
- I don't want it to process twice the same input, it's not necessary. (cache)
- It's designed to work with words or numbers, so I don't want spaces around. (validation)

Let's get dirty here, and start with a spent time logging decorator.

palindrome.py

```
import datetime
from functools import wraps


def spent_time_logging_decorator(function):
    @wraps(function)
    def wrapper(*args):
        start = datetime.datetime.now()
        result = function(*args)
```

```
        end = datetime.datetime.now()
        spent_time = end - start
        print("spent {} microseconds in {} with arguments {}. Result was: {}".format( ↩
            spent_time.microseconds, function.__name__, str(args), result))
        return result

    return wrapper


def is_palindrome(string_value):
    char_array = list(string_value)
    size = len(char_array)
    half_size = int(size / 2)
    for i in range(0, half_size):
        if char_array[i] != char_array[size - i - 1]:
            return False
    return True


@spent_time_logging_decorator
def convert_to_palindrome(v):
    def action(string_value, chars):
        chars_to_append = list(string_value)[0:chars]
        chars_to_append.reverse()
        new_value = string_value + "".join(chars_to_append)
        if not is_palindrome(new_value):
            new_value = action(string_value, chars + 1)
        return new_value

    return action(v, 0)


user_input = input("string to convert to palindrome (exit to terminate program): ")
while user_input != "exit":
    print(str(convert_to_palindrome(user_input)))
    user_input = input("string to check (exit to terminate program): ")
```

It's pretty simple, we wrote a decorator which returns a wrapper that gets the time before and after executor, and then calculates the spent time and logs it, with the called function, parameters and result. The output looks like:

```
string to check (exit to terminate program): anitalavalatina
spent 99 microseconds in convert_to_palindrome with arguments ('anitalavalatina',). Result  ↩
    was: anitalavalatina
anitalavalatina
string to check (exit to terminate program): exit
```

As you see, there is plenty information in that log line, and our implementation was not changed at all. Now, let's add cache: palindrome.py

```
import datetime
from functools import wraps


def cache_decorator(function):
    cache = {}

    @wraps(function)
    def wrapper(*args):
        hashed_arguments = hash(str(args))
        if hashed_arguments not in cache:
            print("result for args {} was not found in cache...".format(str(args)))
            cache[hashed_arguments] = function(*args)
```

```
        return cache[hashed_arguments]
    return wrapper


def spent_time_logging_decorator(function):
    @wraps(function)
    def wrapper(*args):
        start = datetime.datetime.now()
        result = function(*args)
        end = datetime.datetime.now()
        spent_time = end - start
        print("spent {} microseconds in {} with arguments {}. Result was: {}".format( ↩
            spent_time.microseconds, function.__name__, str(args), result))
        return result

    return wrapper


def is_palindrome(string_value):
    char_array = list(string_value)
    size = len(char_array)
    half_size = int(size / 2)
    for i in range(0, half_size):
        if char_array[i] != char_array[size - i - 1]:
            return False
    return True


@spent_time_logging_decorator
@cache_decorator
def convert_to_palindrome(v):
    def action(string_value, chars):
        chars_to_append = list(string_value)[0:chars]
        chars_to_append.reverse()
        new_value = string_value + "".join(chars_to_append)
        if not is_palindrome(new_value):
            new_value = action(string_value, chars + 1)
        return new_value

    return action(v, 0)


user_input = input("string to convert to palindrome (exit to terminate program): ")
while user_input != "exit":
    print(str(convert_to_palindrome(user_input)))
    user_input = input("string to check (exit to terminate program): ")
```

This is a very simple implementation of cache. No TTL, no thread safety. It's just a dictionary which keys are the hash of the arguments. If no value with the given key was found, it creates it, then retrieves it. Output:

```
string to convert to palindrome (exit to terminate program): anitalavalatina
result for args ('anitalavalatina',) was not found in cache...
spent 313 microseconds in convert_to_palindrome with arguments ('anitalavalatina',). Result ↩
     was: anitalavalatina
anitalavalatina
string to check (exit to terminate program): anitalavalatina
spent 99 microseconds in convert_to_palindrome with arguments ('anitalavalatina',). Result  ↩
    was: anitalavalatina
anitalavalatina
string to check (exit to terminate program): exit
```

There it is. The first execution with "anitalavalatina" outputs a line informing us that the result for that input was not found. But when we input it again, that line is gone. Awesome! But we still receive spaces, let's validate that: palindrome.py

```
import datetime
from functools import wraps


def validation_decorator(validator, if_invalid=None):
    def decorator(function):
        @wraps(function)
        def wrapper(*args):
            if validator(*args):
                return function(*args)
            else:
                return if_invalid
        return wrapper
    return decorator


def cache_decorator(function):
    cache = {}

    @wraps(function)
    def wrapper(*args):
        hashed_arguments = hash(str(args))
        if hashed_arguments not in cache:
            print("result for args {} was not found in cache...".format(str(args)))
            cache[hashed_arguments] = function(*args)
        return cache[hashed_arguments]
    return wrapper


def spent_time_logging_decorator(function):
    @wraps(function)
    def wrapper(*args):
        start = datetime.datetime.now()
        result = function(*args)
        end = datetime.datetime.now()
        spent_time = end - start
        print("spent {} microseconds in {} with arguments {}. Result was: {}".format( ↩
            spent_time.microseconds, function.__name__, str(args), result))
        return result

    return wrapper


def is_palindrome(string_value):
    char_array = list(string_value)
    size = len(char_array)
    half_size = int(size / 2)
    for i in range(0, half_size):
        if char_array[i] != char_array[size - i - 1]:
            return False
    return True


def should_not_contain_spaces(*args):
    return False not in map(lambda x: " " not in str(x), args)


@spent_time_logging_decorator
@validation_decorator(should_not_contain_spaces, "input shouldn't contain spaces.")
```

```
@cache_decorator
def convert_to_palindrome(v):
    def action(string_value, chars):
        chars_to_append = list(string_value)[0:chars]
        chars_to_append.reverse()
        new_value = string_value + "".join(chars_to_append)
        if not is_palindrome(new_value):
            new_value = action(string_value, chars + 1)
        return new_value

    return action(v, 0)


user_input = input("string to convert to palindrome (exit to terminate program): ")
while user_input != "exit":
    print(str(convert_to_palindrome(user_input)))
    user_input = input("string to check (exit to terminate program): ")
```

Now, this one is a little tricky. To pass arguments to the decorator we need to wrap it. Yeah, we need a wrapper of the wrapper. Thanks to that, we can pass the validation function and a message if input is invalid. The output looks like:

```
string to convert to palindrome (exit to terminate program): anita lava la tina
spent 87 microseconds in convert_to_palindrome with arguments ('anita lava la tina',).  ↩
    Result was: input shouldn't contain spaces.
input shouldn't contain spaces.
string to check (exit to terminate program): anitalavalatina
result for args ('anitalavalatina',) was not found in cache...
spent 265 microseconds in convert_to_palindrome with arguments ('anitalavalatina',). Result ↩
     was: anitalavalatina
anitalavalatina
string to check (exit to terminate program): exit
```

2.4 Download the Code Project

This was an example of how to write decorators in Python.

Download You can download the full source code of this example here: **python-decorator**

Chapter 3

Threading / Concurrency Example

Threads are processes which run in parallel to other threads. In a utopian scenario, if you split a big process in 2 threads, these threads will run in parallel so it would take half the time.

This is not true in most cases. Using CPython, there is a mutex that prevents multiple native threads from executing Python byte codes at once. It's called GIL (global interpreter lock). This lock is necessary mainly because CPython's memory management is not thread-safe, but notice that I/O, image processing, and other potentially blocking operations, happen outside the GIL, so it will only become a bottle neck in processes that spend a lot of time inside the GIL.

In most applications nowadays, concurrency is something we all must be able to handle. Mostly in web applications, where one request usually starts a thread, we need to have concurrency and threading in mind so we can write our programs accordingly.

Threading is also a good solution to optimize response times. Given a scenario in which we have to process 4 million objects, a good thing to do would be divide them in 4 groups of a million objects and process them in 4 separated threads.

3.1 Python _thread module

The `_thread` module is very effective for low level threading, let's see an example to understand the concept. But keep in mind that since Python 2.4, this module is not used anymore.

The process of spawning a thread is pretty simple. We just need to call a method called `start_new_thread`, available in the `_thread` module, which receives a function, and arguments to pass to it. It returns immediately and a thread will run in parallel. Let's see:

```
import _thread as thread
import time

executed_count = 0


# Define a function for the thread
def print_time(thread_name, delay):
    global executed_count
    count = 0
    while count < 5:
        time.sleep(delay)
        count += 1
        print("%s: %s" % (thread_name, time.ctime(time.time())))
    executed_count += 1


# Create two threads as follows
try:
    threads = [thread.start_new_thread(print_time, ("Thread-1", 2,)),
```

```
                thread.start_new_thread(print_time, ("Thread-2", 4,))]
except:
    print("Error: unable to start thread")

while executed_count < 2:
    pass
```

When we run this script we'll see an output that looks like:

```
Thread-1: Mon Dec 21 12:55:23 2015
Thread-2: Mon Dec 21 12:55:25 2015
Thread-1: Mon Dec 21 12:55:25 2015
Thread-1: Mon Dec 21 12:55:27 2015
Thread-2: Mon Dec 21 12:55:29 2015
Thread-1: Mon Dec 21 12:55:29 2015
Thread-1: Mon Dec 21 12:55:31 2015
Thread-2: Mon Dec 21 12:55:33 2015
Thread-2: Mon Dec 21 12:55:37 2015
Thread-2: Mon Dec 21 12:55:41 2015
```

So, let's see what is going on there.

Then we create two threads, each of them containing our `print_time` function, with a name and a delay assigned to them. And we have a while which makes sure the program won't exit until `executed_count` is equal or greater than 2.

3.2 Python threading module

The newer `threading` module included with Python 2.4 provides much more powerful, high-level support for threads than the `_thread` module.

3.2.1 Extending Thread

The most used procedure for spawning a thread using this module, is defining a subclass of the `Thread` class. Once you've done it, you should override the `__init__` and `run` methods.

Once you've got your class, you just instantiate an object of it and call the method called `start`. Let's see an example of this:

```
import threading

import time


class MyThread(threading.Thread):
    def __init__(self, name, sleep_time):
        threading.Thread.__init__(self)
        self.name = name
        self.sleep_time = sleep_time

    def run(self):
        print("{} start".format(self.name))
        time.sleep(self.sleep_time)
        print("{} end".format(self.name))


threads = [MyThread("Thread-{}".format(i), i) for i in range(1, 4)]
for t in threads:
    t.start()
```

Then we run it an see something like:

```
Thread-1 start
Thread-2 start
Thread-3 start
Thread-1 end
Thread-2 end
Thread-3 end
```

Of course, we don't need to name our threads like that. Each Thread instance has a name with a default value that can be changed as the thread is created. Naming threads is useful in server processes with multiple service threads handling different operations. So, let's change our code to avoid reinventing the wheel. In our constructor:

```
def __init__(self, sleep_time):
    threading.Thread.__init__(self)
    threading.Thread.__init__(self)
    self.sleep_time = sleep_time
```

Then the output will look like:

```
Thread-2 start
Thread-4 start
Thread-6 start
Thread-2 end
Thread-4 end
Thread-6 end
```

But it usually isn't so simple, the logic isn't in the run method ouf our `Thread` class, so let's make it a little more real and do the `print` and sleep in a method outside our `MyThread`.

3.2.2 Getting Current Thread Information

Our problem here is that we don't have our thread name outside of it, and we don't want that information going around in our function's signature. But the `threading` module, has some methods that give us access to the current thread information:

```
import threading
import time


def my_logic(sleep_time):
    thread_name = threading.current_thread().getName()
    print("{} start".format(thread_name))
    time.sleep(sleep_time)
    print("{} end".format(thread_name))


class MyThread(threading.Thread):
    def __init__(self, sleep_time):
        threading.Thread.__init__(self)
        threading.Thread.__init__(self)
        self.sleep_time = sleep_time

    def run(self):
        my_logic(self.sleep_time)


threads = [MyThread(i) for i in range(1, 4)]
for t in threads:
    t.start()
```

By executing `threading.current_thread()`, we gain access to the current thread information. Among that information we can find its status (`is_alive()`), its daemon flag (`isDaemon()`), and other useful methods.

3.2.3 Daemon Threads

Now, let's talk about daemon threads. Until now, our programs waited for every thread to end before actually terminating, but sometimes we don't want that behaviour. If we have a thread, pushing status or metrics to a service, we usually don't care if it has finished or not when we shut down our program, and maybe we don't want to explicitly terminate it before exiting.

Daemon threads run without blocking the main thread from exiting. They are useful when we have services where there may not be an easy way to interrupt the thread or where letting the thread die in the middle of its work does not lose or corrupt data.

To spawn a daemon thread, we just spawn a normal thread an call its `setDaemon()` method with `True` as a parameter. By default threads are not daemon. Let's se how our program behaves when we make one of those threads daemon:

```
threads = [MyThread(i) for i in range(1, 4)]
threads[2].setDaemon(True)
for t in threads:
    t.start()
```

We are now grabbing the last thread we create, and making it daemon. The output will now look like:

```
Thread-2 start
Thread-4 start
Thread-6 start
Thread-2 end
Thread-4 end
```

As you can see, the main thread is not waiting for Thread-6 to finish before exiting, daemon threads are terminated when the main thread finished its execution.

Let's write something that resembles a real-life problem solution. Let's make a script, that given an array of URL's, crawls them and saves the html in files.

site-crawler.py

```
import http.client
import threading
import logging

logging.basicConfig(level=logging.INFO, format='(%(threadName)-10s) %(message)s', )


def save(html, file_absolute_path):
    logging.info("saving {} bytes to {}".format(len(html), file_absolute_path))
    with open(file_absolute_path, 'wb+') as file:
        file.write(html)
        file.flush()


def crawl(req):
    logging.info("executing get request for parameters: {}".format(str(req)))
    connection = http.client.HTTPConnection(req["host"], req["port"])
    connection.request("GET", req["path"])
    response = connection.getresponse()
    logging.info("got {} response http code".format(response.status))
    logging.debug("headers: {}".format(str(response.headers)))
    response_content = response.read()
    logging.debug("actual response: {}".format(response_content))
    return response_content


class MyCrawler(threading.Thread):
    def __init__(self, req, file_path):
        threading.Thread.__init__(self, name="Crawler-{}".format(req["host"]))
        self.req = req
```

```
        self.file_path = file_path

    def run(self):
        global executed_crawlers
        html = crawl(self.req)
        save(html, self.file_path)


def __main__():
    continue_input = True
    threads = []
    while continue_input:
        host = input("host: ")
        port = 80  # int(input("port: "))
        path = "/"  # input("path: ")
        file_path = input("output file absolute path: ")
        req = {"host": host, "port": port, "path": path}
        threads.append(MyCrawler(req, file_path))
        continue_input = input("add another? (y/N) ") == "y"

    for t in threads:
        t.start()
        # t.join()

__main__()
```

So, what do we've got here? This is a script that asks the user to input a host and the absolute path of the file to which it'll write the site's output html, and measures the time it'll take. The script as is gives the following input:

```
host: www.google.com.ar
output file absolute path: /tmp/google-ar.html
add another? (y/N) y
host: www.google.com.br
output file absolute path: /tmp/google-br.html
add another? (y/N) y
host: www.google.com.co
output file absolute path: /tmp/google-co.html
add another? (y/N) y
host: www.google.cl
output file absolute path: /tmp/google-cl.html
add another? (y/N)
(Crawler-www.google.com.ar) executing get request for parameters: {'path': '/', 'host': ' ↩
    www.google.com.ar', 'port': 80}
(Crawler-www.google.com.br) executing get request for parameters: {'path': '/', 'host': ' ↩
    www.google.com.br', 'port': 80}
(Crawler-www.google.com.co) executing get request for parameters: {'path': '/', 'host': ' ↩
    www.google.com.co', 'port': 80}
(Crawler-www.google.cl) executing get request for parameters: {'path': '/', 'host': 'www. ↩
    google.cl', 'port': 80}
(Crawler-www.google.com.co) got 200 response http code
(Crawler-www.google.com.ar) got 200 response http code
(Crawler-www.google.com.br) got 200 response http code
(Crawler-www.google.com.co) saving 53181 bytes to /tmp/google-co.html
(Crawler-www.google.com.ar) saving 53008 bytes to /tmp/google-ar.html
(Crawler-www.google.com.br) saving 53605 bytes to /tmp/google-br.html
(Crawler-www.google.cl) got 200 response http code
(Crawler-www.google.cl) saving 53069 bytes to /tmp/google-cl.html
```

As the log shows us, threads are running in parallel, a thread runs while the others make I/O operations (write to a file, or connect via http to a server). Now, there is a line commented on the `__main__()` function that says `t.join()`, the join method, causes the current thread to wait for the thread it joined before continuing its execution, and the log looks like:

```
host: www.google.com.ar
output file absolute path: /tmp/google-ar.html
add another? (y/N) y
host: www.google.com.br
output file absolute path: /tmp/google-ar.html
add another? (y/N) y
host: www.google.com.co
output file absolute path: /tmp/google-co.html
add another? (y/N) y
host: www.google.cl
output file absolute path: /tmp/google-cl.html
add another? (y/N)
(Crawler-www.google.com.ar) executing get request for parameters: {'port': 80, 'path': '/', ↩
    'host': 'www.google.com.ar'}
(Crawler-www.google.com.ar) got 200 response http code
(Crawler-www.google.com.ar) saving 52973 bytes to /tmp/google-ar.html
(Crawler-www.google.com.br) executing get request for parameters: {'port': 80, 'path': '/', ↩
    'host': 'www.google.com.br'}
(Crawler-www.google.com.br) got 200 response http code
(Crawler-www.google.com.br) saving 54991 bytes to /tmp/google-ar.html
(Crawler-www.google.com.co) executing get request for parameters: {'port': 80, 'path': '/', ↩
    'host': 'www.google.com.co'}
(Crawler-www.google.com.co) got 200 response http code
(Crawler-www.google.com.co) saving 53172 bytes to /tmp/google-co.html
(Crawler-www.google.cl) executing get request for parameters: {'port': 80, 'path': '/', ' ↩
   host': 'www.google.cl'}
(Crawler-www.google.cl) got 200 response http code
(Crawler-www.google.cl) saving 53110 bytes to /tmp/google-cl.html
```

See? First it crawls google Argentina, then Brazil, and so on. You sure are wondering why would someone do this. Well... this is not the only use case of the `join` method. Imagine these threads where daemon, and you don't have control over that. You would have to instantiate a variable which holds the amount of threads executed and then wait for it to equal the number of threads that must be executed before exiting the main thread. It's not very elegant.

3.2.4 Joining Threads

Well, there is another way, let's make these threads daemon, just to experiment a little bit, and wait for all of them to finish before exiting the main thread:

site-crawler.py

```
import http.client
import threading
import logging

logging.basicConfig(level=logging.INFO, format='(%(threadName)-10s) %(message)s', )


def save(html, file_absolute_path):
    logging.info("saving {} bytes to {}".format(len(html), file_absolute_path))
    with open(file_absolute_path, 'wb+') as file:
        file.write(html)
        file.flush()


def crawl(req):
    logging.info("executing get request for parameters: {}".format(str(req)))
    connection = http.client.HTTPConnection(req["host"], req["port"])
    connection.request("GET", req["path"])
    response = connection.getresponse()
```

```
    logging.info("got {} response http code".format(response.status))
    logging.debug("headers: {}".format(str(response.headers)))
    response_content = response.read()
    logging.debug("actual response: {}".format(response_content))
    return response_content


class MyCrawler(threading.Thread):
    def __init__(self, req, file_path):
        threading.Thread.__init__(self, name="Crawler-{}".format(req["host"]), daemon=True)
        self.req = req
        self.file_path = file_path

    def run(self):
        global executed_crawlers
        html = crawl(self.req)
        save(html, self.file_path)


def __main__():
    continue_input = True
    threads = []
    while continue_input:
        host = input("host: ")
        port = 80  # int(input("port: "))
        path = "/"  # input("path: ")
        file_path = input("output file absolute path: ")
        req = {"host": host, "port": port, "path": path}
        threads.append(MyCrawler(req, file_path))
        continue_input = input("add another? (y/N) ") == "y"

    for t in threads:
        t.start()

    current_thread = threading.currentThread()
    for thread in threading.enumerate():
        if thread is not current_thread:
            thread.join()


__main__()
```

Here, we are creating every thread as daemon, but we are enumerating every active thread by calling `threading.enumerate()` and joining every thread which is not the main one. The behavior remains the same:

```
host: www.google.com.ar
output file absolute path: /tmp/google-ar.html
add another? (y/N) y
host: www.google.com.br
output file absolute path: /tmp/google-br.html
add another? (y/N) y
host: www.google.com.co
output file absolute path: /tmp/google-co.html
add another? (y/N) y
host: www.google.cl
output file absolute path: /tmp/google-cl.html
add another? (y/N)
(Crawler-www.google.com.ar) executing get request for parameters: {'port': 80, 'host': 'www ↩
    .google.com.ar', 'path': '/'}
(Crawler-www.google.com.br) executing get request for parameters: {'port': 80, 'host': 'www ↩
    .google.com.br', 'path': '/'}
(Crawler-www.google.com.co) executing get request for parameters: {'port': 80, 'host': 'www ↩
```

```
    .google.com.co', 'path': '/'}
(Crawler-www.google.cl) executing get request for parameters: {'port': 80, 'host': 'www. ↩
    google.cl', 'path': '/'}
(Crawler-www.google.com.ar) got 200 response http code
(Crawler-www.google.cl) got 200 response http code
(Crawler-www.google.com.br) got 200 response http code
(Crawler-www.google.com.ar) saving 52980 bytes to /tmp/google-ar.html
(Crawler-www.google.cl) saving 53088 bytes to /tmp/google-cl.html
(Crawler-www.google.com.br) saving 53549 bytes to /tmp/google-br.html
(Crawler-www.google.com.co) got 200 response http code
(Crawler-www.google.com.co) saving 53117 bytes to /tmp/google-co.html
```

3.2.5 Time Threads

Another thing that's worthy of being pointed out, is the existence of the class `threading.Timer`. It is basically a subclass of `Thread` which, given a delay and a function, it executes the function after the delay has passed. Also, it can be cancelled at any point.

```
import threading
import time
import logging

logging.basicConfig(level=logging.DEBUG, format='(%(threadName)-10s) %(message)s',)

def delayed():
    logging.debug('worker running')
    return

t1 = threading.Timer(3, delayed)
t1.setName('t1')
t2 = threading.Timer(3, delayed)
t2.setName('t2')

logging.debug('starting timers')
t1.start()
t2.start()

logging.debug('waiting before canceling %s', t2.getName())
time.sleep(2)
logging.debug('canceling %s', t2.getName())
t2.cancel()
logging.debug('done')
```

If we execute it:

```
(MainThread) starting timers
(MainThread) waiting before canceling t2
(MainThread) canceling t2
(MainThread) done
(t1        ) worker running
```

Here, we are creating two timers, both execute the same function after 3 seconds. Then we wait 2 seconds and cancel one of them. In the output we can see only one of the timers executed the delayed function.

This is useful on scenarios where we need to execute some process if something didn't happen in an interval of time, or even for scheduling.

3.2.6 Events: Communication Between Threads

Now, we all know that the idea of using threads is making tasks independent from each other, but some times we need for a thread to wait for an event caused by another. Python provides a way of signaling between threads. To experiment with this, we'll make a race:

race.py

```
import threading


class Racer(threading.Thread):

    def __init__(self, name, start_signal):
        threading.Thread.__init__(self, name=name)
        self.start_signal = start_signal

    def run(self):
        self.start_signal.wait()
        print("I, {}, got to the goal!".format(self.name))


class Race:

    def __init__(self, racer_names):
        self.start_signal = threading.Event()
        self.racers = [Racer(name, self.start_signal) for name in racer_names]
        for racer in self.racers:
            racer.start()

    def start(self):
        self.start_signal.set()


def __main__():
    race = Race(["rabbit", "turtle", "cheetah", "monkey", "cow", "horse", "tiger", "lion"])
    race.start()


__main__()
```

We create a couple of threads, an then set the event, so they all try to start at the same time, the output is interesting:

first run output

```
I, rabbit, got to the goal!
I, lion, got to the goal!
I, turtle, got to the goal!
I, cheetah, got to the goal!
I, monkey, got to the goal!
I, cow, got to the goal!
I, horse, got to the goal!
I, tiger, got to the goal!
```

second run output

```
I, lion, got to the goal!
I, turtle, got to the goal!
I, monkey, got to the goal!
I, horse, got to the goal!
I, cow, got to the goal!
I, tiger, got to the goal!
I, cheetah, got to the goal!
```

```
I, rabbit, got to the goal!
```

Here we can see how the rabbit won the first race, but ended last on the second. Either he got tired, or our event behaves just as we wanted it to behave.

If we didn't use the event, and start each thread in a loop, the first thread would have an advantage of milliseconds over the last one. And we all know every millisecond counts on computer times.

3.2.7 Locking Resources

Sometimes we have a couple threads accessing the same resource, and if it's not thread safe, we don't want threads to access it at the same time. One solution to this problem could be locking.

A side note: Python's built-in data structures (lists, dictionaries, etc.) are thread-safe as a side-effect of having atomic byte-codes for manipulating them (the GIL is not released in the middle of an update). Other data structures implemented in Python, or simpler types like integers and floats, don't have that protection.

Let's imagine, just to make a fun example, that dictionaries in python are not thread safe. We'll make an on-memory repository and make a couple of threads read and write data to it:

locking.py

```
import random
import threading
import logging

logging.basicConfig(level=logging.INFO,
                    format='[%(levelname)s] (%(threadName)-s) (%(module)-s) (%(funcName)-s) ↩
                        %(message)s',
                    filename='/tmp/locking-py.log')


class Repository:
    def __init__(self):
        self.repo = {}
        self.lock = threading.Lock()

    def create(self, entry):
        logging.info("waiting for lock")
        self.lock.acquire()
        try:
            logging.info("acquired lock")
            new_id = len(self.repo.keys())
            entry["id"] = new_id
            self.repo[new_id] = entry
        finally:
            logging.info("releasing lock")
            self.lock.release()

    def find(self, entry_id):
        logging.info("waiting for lock")
        self.lock.acquire()
        try:
            logging.info("acquired lock")
            return self.repo[entry_id]
        except KeyError:
            return None
        finally:
            logging.info("releasing lock")
            self.lock.release()

    def all(self):
```

```
        logging.info("waiting for lock")
        self.lock.acquire()
        try:
            logging.info("acquired lock")
            return self.repo
        finally:
            logging.info("releasing lock")
            self.lock.release()


class ProductRepository(Repository):
    def __init__(self):
        Repository.__init__(self)

    def add_product(self, description, price):
        self.create({"description": description, "price": price})


class PurchaseRepository(Repository):
    def __init__(self, product_repository):
        Repository.__init__(self)
        self.product_repository = product_repository

    def add_purchase(self, product_id, qty):
        product = self.product_repository.find(product_id)
        if product is not None:
            total_amount = product["price"] * qty
            self.create({"product_id": product_id, "qty": qty, "total_amount": total_amount ↩
                })

    def sales_by_product(self, product_id):
        sales = {"product_id": product_id, "qty": 0, "total_amount": 0}
        all_purchases = self.all()
        for k in all_purchases:
            purchase = all_purchases[k]
            if purchase["product_id"] == sales["product_id"]:
                sales["qty"] += purchase["qty"]
                sales["total_amount"] += purchase["total_amount"]
        return sales


class Buyer(threading.Thread):
    def __init__(self, name, product_repository, purchase_repository):
        threading.Thread.__init__(self, name="Buyer-" + name)
        self.product_repository = product_repository
        self.purchase_repository = purchase_repository

    def run(self):
        for i in range(0, 1000):
            max_product_id = len(self.product_repository.all().keys())
            product_id = random.randrange(0, max_product_id + 1, 1)
            qty = random.randrange(0, 100, 1)
            self.purchase_repository.add_purchase(product_id, qty)


class ProviderAuditor(threading.Thread):
    def __init__(self, product_id, purchase_repository):
        threading.Thread.__init__(self, name="Auditor-product_id=" + str(product_id))
        self.product_id = product_id
        self.purchase_repository = purchase_repository

    def run(self):
```

```
        logging.info(str(self.purchase_repository.sales_by_product(self.product_id)))


def __main__():
    product_repository = ProductRepository()
    purchase_repository = PurchaseRepository(product_repository)

    input_another_product = True
    while input_another_product:
        description = input("product description: ")
        price = float(input("product price: "))
        product_repository.add_product(description, price)
        input_another_product = input("continue (y/N): ") == "y"

    buyers = [Buyer("carlos", product_repository, purchase_repository),
              Buyer("juan", product_repository, purchase_repository),
              Buyer("mike", product_repository, purchase_repository),
              Buyer("sarah", product_repository, purchase_repository)]

    for b in buyers:
        b.start()
        b.join()

    for i in product_repository.all():
        ProviderAuditor(i, purchase_repository).start()


__main__()
```

As you see, both resources (purchases and products) are extending a class `Repository` which has locks for every access method (let's assume every developer will know that he mustn't access the repository's dictionary directly).

This lock will guarantee that only one thread at a time can access one repository. One thing to notice is how the lock is released in a `finally` block, you should be very careful with that. If you don't put the release in a finally block, whenever an exception is raised and interrupts the function's execution, your lock will not be released, and there will be no way to access that resource anymore.

Now, let's execute this code and input something like:

```
product description: a
product price: 1
continue (y/N): y
product description: b
product price: 2
continue (y/N): y
product description: c
product price: 3
continue (y/N): y
product description: d
product price: 4
continue (y/N): y
product description: e
product price: 5
continue (y/N): y
product description: f
product price: 6
continue (y/N): y
product description: g
product price: 7
continue (y/N): y
product description: h
product price: 8
```

```
continue (y/N): y
product description: i
product price: 9
continue (y/N):
```

As you see, the logger was configured to output its log to a file. We don't care about the logging of the `Buyer` threads, since they perform a thousand actions each. That log won't be readable, BUT, `ProviderAuditor` threads will log some very interesting information. So we run `grep "Auditor" /tmp/locking-py.log` and see:

```
[INFO] (Auditor-product_id=0) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=0) (locking) (all) acquired lock
[INFO] (Auditor-product_id=0) (locking) (all) releasing lock
[INFO] (Auditor-product_id=1) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=0) (locking) (run) {'total_amount': 19850.0, 'product_id': 0, ' ↩
    qty': 19850}
[INFO] (Auditor-product_id=2) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=1) (locking) (all) acquired lock
[INFO] (Auditor-product_id=3) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=4) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=5) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=1) (locking) (all) releasing lock
[INFO] (Auditor-product_id=6) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=7) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=1) (locking) (run) {'total_amount': 41586.0, 'product_id': 1, ' ↩
    qty': 20793}
[INFO] (Auditor-product_id=2) (locking) (all) acquired lock
[INFO] (Auditor-product_id=2) (locking) (all) releasing lock
[INFO] (Auditor-product_id=2) (locking) (run) {'total_amount': 60294.0, 'product_id': 2, ' ↩
    qty': 20098}
[INFO] (Auditor-product_id=3) (locking) (all) acquired lock
[INFO] (Auditor-product_id=3) (locking) (all) releasing lock
[INFO] (Auditor-product_id=3) (locking) (run) {'total_amount': 86752.0, 'product_id': 3, ' ↩
    qty': 21688}
[INFO] (Auditor-product_id=4) (locking) (all) acquired lock
[INFO] (Auditor-product_id=8) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=4) (locking) (all) releasing lock
[INFO] (Auditor-product_id=4) (locking) (run) {'total_amount': 93960.0, 'product_id': 4, ' ↩
    qty': 18792}
[INFO] (Auditor-product_id=5) (locking) (all) acquired lock
[INFO] (Auditor-product_id=5) (locking) (all) releasing lock
[INFO] (Auditor-product_id=5) (locking) (run) {'total_amount': 109776.0, 'product_id': 5, ' ↩
    qty': 18296}
[INFO] (Auditor-product_id=6) (locking) (all) acquired lock
[INFO] (Auditor-product_id=6) (locking) (all) releasing lock
[INFO] (Auditor-product_id=6) (locking) (run) {'total_amount': 140945.0, 'product_id': 6, ' ↩
    qty': 20135}
[INFO] (Auditor-product_id=7) (locking) (all) acquired lock
[INFO] (Auditor-product_id=7) (locking) (all) releasing lock
[INFO] (Auditor-product_id=7) (locking) (run) {'total_amount': 164152.0, 'product_id': 7, ' ↩
    qty': 20519}
[INFO] (Auditor-product_id=8) (locking) (all) acquired lock
[INFO] (Auditor-product_id=8) (locking) (all) releasing lock
[INFO] (Auditor-product_id=8) (locking) (run) {'total_amount': 182475.0, 'product_id': 8, ' ↩
    qty': 20275}
```

There are our 8 `ProviderAuditor` threads, the first one to acquire the lock is the Auditor-product_id=0, then releases it and prints our sales. Then goes Auditor-product_id=1 and 2, 3, 4 and 5 are waiting. And it goes on and on. Now (again, imagining python's dictionaries are not thread safe) our resources are thread safe.

Another side note here: Let's imagine another scenario. We have a thread and a resource. The thread locks de resource to write some data, and in the middle, it needs to lock it again to read some other data without releasing the first lock. Well, we have a

problem here... Normal `Lock` objects can not be acquired more than once, even by the same thread. Changing it is easy, just substitute `Lock` with `RLock`, which is a Re-entrant Lock and will provide access to a locked resource, only to the thread which performed the lock.

Locks implement the context manager API and are compatible with the with statement. Using with removes the need to explicitly acquire and release the lock. Let's modify our `Repository` class code to make it prettier:

locking.py

```
class Repository:
    def __init__(self):
        self.repo = {}
        self.lock = threading.Lock()

    def create(self, entry):
        logging.info("waiting lock")
        with self.lock:
            logging.info("acquired lock")
            new_id = len(self.repo.keys())
            entry["id"] = new_id
            self.repo[new_id] = entry

    def find(self, entry_id):
        logging.info("waiting for lock")
        with self.lock:
            try:
                logging.info("acquired lock")
                return self.repo[entry_id]
            except KeyError:
                return None

    def all(self):
        logging.info("waiting for lock")
        with self.lock:
            logging.info("acquired lock")
            return self.repo
```

And the behavior remains the same:

```
[INFO] (Auditor-product_id=0) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=0) (locking) (all) acquired lock
[INFO] (Auditor-product_id=1) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=0) (locking) (run) {'product_id': 0, 'total_amount': 19098.0, ' ↩
    qty': 19098}
[INFO] (Auditor-product_id=2) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=1) (locking) (all) acquired lock
[INFO] (Auditor-product_id=3) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=4) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=5) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=1) (locking) (run) {'product_id': 1, 'total_amount': 36344.0, ' ↩
    qty': 18172}
[INFO] (Auditor-product_id=6) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=2) (locking) (all) acquired lock
[INFO] (Auditor-product_id=7) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=8) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=9) (locking) (all) waiting for lock
[INFO] (Auditor-product_id=2) (locking) (run) {'product_id': 2, 'total_amount': 57555.0, ' ↩
    qty': 19185}
[INFO] (Auditor-product_id=3) (locking) (all) acquired lock
[INFO] (Auditor-product_id=3) (locking) (run) {'product_id': 3, 'total_amount': 72292.0, ' ↩
    qty': 18073}
[INFO] (Auditor-product_id=4) (locking) (all) acquired lock
```

```
[INFO] (Auditor-product_id=4) (locking) (run) {'product_id': 4, 'total_amount': 88835.0, ' ↩
    qty': 17767}
[INFO] (Auditor-product_id=5) (locking) (all) acquired lock
[INFO] (Auditor-product_id=5) (locking) (run) {'product_id': 5, 'total_amount': 110754.0, ' ↩
    qty': 18459}
[INFO] (Auditor-product_id=6) (locking) (all) acquired lock
[INFO] (Auditor-product_id=6) (locking) (run) {'product_id': 6, 'total_amount': 129766.0, ' ↩
    qty': 18538}
[INFO] (Auditor-product_id=7) (locking) (all) acquired lock
[INFO] (Auditor-product_id=7) (locking) (run) {'product_id': 7, 'total_amount': 152576.0, ' ↩
    qty': 19072}
[INFO] (Auditor-product_id=8) (locking) (all) acquired lock
[INFO] (Auditor-product_id=8) (locking) (run) {'product_id': 8, 'total_amount': 150210.0, ' ↩
    qty': 16690}
[INFO] (Auditor-product_id=9) (locking) (all) acquired lock
[INFO] (Auditor-product_id=9) (locking) (run) {'product_id': 9, 'total_amount': 160150.0, ' ↩
    qty': 16015}
```

3.2.8 Limiting Concurrent Access to Resources

Using a lock like that, ensures only one thread at a time can access a resource, but imagine a data base access. Maybe you have a connection pool of, at most, 100 connections. In this case you want concurrent access, but you don't want more than a 100 threads to access this resource at once. `Semaphore` comes to help, it tells the `Lock` how many threads can acquire lock at once. Let's modify our `ProviderAuditor` code to let at most 5 providers acquire lock at the same time:

locking.py

```
class ProviderAuditor(threading.Thread):
    def __init__(self, product_id, purchase_repository):
        threading.Thread.__init__(self, name="Auditor-product_id=" + str(product_id))
        self.product_id = product_id
        self.purchase_repository = purchase_repository
        self.semaphore = threading.Semaphore(5)

    def run(self):
        with self.semaphore:
            logging.info(str(self.purchase_repository.sales_by_product(self.product_id)))
```

3.2.9 Thread-Specific Data

We often need data to be only accessible from one thread (eg.: identifiers of the current process), python provides the `local()` method which returns data that is only accessible from one thread, here goes an example:

process-identifier.py

```
import logging
import random

from threading import Thread, local

logging.basicConfig(level=logging.INFO,
                    format='[%(levelname)s] (%(threadName)-s) (%(module)-s) (%(funcName)-s) ↩
                        %(message)s',)


def my_method(data):
    try:
        logging.info(str(data.value))
    except AttributeError:
```

```
        logging.info("data does not have a value yet")


class MyProcess(Thread):
    def __init__(self):
        Thread.__init__(self)

    def run(self):
        data = local()
        my_method(data)
        data.value = {"process_id": random.randint(0, 1000)}
        my_method(data)

for i in range(0, 4):
    MyProcess().start()
```

It's pretty easy to use, a very useful. If you run it you'll see something like:

```
[INFO] (Thread-1) (process-identifier) (my_method) data does not have a value yet
[INFO] (Thread-1) (process-identifier) (my_method) {'process_id': 567}
[INFO] (Thread-2) (process-identifier) (my_method) data does not have a value yet
[INFO] (Thread-2) (process-identifier) (my_method) {'process_id': 477}
[INFO] (Thread-3) (process-identifier) (my_method) data does not have a value yet
[INFO] (Thread-3) (process-identifier) (my_method) {'process_id': 812}
[INFO] (Thread-4) (process-identifier) (my_method) data does not have a value yet
[INFO] (Thread-4) (process-identifier) (my_method) {'process_id': 981}
```

Every thread has to initialize `local().value`, data from other threads will never be available there.

Chapter 4

Logging Example

Logging is a process through which an application pushes strings to a handler. That string should be a line containing information about the piece of code from which it was sent and the context in that moment.

It is a feature every application must have, and it's as important as any other functionality you add to the application in question. It is the easiest way to know what your application is doing, how, how long does it take, etc.

Any application which is not logging, will be a real pain to maintain, for there is no easy way to tell if it's working or not, and if it's not, why.

Through the log, we should be able to know a lot of useful information about our application:

- What is it doing?
- How is it doing it?
- How long does it take it?
- Is it failing? Why?

4.1 The Theory

Python provides a module called `logging` which has some really useful tools and configurations. But before we jump into the code, let's look around some concepts.

4.1.1 Log Levels

Log levels are the levels of severity of each log line. Is a good solution to differentiate errors from common information in our log files.

The `logging` module provides 5 levels of severity: `DEBUG, INFO, WARNING, ERROR, CRITICAL`. Python's logging documentations defines each of them as:

- **DEBUG**: Detailed information, typically of interest only when diagnosing problems.
- **INFO**: Confirmation that things are working as expected.
- **WARNING**: An indication that something unexpected happened, or indicative of some problem in the near future (e.g. 'disk space low'). The software is still working as expected.
- **ERROR**: Due to a more serious problem, the software has not been able to perform some function.
- **CRITICAL**: A serious error, indicating that the program itself may be unable to continue running.

Loggers can be configured to output only lines above a threshold, which is configured as the global log level of that logger.

4.1.2 Handlers

As we said, a logger is an object which outputs strings to a handler. The handler receives the record and processes it, which means, in some cases, output to a file, console or even by UDP.

The `logging` module provides some useful handlers, and of course you can extend them and create your own. A list with handlers provided by Python below:

- **StreamHandler**: instances send messages to streams (file-like objects).
- **FileHandler**: instances send messages to disk files.
- **BaseRotatingHandler**: is the base class for handlers that rotate log files at a certain point. It is not meant to be instantiated directly. Instead, use RotatingFileHandler or TimedRotatingFileHandler.
- **RotatingFileHandler**: instances send messages to disk files, with support for maximum log file sizes and log file rotation.
- **TimedRotatingFileHandler**: instances send messages to disk files, rotating the log file at certain timed intervals.
- **SocketHandler**: instances send messages to TCP/IP sockets. Since 3.4, Unix domain sockets are also supported.
- **DatagramHandler**: instances send messages to UDP sockets. Since 3.4, Unix domain sockets are also supported.
- **SMTPHandler**: instances send messages to a designated email address.
- **SysLogHandler**: instances send messages to a Unix syslog daemon, possibly on a remote machine.
- **NTEventLogHandler**: instances send messages to a Windows NT/2000/XP event log.
- **MemoryHandler**: instances send messages to a buffer in memory, which is flushed whenever specific criteria are met.
- **HTTPHandler**: instances send messages to an HTTP server using either GET or POST semantics.
- **WatchedFileHandler**: instances watch the file they are logging to. If the file changes, it is closed and reopened using the file name. This handler is only useful on Unix-like systems; Windows does not support the underlying mechanism used.
- **QueueHandler**: instances send messages to a queue, such as those implemented in the queue or multiprocessing modules.
- **NullHandler**: instances do nothing with error messages. They are used by library developers who want to use logging, but want to avoid the 'No handlers could be found for logger XXX' message which can be displayed if the library user has not configured logging.

The NullHandler, StreamHandler and FileHandler classes are defined in the core logging package. The other handlers are defined in a sub-module, logging.handlers.

4.1.3 Format

There is some data that we want to be present in every line of log. Context information such as the thread name, time, severity, module and method. Python provides a way to format every message appending this information without explicitly having to add it to the message.

This is done by providing a format to the logger's configuration. This format consists of attribute names that represent that context data. Below is the list of every available attribute:

- **args** (You shouldn't need to format this yourself): The tuple of arguments merged into msg to produce message, or a dict whose values are used for the merge (when there is only one argument, and it is a dictionary).
- **asctime** (`%(asctime)s`): Human-readable time when the LogRecord was created. By default this is of the form `2003-07-08 16:49:45,896' (the numbers after the comma are millisecond portion of the time).
- **created** (`%(created)f`): Time when the LogRecord was created (as returned by time.time()).
- **exc_info** (You shouldn't need to format this yourself): Exception tuple or, if no exception has occurred, None.

- **filename** (`%(filename)s`): Filename portion of pathname.
- **funcName** (`%(funcName)s`): Name of function containing the logging call.
- **levelname** (`%(levelname)s`): Text logging level for the message (*DEBUG*, *INFO*, *WARNING*, *ERROR*, *CRITICAL*).
- **levelno** (`%(levelno)s`): Numeric logging level for the message (DEBUG, INFO, WARNING, ERROR, CRITICAL).
- **lineno** (`%(lineno)d`): Source line number where the logging call was issued (if available).
- **module** (`%(module)s`): Module (name portion of filename).
- **msecs** (`%(msecs)d`): Millisecond portion of the time when the LogRecord was created.
- **message** (`%(message)s`): The logged message, computed as msg % args. This is set when Formatter.format() is invoked.
- **msg** (You shouldn't need to format this yourself): The format string passed in the original logging call. Merged with args to produce message, or an arbitrary object.
- **name** (`%(name)s`): Name of the logger used to log the call.
- **pathname** (`%(pathname)s`): Full pathname of the source file where the logging call was issued (if available).
- **process** (`%(process)d`): Process ID (if available).
- **processName** (`%(processName)s`): Process name (if available).
- **relativeCreated** (`%(relativeCreated)d`): Time in milliseconds when the LogRecord was created, relative to the time the logging module was loaded.
- **stack_info** (You shouldn't need to format this yourself): Stack frame information (where available) from the bottom of the stack in the current thread, up to and including the stack frame of the logging call which resulted in the creation of this record.
- **thread** (`%(thread)d`): Thread ID (if available).
- **threadName** (`%(threadName)s`): Thread name (if available).

A pretty useful and common format would look like:

```
'%(asctime)s - [%(levelname)s] [%(threadName)s] (%(module)s:%(lineno)d) %(message)s'
```

4.2 The Practice

So, let's experiment a little bit. Let's create a file called `basic.py` and make a logging example in it.

Let's start with checking the behavior of the default logger, without modifying any of its configuration.

basic.py

```
import logging

logging.debug('Some additional information')
logging.info('Working...')
logging.warning('Watch out!')
logging.error('Oh NO!')
logging.critical('x.x')
```

Here we are just importing logging, and logging a line for each severity, to see the format and the threshold. When we run it we see:

```
WARNING:root:Watch out!
ERROR:root:Oh NO!
CRITICAL:root:x.x
```

So, the default format would be something like:

```
'%(levelname)s:%(name)s:%(message)s'
```

And the threshold is warning, as our info and warn lines are not being logged. Let's make it a little nicer:

basic.py

```
import logging

logging.basicConfig(level=logging.INFO,
                    format="%(asctime)s - [%(levelname)s] [%(threadName)s] (%(module)s:%( ↩
                        lineno)d) %(message)s", )

logging.debug('Some additional information')
logging.info('Working...')
logging.warning('Watch out!')
logging.error('Oh NO!')
logging.critical('x.x')
```

And the output:

```
2015-12-30 16:43:39,600 - [INFO] [MainThread] (basic:7) Working...
2015-12-30 16:43:39,600 - [WARNING] [MainThread] (basic:8) Watch out!
2015-12-30 16:43:39,600 - [ERROR] [MainThread] (basic:9) Oh NO!
2015-12-30 16:43:39,600 - [CRITICAL] [MainThread] (basic:10) x.x
```

We've declared a threshold of `INFO` and a format which gives a lot of useful information. We now have the time, the severity, the thread, the module and the line in every log record.

Let's make a tiny application which would save a contact list to a CSV file and then retrieve them, and then let's add some useful log to it:

contacts.py

```
import csv
import os.path


class Repository:
    def __init__(self, full_file_path):
        self.full_file_path = full_file_path

    def add(self, contact):
        headers = [h for h in contact]
        headers.sort()
        write_headers = not os.path.isfile(
            self.full_file_path)  # let's assume no one will go and erase the headers by  ↩
                hand
        with open(self.full_file_path, 'a+') as file:
            writer = csv.DictWriter(file, fieldnames=headers)
            if write_headers:
                writer.writeheader()
            writer.writerow(contact)

    def names(self):
        with open(self.full_file_path, 'r+') as file:
            names = list(map(lambda r: r['name'], csv.DictReader(file)))
            return names

    def full_contact(self, name):
        with open(self.full_file_path, 'r+') as file:
            for contact in list(csv.DictReader(file)):
```

```
                if contact['name'] == name:
                    return contact
            return


class Main:

    def __init__(self, contacts_file):
        self.repo = Repository(contacts_file)

    def create(self):
        name = input("name: ")
        number = input("number: ")
        contact = {"name": name, "number": number}
        self.repo.add(contact)

    def names(self):
        names = self.repo.names()
        if len(names) > 0:
            for n in names:
                print("- {}".format(n))
        else:
            print("no contacts were found")

    def full_contact(self):
        name = input("name: ")
        contact = self.repo.full_contact(name)
        if contact is not None:
            print("name: {}".format(contact["name"]))
            print("number: {}".format(contact["number"]))
        else:
            print("contact not found.")

    def menu(self):
        actions = {"1": self.create, "2": self.names, "3": self.full_contact}
        options = ["1) Create Contact", "2) All Contacts", "3) Detail of a contact", "0)  ↩
            Exit"]
        for o in options:
            print(o)
        selected = input("What do you want to do? ")
        if selected in actions:
            actions[selected]()
            self.menu()

Main("/tmp/contacts.csv").menu()
```

So now we run it and create two contacts: Michael and Sarah.

```
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 1
name: Michael
number: 1234-5678
Done!
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 2
- Michael
```

```
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 1
name: Sarah
number: 9876-5432
Done!
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 2
- Michael
- Sarah
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 3
name: Sarah
name: Sarah
number: 9876-5432
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 0
```

What if we add some logging to know what's going on there?

contacts.py

```
import csv
import os.path
import logging

logging.basicConfig(level=logging.INFO,
                    format="%(asctime)s - [%(levelname)s] [%(threadName)s] (%(module)s:%( ↩
                        lineno)d) %(message)s", )


class Repository:
    def __init__(self, full_file_path):
        logging.info("initializing contacts repository with file at: {}".format( ↩
            full_file_path))
        self.full_file_path = full_file_path

    def add(self, contact):
        logging.info("creating contact: {}".format(contact))
        headers = [h for h in contact]
        headers.sort()
        write_headers = not os.path.isfile(
            self.full_file_path)  # let's assume no one will go and erase the headers by  ↩
                hand
        with open(self.full_file_path, 'a+') as file:
            writer = csv.DictWriter(file, fieldnames=headers)
            if write_headers:
                logging.debug("this is the first contact in the given file. writing headers ↩
                    .")
                writer.writeheader()
            writer.writerow(contact)
```

```
    def names(self):
        logging.info("retrieving all contact names")
        with open(self.full_file_path, 'r+') as file:
            names = list(map(lambda r: r['name'], csv.DictReader(file)))
            logging.debug("found {} contacts".format(len(names)))
            return names

    def full_contact(self, name):
        logging.info("retrieving full contact for name: {}".format(name))
        with open(self.full_file_path, 'r+') as file:
            for contact in list(csv.DictReader(file)):
                if contact['name'] == name:
                    logging.debug("contact was found")
                    return contact
            logging.warning("contact was not found for name: {}".format(name))
            return


class Main:

    def __init__(self, contacts_file):
        self.repo = Repository(contacts_file)

    def create(self):
        name = input("name: ")
        number = input("number: ")
        contact = {"name": name, "number": number}
        self.repo.add(contact)

    def names(self):
        names = self.repo.names()
        if len(names) > 0:
            for n in names:
                print("- {}".format(n))
        else:
            print("no contacts were found")

    def full_contact(self):
        name = input("name: ")
        contact = self.repo.full_contact(name)
        if contact is not None:
            print("name: {}".format(contact["name"]))
            print("number: {}".format(contact["number"]))
        else:
            print("contact not found.")

    def menu(self):
        actions = {"1": self.create, "2": self.names, "3": self.full_contact}
        options = ["1) Create Contact", "2) All Contacts", "3) Detail of a contact", "0)  ↩
            Exit"]
        for o in options:
            print(o)
        selected = input("What do you want to do? ")
        if selected in actions:
            actions[selected]()
            self.menu()

Main("/tmp/contacts.csv").menu()
```

So, now doing the exact same thing we'll see:

```
2015-12-30 17:32:15,788 - [INFO] [MainThread] (contacts:11) initializing contacts  ↩
    repository with file at: /tmp/contacts.csv
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 1
name: Michael
number: 1234-5678
2015-12-30 17:32:33,732 - [INFO] [MainThread] (contacts:15) creating contact: {'number': ' ↩
    1234-5678', 'name': 'Michael'}
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 1
name: Sarah
number: 9876-5432
2015-12-30 17:32:41,828 - [INFO] [MainThread] (contacts:15) creating contact: {'number': ' ↩
    9876-5432', 'name': 'Sarah'}
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 2
2015-12-30 17:32:45,140 - [INFO] [MainThread] (contacts:28) retrieving all contact names
- Michael
- Sarah
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 3
name: Sarah
2015-12-30 17:32:48,532 - [INFO] [MainThread] (contacts:35) retrieving full contact for  ↩
    name: Sarah
name: Sarah
number: 9876-5432
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 0
```

Now, there is our logging, but there is also a new problem. The user of our script doesn't care about our log lines, they actually bothers him. Let's log to a file modifying only the configuration of the logger:

contacts.py

```
...
logging.basicConfig(level=logging.INFO,
                    format="%(asctime)s - [%(levelname)s] [%(threadName)s] (%(module)s:%( ↩
                        lineno)d) %(message)s",
                    filename="/tmp/contacts.log")
...
```

So, now in the output we see everything normal again:

```
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
```

```
What do you want to do? 2
- Michael
- Sarah
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 3
name: Michael
name: Michael
number: 1234-5678
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 3
name: Qwerty
contact not found.
1) Create Contact
2) All Contacts
3) Detail of a contact
0) Exit
What do you want to do? 0
```

No log lines for the world to see. Not our script is actually usable again. So with a simple `cat /tmp/contacts.log` we can see what happened there:

```
2015-12-30 17:34:54,112 - [INFO] [MainThread] (contacts:12) initializing contacts  ↩
    repository with file at: /tmp/contacts.csv
2015-12-30 17:34:56,219 - [INFO] [MainThread] (contacts:29) retrieving all contact names
2015-12-30 17:35:01,819 - [INFO] [MainThread] (contacts:36) retrieving full contact for  ↩
    name: Michael
2015-12-30 17:35:06,827 - [INFO] [MainThread] (contacts:36) retrieving full contact for  ↩
    name: Qwerty
2015-12-30 17:35:06,827 - [WARNING] [MainThread] (contacts:42) contact was not found for  ↩
    name: Qwerty
```

4.3 Download the Code Project

This was an example on Python Logging.

Download You can download the full source code of this example here: **python-logging**

Chapter 5

Django Tutorial

Django is an open source web framework which solves most common problems of a web application development. It has some very useful features like an auto generated admin page and an ORM.

It's a high-level framework, so we just need to focus on the actual core logic of our business and let Django take care of mappings, filters and such. It works by defining some standards that, if you follow them, will make the application's development so much easier and faster.

By using Python 3.4.3 and Django 1.9.1, we'll build a web application which, through an admin site only available to us, will let us upload riddles. Then we'll make a public user interface which will let the users answer the riddles we upload.

5.1 Creating the project

If you've installed django correctly, when you run `django-admin --version`, you'll see the version in the output of that command. Now if we run `django-admin startproject django_example`, we'll now have the skeleton of our project. Let's cd to our project's directory and see:

- **django_example** is the actual python package for your project.
- **django_example/*init*.py** is an empty file that tells python that this directory should be considered a package.
- **django_example/settings.py** holds the configuration for this Django project.
- **django_example/urls.py** holds the URL declarations.
- **django_example/wsgi.py** is an entry-point for WSGI-compatible web servers to serve your project.
- **manage.py** is a command-line utility that lets you interact with this Django project in various ways. We'll see some of them below.

Now, let's see how to run this application. Run `python3 manage.py runserver` and you'll see some output on the console to finally see `Starting development server at https://127.0.0.1:8000/`. Now if we hit that location we'll see a Django welcome screen telling us that everything worked fine.

You will probably see something like "You have unapplied migrations; your app may not work properly until they are applied.". We'll get back to that later, don't worry about it right now.

5.2 Creating our application

Now we have our project up and running and we can start writing our application now. Something to notice here is the difference about the terms project and application. An application is a python program that does something, and a project is a group of

applications. The same application can be contained in more than one project. A project will almost always contain many applications.

Each application in Django consists of a package that follows some Django standards. One of the many tools that come with this framework is an utility that automatically generates the basic directory structure of an application, so let's run `python3 manage.py startapp riddles`, and now there will be a directory called riddles wich will hold our application's source code.

Let's write a simple view to see how everything works. Django convention tells us to write them in a file called `views.py` within our application's module.

riddles/views.py

```
from django.http import HttpResponse


def index(request):
    return HttpResponse("Hello, World!")
```

Now, to map this view to a URL we'll create a `urls.py` in our application's module.

riddles/urls.py

```
from django.conf.urls import url

from . import views

app_name = 'riddles'

urlpatterns = [
    url(r'^$', views.index, name='index'),
]
```

And in our project's urls we include this urls.

django_example/urls.py

```
from django.conf.urls import include, url
from django.contrib import admin

urlpatterns = [
    url(r'^riddles/', include('riddles.urls')),
    url(r'^admin/', admin.site.urls),
]
```

Now we run our server and hit https://localhost:8000/riddles/ to see our message "Hello, World!". If everything worked fine we'll start with our database.

5.3 Database Setup

Now, when we see `django_example/settings.py`, it's a normal module with some variables representing Django settings. There is a variable named `DATABASES` which by default uses SQLite which is included within Python, so you won't need to install anything. For this example, we'll stick with the default here.

Also, see the variable `INSTALLED_APPS`, it holds the names of all Django applications which are activated in the project. By default it contains:

- ***django.contrib.admin*** is the admin site, we'll be seeing it in no time.
- ***django.contrib.auth*** as an authentication system.
- ***django.contrib.contenttypes*** a framework for content types.

- ***django.contrib.sessions*** a session framework.
- ***django.contrib.messages*** a messaging framework.
- ***django.contrib.staticfiles*** a static content framework.

This applications are useful in most cases, but some of them uses databases (like django.contrib.auth), and these are not created yet. That's why you see that migrations message on the output of the runserver command. To create all tables we just run the command `python3 manage.py migrate`. You should see something like:

```
Operations to perform:
  Apply all migrations: admin, sessions, auth, contenttypes
Running migrations:
  Rendering model states... DONE
  Applying contenttypes.0001_initial... OK
  Applying auth.0001_initial... OK
  Applying admin.0001_initial... OK
  Applying admin.0002_logentry_remove_auto_add... OK
  Applying contenttypes.0002_remove_content_type_name... OK
  Applying auth.0002_alter_permission_name_max_length... OK
  Applying auth.0003_alter_user_email_max_length... OK
  Applying auth.0004_alter_user_username_opts... OK
  Applying auth.0005_alter_user_last_login_null... OK
  Applying auth.0006_require_contenttypes_0002... OK
  Applying auth.0007_alter_validators_add_error_messages... OK
  Applying sessions.0001_initial... OK
```

Now let's create our model, which is our database layout with some additional metadata. For now we'll just create `Riddle` and `Option`. A Riddle contains the actual riddle, and an Option is one of the possible answers of a riddle, with a flag that will tell us if it's the correct one.

riddles/models.py

```
from django.db import models


class Riddle(models.Model):
    riddle_text = models.CharField(max_length=255)
    pub_date = models.DateTimeField('date published')


class Option(models.Model):
    riddle = models.ForeignKey(Riddle, on_delete=models.CASCADE)
    text = models.CharField(max_length=255)
    correct = models.BooleanField(default=False)
```

This model gives Django all the information needed to create the database schema and the database-access API for accessing these objects, but we need to tell our project that the riddles app is installed, we'll just add an element to the `INSTALLED_APPS` in `django_example/settings.py`:

django_example/settings.py

```
INSTALLED_APPS = [
    'riddles.apps.RiddlesConfig',
    'django.contrib.admin',
    'django.contrib.auth',
    'django.contrib.contenttypes',
    'django.contrib.sessions',
    'django.contrib.messages',
    'django.contrib.staticfiles',
]
```

Now we make the migration for this application running `python3 manage.py makemigrations riddles` and we'll see:

```
Migrations for 'riddles':
  0001_initial.py:
    - Create model Option
    - Create model Riddle
    - Add field riddle to option
```

By doing this we are telling Django that we made some changes to the models and that these should be stored as a migration, these are just files on the disk, you can read them in a directory called migrations within your application. They are designed to be human readable so you would be able to change them if you need.

We can see de actual SQL statements that te migration will run if we want by running `python3 manage.py sqlmigrate riddles 0001`. That 0001 is the version of the migration we want to check. In the output we'll see:

```
BEGIN;
--
-- Create model Option
--
CREATE TABLE "riddles_option" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "text"  ↩
    varchar(255) NOT NULL, "correct" bool NOT NULL);
--
-- Create model Riddle
--
CREATE TABLE "riddles_riddle" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, " ↩
    riddle_text" varchar(255) NOT NULL, "pub_date" datetime NOT NULL);
--
-- Add field riddle to option
--
ALTER TABLE "riddles_option" RENAME TO "riddles_option__old";
CREATE TABLE "riddles_option" ("id" integer NOT NULL PRIMARY KEY AUTOINCREMENT, "text"  ↩
    varchar(255) NOT NULL, "correct" bool NOT NULL, "riddle_id" integer NOT NULL REFERENCES  ↩
    "riddles_riddle" ("id"));
INSERT INTO "riddles_option" ("riddle_id", "id", "text", "correct") SELECT NULL, "id", " ↩
    text", "correct" FROM "riddles_option__old";
DROP TABLE "riddles_option__old";
CREATE INDEX "riddles_option_a7c97949" ON "riddles_option" ("riddle_id");

COMMIT;
```

Now we run migrate, and this script will be executed in the database. Notice that the sqlmigrate command was only to check the script it will execute, it's not necessary to run it every time.

Now we can start using the admin page, but we need a user to log in. By running the createsuperuser command we'll be able to create the root user for our application. It will prompt for a name, a mail and a password. Just run `python3 manage.py createsuperuser`. Now if you run the server and visit https://127.0.0.1:8000/admin/ you will be able to log in with the credentials you just provided.

Let's make our model modifiable by the admin. In `riddles/admin.py` we give Django all the models that are modifiable via the admin page. Let's change it to look like this:

riddles/admin.py

```
from django.contrib import admin

from .models import Option, Riddle

admin.site.register(Riddle)
admin.site.register(Option)
```

Now, let's check out that admin page. First, we hit https://localhost:8000/admin, it will redirect us to a login page.

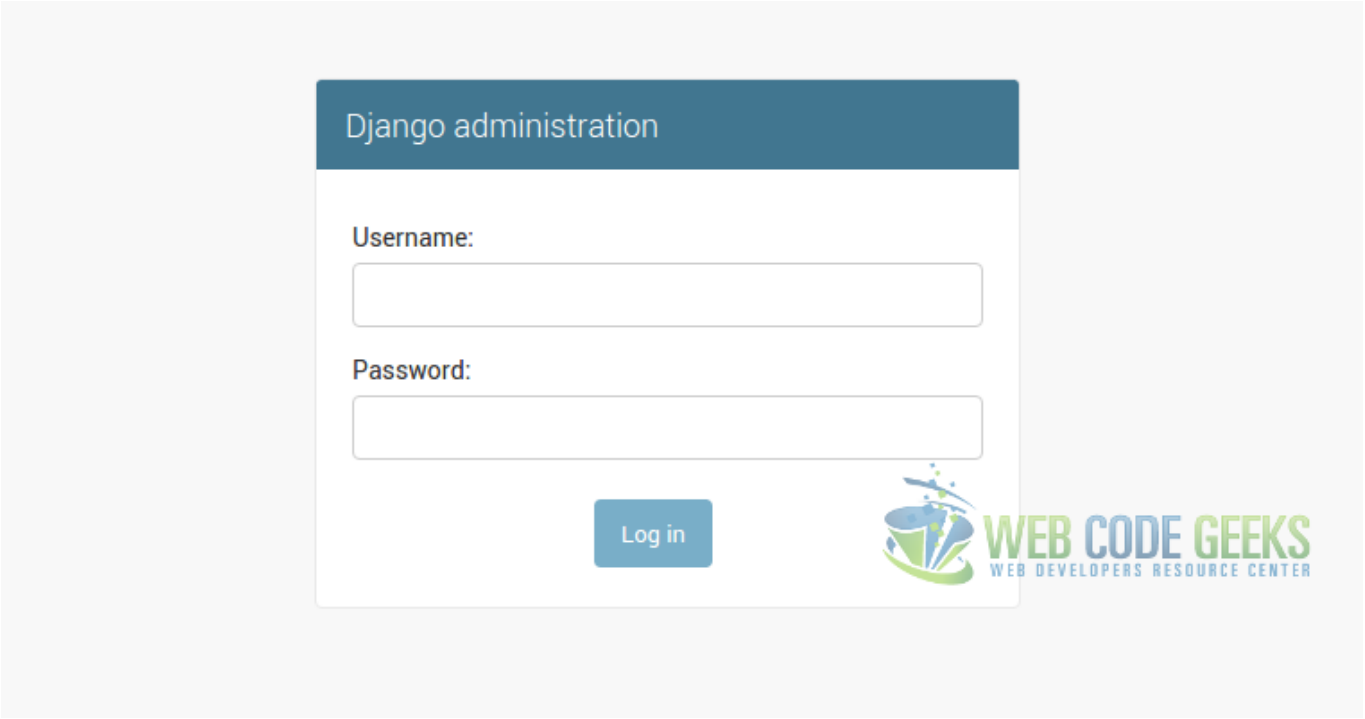

Figure 5.1: Login Page

We input the credentials of the super user we created before, and now we see the home.

Figure 5.2: Home

There we see a couple useful things:

Authentication and Authorization: These are purely administrative data for our project. We can create, modify, delete and list users and groups. **Riddles:** Here it is. We can administer our riddles and options in this page. Full CRUD operations are available out of the box, it all comes free if you follow Django conventions.

Take a little pause here and navigate this admin page. Create a couple riddles and options, even a user. Explore the permissions system and how groups work. It's pretty awesome.

5.4 The Public Page

What do we need to make our public page?

- Templates: The skeleton of the page the user will see.
- Views: The python's function that will serve the page content.

Let's start with the templates. Create a module called templates within your application module, and there we'll write our index.html.

riddles/templates/index.html

```
<h1>Available Riddles</h1>

{% if message %}
    <strong>{{ message }}</strong>
{% endif %}

{% if latest_riddles %}

    <ul>
        {% for riddle in latest_riddles %}
            <li>
                <a href="/riddles/{{ riddle.id }}/">
                    {{ riddle.riddle_text }}
                </a>
            </li>
        {% endfor %}
    </ul>

{% else %}
    No riddles are available right now.
{% endif %}
```

As you can see, this is an actual template with a decision and an iteration. If there are riddles available, it iterates through them and puts its text in a link to try and answer it (will check it out later). If no riddle is available, it informs the user. The answer template looks like this:

riddles/templates/answer.html

```
<h1>{{ riddle.riddle_text }}</h1>

{% if error_message %}

        <strong>{{ error_message }}</strong>

{% endif %}

<form action="answer' riddle.id %}" method="post">
    {% csrf_token %}
    {% for option in riddle.option_set.all %}
        <input type="radio" name="option" id="option{{ forloop.counter }}" value="{{ option ↩
            .id }}" />
```

```
        <label for="option{{ forloop.counter }}">{{ option.text }}</label><br />
    {% endfor %}
    <input type="submit" value="Answer" />
</form>
```

Here, we are informing the riddle text and all the options as radio inputs within a form. There is a `csrf_token` there; its a token provided by Django to avoid Cross Site Request Forgery, every internal form should use it. Pretty simple, but functional. The views to render these templates and answer these requests look like:

riddles/views.py

```
from django.http.response import HttpResponse
from django.shortcuts import get_object_or_404, render

from .models import Riddle, Option


def index(request):
    return render(request, "index.html", {"latest_riddles": Riddle.objects.order_by('- ↩
        pub_date')[:5]})


def detail(request, riddle_id):
    return render(request, "answer.html", {"riddle": get_object_or_404(Riddle, pk=riddle_id ↩
        )})


def answer(request, riddle_id):
    riddle = get_object_or_404(Riddle, pk=riddle_id)
    try:
        option = riddle.option_set.get(pk=request.POST['option'])
    except (KeyError, Option.DoesNotExist):
        return render(request, 'answer.html', {'riddle': riddle, 'error_message': 'Option  ↩
            does not exist'})
    else:
        if option.correct:
            return render(request, "index.html", {"latest_riddles": Riddle.objects.order_by ↩
                ('-pub_date')[:5], "message": "Nice! Choose another one!"})
        else:
            return render(request, 'answer.html', {'riddle': riddle, 'error_message': ' ↩
                Wrong Answer!'})
```

Let's explain one by one each of these functions:

- **index:** Index uses a function called `render`, present in a package called `shortcuts` in Django. This function receives the HttpRequest, the location of the template and its context and returns an HttpReponse with the resulting html.
- **detail:** The detail does pretty much the same, but the function `get_object_or_404` results in an HttpResponse404 if the object is not found for the given primary key.
- **answer:** This function is a little bit more complicated. It looks for the provided riddle (and results in 404 if it is not found), and then it checks that the given option belongs to the given riddle, if not returns a bad request. Once all checks passed, it only checks that the given option has the correct flag in true, and returns accordingly.

Now we just have to map these functions to some urls to make it navigable. And the mapping in urls.py:

riddles/urls.py

```
from django.conf.urls import url

from . import views
```

```
app_name = 'riddles'

urlpatterns = [
    url(r'^$', views.index, name='index'),
    url(r'^(?P[0-9]+)/$', views.detail, name='detail'),
    url(r'^(?P[0-9]+)/answer/$', views.answer, name='answer')
]
```

All set, now by hitting https://localhost:8000/riddles/ we now can see all the riddles we uploaded via de admin page, and choose one to try to answer it.

Now, let's add some CSS to this application. It's pretty ugly right now, right?

5.5 Style Sheets

First, we create a directory called `static` within our application's module and create a `main.css` that makes the site a little bit less ugly.

riddles/static/main.css

```
body{
    margin:40px auto;
    max-width:650px;
    line-height:1.6;
    font-size:18px;
    color:#444;
    padding:0 10px;
}
h1,h2,h3{
    line-height:1.2;
    text-align: center;
}
a {
    color: blue;
}

form {
    margin: 0 auto;
    padding: 1em;
    border: 1px solid #CCC;
    border-radius: 1em;
}
form div + div {
    margin-top: 1em;
}
label {
    display: inline-block;
    text-align: center;
    width: 40%;
}
input {
    font: 1em sans-serif;
    -moz-box-sizing: border-box;
    box-sizing: border-box;
    border: 1px solid #999;
    width: 50%;
}
input:focus {
    border-color: #000;
}
```

```
p, div.button {
    text-align: center;
}

p.error-message {
    color: lightcoral;
}
```

Now, let's change a little bit our templates:

riddles/templates/index.html

```
{% load staticfiles %}

<link rel="stylesheet" type="text/css" href="{% static 'main.css' %}" />

<h1>Available Riddles</h1>

{% if message %}
    <strong>{{ message }}</strong>
{% endif %}

{% if latest_riddles %}

    <ul>
        {% for riddle in latest_riddles %}
            <li>
                <a href="/riddles/{{ riddle.id }}/">
                    {{ riddle.riddle_text }}
                </a>
            </li>
        {% endfor %}
    </ul>

{% else %}
    No riddles are available right now.
{% endif %}
```

riddles/templates/answer.html

```
{% load staticfiles %}

<link rel="stylesheet" type="text/css" href="{% static 'main.css' %}" />

<h1>{{ riddle.riddle_text }}</h1>

{% if error_message %}

        <strong>{{ error_message }}</strong>

{% endif %}

<form action="answer' riddle.id %}" method="post">
    {% csrf_token %}
    {% for option in riddle.option_set.all %}
        <input type="radio" name="option" id="option{{ forloop.counter }}" value="{{ option ↩
            .id }}" />
        <label for="option{{ forloop.counter }}">{{ option.text }}</label><br />
    {% endfor %}
    <input type="submit" value="Answer" />
</form>
```

```
{% load staticfiles %}

<link rel="stylesheet" type="text/css" href="{% static 'main.css' %}" />

<h1>{{ riddle.riddle_text }}</h1>

{% if error_message %}

        <strong>{{ error_message }}</strong>

{% endif %}

<form action="answer' riddle.id %}" method="post">
    {% csrf_token %}
    {% for option in riddle.option_set.all %}
        <input type="radio" name="option" id="option{{ forloop.counter }}" value="{{ option ↩
            .id }}" />
        <label for="option{{ forloop.counter }}">{{ option.text }}</label><br />
    {% endfor %}
    <input type="submit" value="Answer" />
</form>
```

The first line loads static files, and then we use that `{% static '#' %}` where # is the location of the resource you want to import. For javascripts is the exact same procedure, you just need to build the url within a `script` instead of a `link`.

5.6 Download the Code Project

This was a basic example of a Python Django application. You can now build your own, you just saw how easy it is.

Download You can download the full source code of this example here: **python-django-example**

Chapter 6

Dictionary Example

Dictionaries in Python are data structures that optimize element lookups by associating keys to values. You could say that dictionaries are arrays of (key, value) pairs.

6.1 Define

Their syntax is pretty familiar to us, web developers, since its exactly the same as JSON's. An object starts and ends in curly braces, each pair is separated by a comma, and the (key, value) pairs are associated through a colon (:). Let's see:

base_example.py

```
EXAMPLE_DICT = {
    "animals": ["dog", "cat", "fish"],
    "a_number": 1,
    "a_name": "Sebastian",
    "a_boolean": True,
    "another_dict": {
        "you could": "keep going",
        "like this": "forever"
    }
}
```

Here, we are defining a dictionary which contains an array of animals, a number, a name, a boolean and another dictionary inside. This data structure is pretty versatile, as you can see. By the way, to define an empty dictionary you just write `my_dict ={}`.

6.2 Read

Now, let's see how to retrieve a value from this dictionary. There are two ways of doing this: The known "[]" syntax where we write `dict["key"]` and get the result, or the `dict.get("key", "default value")` which also returns the result.

What's the difference? Well, the [] syntax results in a KeyError if the key is not defined, and we usually don't want that. The get method receives the key and an optional default value, and it does not result in any annoying error if the key is not found, it just returns None.

Let's check out the [] syntax a little bit:

base_example.py

```
EXAMPLE_DICT = {
    "animals": ["dog", "cat", "fish"],
    "a_number": 1,
    "a_name": "Sebastian",
```

```
    "a_boolean": True,
    "another_dict": {
        "you could": "keep going",
        "like this": "forever"
    }
}

print(str(EXAMPLE_DICT["animals"]))  # outputs ['dog', 'cat', 'fish']
print(str(EXAMPLE_DICT["a_number"]))  # outputs 1
print(str(EXAMPLE_DICT["this_one_does_not_exist"]))  # throws KeyError
```

In every print statement you will find a comment predicting the result, anyway, let's see the actual output:

```
['dog', 'cat', 'fish']
Traceback (most recent call last):
1
  File "/home/svinci/projects/jcg/dictionaries/base_example.py", line 14, in <module>
    print(str(EXAMPLE_DICT["this_one_does_not_exist"]))
KeyError: 'this_one_does_not_exist'
```

There it is, the key "this_one_does_not_exist" does not exist, so a KeyError is raised. Let's put that KeyError in an except clause, and check the get method.

base_example.py

```
EXAMPLE_DICT = {
    "animals": ["dog", "cat", "fish"],
    "a_number": 1,
    "a_name": "Sebastian",
    "a_boolean": True,
    "another_dict": {
        "you could": "keep going",
        "like this": "forever"
    }
}

default_message = "oops, key not found"

print(str(EXAMPLE_DICT["animals"]))  # outputs ['dog', 'cat', 'fish']
print(str(EXAMPLE_DICT["a_number"]))  # outputs 1

try:
    print(str(EXAMPLE_DICT["this_one_does_not_exist"]))  # throws KeyError
except KeyError:
    print("KeyError")

print(str(EXAMPLE_DICT.get("animals", default_message)))  # outputs ['dog', 'cat', 'fish']
print(str(EXAMPLE_DICT.get("a_number", default_message)))  # outputs 1
print(str(EXAMPLE_DICT.get("this_one_does_not_exist")))  # outputs None
print(str(EXAMPLE_DICT.get("this_one_does_not_exist", default_message)))  # outputs "oops,  ↩
    key not found"
```

Here, we put the missing key in a try/except, which prints "KeyError" on failure. Then we see the get method examples, we ask for a key and return a default message if it's not found. The output:

```
['dog', 'cat', 'fish']
1
KeyError
['dog', 'cat', 'fish']
1
None
oops, key not found
```

It's pretty much the same behavior, but anything that doesn't include a try/except is nicer.

6.3 Write

Dictionaries are mutable data structures. We can add keys to it, update their values and even delete them. Let's see one by one in our example:

base_example.py

```
EXAMPLE_DICT = {
    "animals": ["dog", "cat", "fish"],
    "a_number": 1,
    "a_name": "Sebastian",
    "a_boolean": True,
    "another_dict": {
        "you could": "keep going",
        "like this": "forever"
    }
}


def print_key(dictionary, key):
    print(str(dictionary.get(key, "Key was not found")))

# Create a key
EXAMPLE_DICT["this_one_does_not_exist"] = "it exists now"  # This statement will create the ↩
     key "this_one_does_not_exist" and assign to it the value "it exists now"
print_key(EXAMPLE_DICT, "this_one_does_not_exist")

# Update a key
EXAMPLE_DICT["a_boolean"] = False  # Exactly, it looks the same, and behaves the same. It  ↩
    just overwrites the given key.
print_key(EXAMPLE_DICT, "a_boolean")

# Delete a key
del EXAMPLE_DICT["this_one_does_not_exist"]  # Now "this_one_does_not_exist" ceased from  ↩
    existing (Again).
print_key(EXAMPLE_DICT, "this_one_does_not_exist")
```

There is a method called `print_key` which receives a dictionary and a key. It prints the value of the given key in the given dictionary or "Key was not found" instead.

We are creating the key this_one_does_not_exist first, and printing it out. Then we are updating a_boolean and also printing it. At last, we delete this_one_does_not_exist from the dict and print it again. The result:

```
it exists now
False
Key was not found
```

It works as expected. Notice that, in the context of writing a key, the [] syntax doesn't throw an error if the key was not found, so in this case you can use it without worrying.

6.4 Useful operations

Python provides some useful operations to work with dictionaries, let's see some of them:

6.4.1 In (keyword)

The in keyword is a tool Python provides to, in this case, check whether a key is present in a dictionary or not. Let's make our own implementation of the get method to test this keyword:

dict_get.py

```
def get(dictionary, key, default_value=None):
    if key in dictionary:
        return dictionary[key]
    else:
        return default_value


my_dict = {
    "name": "Sebastian",
    "age": 21
}

print(str(get(my_dict, "name", "Name was not present")))
print(str(get(my_dict, "age", "Age was not present")))
print(str(get(my_dict, "address", "Address was not present")))
```

Withing our custom get method, we use the [] syntax to retrieve the requested key from the given dictionary, but not without checking if the key is present there with the in keyword. We receive a default value which by default is None. It behaves exactly like the native get method, here is the output:

```
Sebastian
21
Address was not present
```

This tool can also be used to iterate over the dictionary keys, let's implement a method which returns an array with all the keys in a dictionary to see it working:

dict_all_keys.py

```
my_dict = {
    "name": "Sebastian",
    "age": 21
}


def keys(dictionary):
    return [k for k in dictionary]


print(str(keys(my_dict)))
```

It's pretty simple, right? In human language, that iterates for every key in the given dictionary and returns an array with them. The output:

```
['name', 'age']
```

6.4.2 Len (built-in function)

The `len` function returns the number of key-value pairs in a dictionary. It's data types do not matter in this context. Notice the fact that it returns the number of **key-value pairs**, so a dictionary that looks like `{"key", "value"}` will return 1 when asked for its length.

Its use is pretty simple, it receives the dictionary in question as argument, and just for testing purposes we'll modify the dict_all_keys.py to check that the number of keys its returning is the same as the len of the dict.

dict_all_keys.py

```
my_dict = {
    "name": "Sebastian",
    "age": 21
}


def keys(dictionary):
    result = [k for k in dictionary]
    # result.append("break the program plz")
    if len(result) != len(dictionary):
        raise Exception('expected {} keys. got {}'.format(len(dictionary), len(result)))
    return result


print(str(keys(my_dict)))
```

There is the check, if we uncomment the second line of the function's body we'll see the exception raised:

```
Traceback (most recent call last):
  File "/home/svinci/projects/jcg/dictionaries/dict_all_keys.py", line 15, in <module<
    print(str(keys(my_dict)))
  File "/home/svinci/projects/jcg/dictionaries/dict_all_keys.py", line 11, in keys
    raise Exception('expected {} keys. got {}'.format(len(dictionary), len(result)))
Exception: expected 2 keys. got 3
```

The message **Exception: expected 2 keys. got 3** is telling us that the keys we are about to return contain an extra key. This is another useless but explicit example.

6.4.3 keys() and values()

Let's see a couple functions that render obsolete the function keys we just wrote. These are `keys()` and `values()`. I think it's pretty intuitive what these functions do, **keys** returns every key in the dictionary and **values** returns every value in the dictionary.

Having a dictionary like `me ={"name":"Sebastian", "age":21},me.keys()` will return `["name", "age"]` and `me.values()` will return `["Sebastian", 21]`. Kind of... see, dictionaries in python are not ordered, then, the result of `keys` and `values` aren't either. If you need to sort either keys or values just call `sorted` passing the array as argument.

Let's see an example:

dict_keys_values.py

```
def print_dict(dictionary):
    ks = list(dictionary.keys())
    vs = list(dictionary.values())
    for i in range(0, len(ks)):
        k = ks[i]
        v = vs[i]
        print("{}: {}".format(str(k), str(v)))


example = {
    "name": "Sebastian",
    "last_name": "Vinci",
    "age": 21
}

print_dict(example)
```

Here we are creating a dictionary containing some keys, and then iterating over the keys and values to print them out. I know it's not the prettiest code but it proves my point. When we execute it one time we see:

```
age: 21
name: Sebastian
last_name: Vinci
```

Then a second time:

```
last_name: Vinci
age: 21
name: Sebastian
```

See? That's because dictionaries' keys are not sorted, to solve this issue (and fix that awful piece of code) let's change that script a little bit. We won't be using the function `values` anymore since it's not necessary, but I think it's pretty clear the way to use it.

dict_keys_values.py

```
def print_dict(dictionary):
    ks = sorted(dictionary.keys())
    for k in ks:
        print("{}: {}".format(str(k), str(dictionary.get(k))))


example = {
    "name": "Sebastian",
    "last_name": "Vinci",
    "age": 21
}

print_dict(example)
```

Now we'll always see the same output, as the keys are being sorted alphabetically. The output:

```
age: 21
last_name: Vinci
name: Sebastian
```

Now, why am I giving so much importance to the lack of sorting of dictionaries? Well, imagine you are storing it in a CSV file. Every time you write a dictionary, as the keys are in arbitrary orders, the CSV's rows won't be consistent, and it will mess up your program by writing each row with the columns in different orders. So when you need your data in a strict order, please keep in mind this little fact.

6.4.4 items()

This method returns a list of tuples containing every key-value pair in the dictionary as `({"a":1, "b":2}.items() => [("a", 1), ("b", 2)])`. With this method we can iterate over this pair array more seamlessly (keep in mind that this is still unordered), so if we just want to print our dictionary and the order doesn't matter we can write:

dict_items.py

```
def print_dict(dictionary):
    for k, v in dictionary.items():
        print("{}: {}".format(str(k), str(v)))


example = {
    "name": "Sebastian",
    "last_name": "Vinci",
    "age": 21
}

print_dict(example)
```

This will output the same as the first dict_keys_values.py, "key: value" with different orders in each execution. Something to notice is the fact that we can not assign new values to tuples. We access values in tuples by `tuple[0]` and `tuple[1]`, but we can't do `tuepl[0] =1`. This will rase a TypeError: *tuple* object does not support item assignment.

6.4.5 update()

This method changes one dictionary to have new values from a second one. It modifies existing values, as dictionaries are mutable (keep this in mind). Let's see an example:

dict_update.py

```
product = {
    "description": "WCG E-Book",
    "price": 2.75,
    "sold": 1500,
    "stock": 5700
}


def sell(p):
    update = {"sold": p.get("sold", 0) + 1, "stock": p.get("stock") - 1}
    p.update(update)


def print_product(p):
    print(", ".join(["{}: {}".format(str(k), str(product[k])) for k in sorted(p.keys())]))

print_product(product)
for i in range(0, 100):
    sell(product)
print_product(product)
```

Here we are printing the product, executing `sell` a hundred times and then printing the product again. The sell function updates de product subtracting one from "stock" and adding one to "sold". The output:

```
description: WCG E-Book, price: 2.75, sold: 1500, stock: 5700
description: WCG E-Book, price: 2.75, sold: 1600, stock: 5600
```

Notice that we don't return a new dictionary with the updated values, it updates the existing one. This is an easy way of updating a couple keys in a dictionary without having to overwrite them one by one.

6.4.6 copy()

A dictionary's copy method will copy the entire dictionary into a new one. One thing to notice is that it's a shallow copy. If you have nested dictionaries, it will copy the first level, but the inner dictionaries will be a reference to the same object. Let's see an example:

dict_copy.py

```
me = {
    "name": {
        "first": "Sebastian",
        "last": "Vinci"
    },
    "age": 21
}
my_clone = me.copy()

my_clone["age"] = 22
print("my age: {}, my clone's age: {}".format(me.get("age"), my_clone.get("age")))
```

```
my_clone.get("name")["first"] = "Michael"
print("my name: {}, my clone's name: {}".format(me.get("name").get("first"), my_clone.get(" ↩
    name").get("first")))
```

In this script, we are creating a dictionary with some data and then copying it. We change the clone's age and print both, and then do the same with the first names.

Now, my clone is a copy of me, it is not a reference to the same memory address, so if I change its age, mine will remain the same. But, the inner dictionary, the one containing the name, is a reference to the same memory address, when I change my clone's name, mine will change too. Let's see the output:

```
my age: 21, my clone's age: 22
my name: Michael, my clone's name: Michael
```

So, having this in mind, we can use the `copy`, which, if used carefully, is a powerful and useful tool.

6.5 Download the Code Project

This was an basic example on Python Dictionaries.

Download You can download the full source code of this example here: **dictionaries**

Chapter 7

Sockets Example

In this tutorial, we'll talk about INET, STREAM sockets. We'll talk about what sockets are, and learn how to work with blocking and non-blocking sockets.

First things first, we'll make a distinction between a "client" socket (endpoint of a conversation), and a "server" socket (switchboard operator). Your client (e.g. your browser) uses only client sockets, and your server uses both client and server sockets.

7.1 Creating a Socket

To open a socket, you only need to import the module `socket` and do as follows:

client.py

```
import socket

s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(("www.webcodegeeks.com", 80))
```

When the `connect` is completed, `s` can be used to send in a request for the text of the page. It will read the reply and then be destroyed. Yes, destroyed. Client sockets are only used for one exchange (or a set of sequential exchanges).

In the server, a server socket is created to accept this connection:

server.py

```
import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind((socket.gethostname(), 80))
serversocket.listen(5)
```

Here, we are creating an INET, STREAM socket and binding it to our host at the 80 port. Notice the use of `socket.gethostname()`, as this is how you make your socket visible to the outer world. If you bind it to localhost or 127.0.0.1, this will still be a server socket, but it only will be visible within the same machine. If you bind it to ' ', the socket will be reachable by any address the machine happens to have.

Also, low number ports are used by system's services (HTTP, SMTP, etc.), if you want your socket and these services to work at the same time, you should stick to high number ports (4 digits)(8080, 9290, 9000, 8000, etc.).

Now, `serversocket.listen`, tells the socket how many connect requests should be queued before refusing outside connections. In other words, the argument to this method is the maximum active connections it will allow at the same time.

Using a while loop, we will start listening to connections:

server.py

```
import socket

serversocket = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
serversocket.bind((socket.gethostname(), 80))
serversocket.listen(5)


def something(clientsocket):
    # Do something with the socket
    return None

while True:
    (clientsocket, address) = serversocket.accept()
    something(clientsocket)
```

We defined a function that receives the socket, and then, within the while loop, we accept connections and send them to our function.

There are a couple of interesting things to notice there, though, like the fact that the server socket doesn't actually send or receive data. It just produces client sockets which will communicate, through a dynamically assigned port which will be recycled after the exchange is done, with the **actual client socket that connected to our server**.

Another thing to have in mind here, is that `something` should be an asynchronous function. If the execution within the while loop blocks the main thread, you will see your response time next to a plane 9000 feet over the ground.

7.2 Using a Socket

Your client's client socket and your web server's client socket are identical, this is a peer to peer (P2P) conversation. The rules for this conversation are only defined by you as the designer. Normally the client starts this conversation by sending a request, but there are no rules for sockets.

There are two verbs to use for communication, `send` and `recv`, these operate on the network buffers. You hand bytes to them, but they won't necessarily handle them all, as their major focus is the network buffer. They return when the associated network buffer has been filled (`send`) or emptied (`recv`), then they will tell you how many bytes they handled. It is **your** responsibility to call them again until your message has been completely dealt with.

When a `recv` returns 0 bytes, it means the other side has closed the connection. You will not receive any more data through this connection, although you may be able to send data successfully. But if you plan to reuse your socket for further transfers, you need to realize that sockets **will not** tell you that there is nothing more to read. Messages must either:

- Be **fixed length**: Not the best option.
- Be **delimited**: Even worse.
- **Indicate how long they are**: Now this is a solution.
- End by **shutting down the connection**: This might be a little violent.

As I said before, there are no rules with sockets, so you can choose any of these options to know when to stop reading from a socket. Although there are some ways that are better than others.

Discarding the fourth option (shutting down the connection), the simplest way of doing this is a fixed length message:

fixed_length_socket.py

```
import socket


class FixedLengthSocket:
```

```
    def __init__(self, message_len, sock=None):
        if sock is None:
            self.sock = socket.socket(
                            socket.AF_INET, socket.SOCK_STREAM)
        else:
            self.sock = sock
        self.message_len = message_len

    def connect(self, host, port):
        self.sock.connect((host, port))

    def send(self, msg):
        total_sent = 0
        while total_sent < self.message_len:
            sent = self.sock.send(msg[total_sent:])
            if sent == 0:
                raise RuntimeError("socket connection broken")
            total_sent += sent

    def receive(self):
        chunks = []
        bytes_recd = 0
        while bytes_recd < self.message_len:
            chunk = self.sock.recv(min(self.message_len - bytes_recd, 2048))
            if chunk == b'':
                raise RuntimeError("socket connection broken")
            chunks.append(chunk)
            bytes_recd += len(chunk)
        return b''.join(chunks)
```

7.3 Disconnecting

There is a method called `shutdown` that you are supposed to call before you actually close it. It is a message to the socket at the other end, which content depends on the flag you send as argument, where `1` is I won't send anything else, but I'm still listening and `2` is Nope! I'm not listening anymore. Bye.

In Python, when a socket is garbage collected it will perform a shutdown if it's needed, but you shouldn't rely on this. A socket disappearing without performing a shutdown before, will cause the socket at the other end to hang indefinitely.

Now, if you are on the other side, you'll see that the worst thing of working with sockets is when the other end just disappears without notifying you. Your socket will hang, as TCP is a reliable protocol and it will wait for a long time before giving up a connection.

In this case, if you are using threads, your thread is essentially dead, you can't do anything about it. But, there is a bright side here: If you are not doing anything stupid, like holding a lock while doing a blocking read, the resources consumed by the thread are not something you should worry about.

Remember not to try and kill a thread, as they don't automatically recycle resources. If you do manage to kill the thread, your whole process is likely to be screwed up.

7.4 A Little Example Here

To see these threads at work, we will create a sample program which receives incoming messages and echos them back to the sender.

server.py

```
import socket

# Create a TCP/IP socket
sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
server_address = ('localhost', 10000)
print('starting up on %s port %s' % server_address)
sock.bind(server_address)
sock.listen(1)

while True:
    print('waiting for a connection')
    connection, client_address = sock.accept()
    try:
        print('connection from', client_address)
        while True:
            data = connection.recv(16)
            print('received "%s"' % data)
            if data:
                print('sending data back to the client')
                connection.sendall(data)
            else:
                print('no more data from', client_address)
                break
    finally:
        connection.shutdown(2)
        connection.close()
```

Here, we are creating a socket and binding it to localhost:10000 (this will only by available to programs running on the same machine), then we listen, at most, 1 connections at a time.

In the while loop, as the operation is fairly easy and fast, we are directly receiving and sending back the data with, as you can see, a fixed length.

Now, let's write the client program:

client.py

```
import socket
import sys

sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)

server_address = ('localhost', 10000)
print('connecting to %s port %s' % server_address)
sock.connect(server_address)

try:
    message = b'This is the message.  It will be repeated.'
    print('sending "%s"' % message)
    sock.sendall(message)
    amount_received = 0
    amount_expected = len(message)
    while amount_received < amount_expected:
        data = sock.recv(16)
        amount_received += len(data)
        print('received "%s"' % data)

finally:
    print(sys.stderr, 'closing socket')
    sock.shutdown(2)
    sock.close()
```

Here we are creating a socket, but we don't bind to anything, as this is a client socket. We connect to our server address and then start sending and receiving data.

You can see the output:

server.py output

```
starting up on localhost port 10000
waiting for a connection
connection from ('127.0.0.1', 52600)
received "b'This is the mess'"
sending data back to the client
received "b'age.  It will be'"
sending data back to the client
received "b' repeated.'"
sending data back to the client
received "b''"
no more data from ('127.0.0.1', 52600)
waiting for a connection
```

client.py output

```
connecting to localhost port 10000
sending "b'This is the message.  It will be repeated.'"
received "b'This is the mess'"
received "b'age.  It will be'"
received "b' repeated.'"
<_io.TextIOWrapper name='<stderr>' mode='w' encoding='UTF-8'> closing socket
```

There you can see the behaviour we talked about. They won't process the entire message at once, they will process the data by chunks and you have to call as long as there is still data to process.

7.5 Non-blocking Sockets

If you reached this point, you now know most of what you need about sockets. With non-blocking sockets, you'll use the same calls in the same ways.

In Python, to make a socket non-blocking you need to call `socket.setblocking(0)`. You should do this after you create the socket, but before you use it (Notice the **should**).

The major difference between blocking and non-blocking sockets is that `send`, `recv`, `connect` and `accept` can return without having done anything. To work with this, se best solution is to use `select`.

```
ready_to_read, ready_to_write, in_error = select.select(
                  potential_readers,
                  potential_writers,
                  potential_errs,
                  timeout)
```

You pass select three lists: the first contains all sockets that you might want to try reading; the second all the sockets you might want to try writing to, and the last (normally left empty) those that you want to check for errors. You should note that a socket can go into more than one list. The select call is blocking, but you can give it a timeout. This is generally a sensible thing to do - give it a nice long timeout (say a minute) unless you have good reason to do otherwise.

In return, you will get three lists. They contain the sockets that are actually readable, writable and in error. Each of these lists is a subset (possibly empty) of the corresponding list you passed in.

If you have a "server" socket, put it in the potential_readers list. If it comes out in the readable list, your accept will (almost certainly) work. If you have created a new socket to connect to someone else, put it in the potential_writers list. If it shows up in the writable list, you have a decent chance that it has connected.

7.6 Download the Code Project

This was a basic example on sockets with Python.

Download You can download the full source code of this example here: **python-sockets**

Chapter 8

Map Example

In this tutorial, by using python 3.4, we'll talk about map functions.

Strictly speaking, a `map` function, is a function that accepts a function and an array as arguments, and applies the given function to each element of the array, returning another array with the result of each transformation.

So, in other words, a map applies a transformation to each element of an array and returns the results.

How would we implement it? Let's see a simple implementation to understand how a map works:

8.1 Map Implementation

custom_map.py

```
def custom_map(transformation, array):
    return [transformation(e) for e in array]
```

So, we defined a function that receives a transformation and an array as arguments, and using for comprehension it returns an array of the transformations of each of these elements. To use it:

custom_map.py

```
def custom_map(transformation, array):
    return [transformation(e) for e in array]


def add_one(x): return x + 1


def test():
    my_array = [1, 2, 3, 4, 5]
    print(str(custom_map(add_one, my_array)))

if __name__ == "__main__":
    test()
```

As functions are first class objects in python, we can define a function called `add_one` and pass it as argument to our `custom_map` with an array containing from 1 to 5. The output, as we expected, is:

```
[2, 3, 4, 5, 6]
```

8.2 Python's Map

Now, Python provides a built-in `map` function, which behaves almost like our custom map. The differences:

- It does not return a list, but a `map object`, but if you apply `list` to it, will become a list.
- It does not accept only one array, but n instead, passing n arguments to the provided transformation, applying it to the items of the provided arrays in parallel (returning None as default if an array is shorter than others) like: `transformation(array 1[0], array2[0], ..., arrayN[0])`.

So, let's see an example of this:

multiply.py

```
def multiply(x, y):
    return x * y


def test():
    xs = [1, 2, 3, 4, 5, 6, 7, 8]
    ys = [2, 3, 4, 5, 6, 7]
    map_object = map(multiply, xs, ys)
    result_list = list(map_object)
    print(str(result_list))


if __name__ == "__main__":
    test()
```

Here, we defined an array `xs` and another one `ys` and `ys` is shorter than `xs`. The result will look like:

```
[2, 6, 12, 20, 30, 42]
```

As you see, the result contains as many arguments as the shorter array, as there are not enough arguments to pass to our transformation function (`multiply`).

Also, notice we apply the function `list` to the result, instead of printing it directly, as the result of a map function is not a list, but a `map object` as we said before.

In real life, we often don't want to define a function to pass it to a map, as it probably will only be used there. Lambdas come to help. Let's refactor our multiply to see this:

multiply.py

```
def test():
    xs = [1, 2, 3, 4, 5, 6, 7, 8]
    ys = [2, 3, 4, 5, 6, 7]
    map_object = map(lambda x, y: x * y, xs, ys)
    result_list = list(map_object)
    print(str(result_list))


if __name__ == "__main__":
    test()
```

The output will look the same, and our function is now defined right there, one the run, so we don't have to worry about someone reusing it if they are not meant to.

Let's see a real life case scenario in which map comes to save the day. Imagine a scenario in which you have a catalog of products and for each product a history of prices that looks like this:

example.py

```
from datetime import datetime

products = {
    1: {
        "id": 1,
        "name": "Python Book"
    },
    2: {
        "id": 2,
        "name": "JavaScript Book"
    },
    3: {
        "id": 3,
        "name": "HTML Book"
    }
}
prices = {
    1: [
        {
            "id": 1,
            "product_id": 1,
            "amount": 2.75,
            "date": datetime(2016, 2, 1, 11, 11, 17, 683987)
        },
        {
            "id": 4,
            "product_id": 1,
            "amount": 3.99,
            "date": datetime(2016, 2, 5, 11, 11, 17, 683987)
        }
    ],
    2: [
        {
            "id": 2,
            "product_id": 2,
            "amount": 1.10,
            "date": datetime(2015, 1, 5, 11, 11, 17, 683987)
        },
        {
            "id": 2,
            "product_id": 2,
            "amount": 1.99,
            "date": datetime(2015, 12, 5, 11, 11, 17, 683987)
        }
    ],
    3: [
        {
            "id": 3,
            "product_id": 3,
            "amount": 3,
            "date": datetime(2015, 12, 20, 11, 11, 17, 683987)
        }
    ]
}
```

When you are going to show these products to the public, you want to attach to each of them the last updated price. So, given this issue, a function which does attach the last price to a product looks like this:

example.py

```
...
```

```
def complete_product(product):
    if product is None:
        return None
    product_prices = prices.get(product.get("id"))
    if product_prices is None:
        return None
    else:
        p = product.copy()
        p["price"] = max(product_prices, key=lambda price: price.get("date")).copy()
        return p
```

This function receives a product as argument, and if it's not None and there are prices for this product, it returns a copy of the product with a copy of its last price attached. The functions that use this one looks like this:

example.py

```
...
def get_product(product_id):
    return complete_product(products.get(product_id))


def many_products(ids):
    return list(map(get_product, ids))


def all_products():
    return list(map(complete_product, products.values()))
```

The first one, `get_product`, is pretty simple, it just reads one product and returns the result of applying `complete_product` to it. The other ones are the functions that are of interest to us.

The second function assumes, without any validation, that `ids` is an iterable of integers. It applies a map to it passing the function `get_product` as transformation, and with the resulting map object, it creates a list.

The third function receives no parameters, it just applies a map to every product, passing `complete_product` as transformation.

Another example of a use case of `map` would be when we need to discard information. Imagine a service that provides all users in an authentication system. Of course, the passwords would be hashed, therefor, not readable, but still it would be a bad idea to return those hashed with the users. Let's see:

example2.py

```
users = [
    {
        "name": "svinci",
        "salt": "098u n4v04",
        "hashed_password": "q423uinf9304fh2u4nf3410uth1394hf"
    },
    {
        "name": "admin",
        "salt": "0198234nva",
        "hashed_password": "3894tumn13498ujc843jmcv92384vmqv"
    }
]


def all_users():
    def discard_passwords(user):
        u = user.copy()
        del u["salt"]
        del u["hashed_password"]
        return u
```

```
    return list(map(discard_passwords, users))


print(str(all_users()))
```

In `all_users` we are applying `discard_passwords` to all users an creating a list with the result. The transformation function receives a user, an returns a copy of it with the salt and hashed_passwords removed.

8.3 Map Object

Now, we've been transforming this map object to a list, which is not wrong at all, it's the most common use case, but there is another way.

A map object is an iterator that applies the transformation function to every item of the provided iterable, yielding the results. This is a neat solution, which optimizes a lot memory consumption, as the transformation is applied only when the element is requested.

Also, as the map objects are iterators, we can use them directly. Let's see the full example and refactor it a bit to use map objects instead of lists.

example.py

```
def complete_product(product):
    if product is None:
        return None
    product_prices = prices.get(product.get("id"))
    if product_prices is None:
        return None
    else:
        p = product.copy()
        p["price"] = max(product_prices, key=lambda price: price.get("date")).copy()
        return p


def get_product(product_id):
    return complete_product(products.get(product_id))


def many_products(ids):
    return map(get_product, ids)


def all_products():
    return map(complete_product, products.values())


if __name__ == "__main__":
    print("printing all...")
    ps = all_products()
    for p in ps:
        print(str(p))
    print("printing all, but by all ids...")
    ids = map(lambda p: p.get("id"), all_products())
    ps = many_products(ids)
    for p in ps:
        print(str(p))
```

The functions that changed are `many_products` and `all_products`, which now retrieve the map object instead of a list.

We are retrieving all products, and iterating through the map object printing each product. Notice that map objects, as they are iterators, can be iterated only once, so, we are retrieving all once again to get the ids and then retrieving by its ids and printing all again. The output below.

```
printing all...
{'id': 1, 'name': 'Python Book', 'price': {'id': 4, 'product_id': 1, 'amount': 3.99, 'date' ↩
    : datetime.datetime(2016, 2, 5, 11, 11, 17, 683987)}}
{'id': 2, 'name': 'JavaScript Book', 'price': {'id': 2, 'product_id': 2, 'amount': 1.99, ' ↩
    date': datetime.datetime(2015, 12, 5, 11, 11, 17, 683987)}}
{'id': 3, 'name': 'HTML Book', 'price': {'id': 3, 'product_id': 3, 'amount': 3, 'date':  ↩
    datetime.datetime(2015, 12, 20, 11, 11, 17, 683987)}}
printing all, but by all ids...
{'id': 1, 'name': 'Python Book', 'price': {'id': 4, 'product_id': 1, 'amount': 3.99, 'date' ↩
    : datetime.datetime(2016, 2, 5, 11, 11, 17, 683987)}}
{'id': 2, 'name': 'JavaScript Book', 'price': {'id': 2, 'product_id': 2, 'amount': 1.99, ' ↩
    date': datetime.datetime(2015, 12, 5, 11, 11, 17, 683987)}}
{'id': 3, 'name': 'HTML Book', 'price': {'id': 3, 'product_id': 3, 'amount': 3, 'date':  ↩
    datetime.datetime(2015, 12, 20, 11, 11, 17, 683987)}}
```

In the users example the code will look like this:

```
users = [
    {
        "name": "svinci",
        "salt": "098u n4v04",
        "hashed_password": "q423uinf9304fh2u4nf3410uth1394hf"
    },
    {
        "name": "admin",
        "salt": "0198234nva",
        "hashed_password": "3894tumn13498ujc843jmcv92384vmqv"
    }
]


def all_users():
    def discard_passwords(user):
        u = user.copy()
        del u["salt"]
        del u["hashed_password"]
        return u
    return map(discard_passwords, users)


if __name__ == "__main__":
    us = all_users()
    for u in us:
        print(str(u))
```

Again, the `all_users` function didn't change too much, it just returns the map object directly. The main code now iterates over the map printing each user one by one.

8.4 Download the Code Project

This was an basic example on Python's map function.

Download You can download the full source code of this example here: **python-map**

Chapter 9

Subprocess Example

In this article, using Python 3.4, we'll learn about Python's `subprocess` module, which provides a high-level interface to interact with external commands.

This module is intended to be a replacement to the old `os.system` and such modules. It provides all of the functionality of the other modules and functions it replaces. The API is consistent for all uses, and operations such as closing files and pipes are built in instead of being handled by the application code separately.

One thing to notice is that the API is roughly the same, but the underlying implementation is slightly different between Unix and Windows. The examples provided here were tested on a Unix based system. Behavior on a non-Unix OS will vary.

The `subprocess` module defines a class called `Popen` and some wrapper functions which take almost the same arguments as its constructor does. But for more advanced use cases, the `Popen` interface can be used directly.

We'll first take a look to all the wrapper functions and see what they do and how to call them. Then we'll see how to use `Popen` directly to ease our thirst for knowledge.

9.1 Convenience Functions

9.1.1 subprocess.call

The first function we'll overview is `subprocess.call`. The most commonly used arguments it takes are `(args, *, stdin=None, stdout=None, stderr=None, shell=False, timeout=None)`. Although these are the most commonly used arguments, this function takes roughly as many arguments as `Popen` does, and it passes almost them all directly through to that interface. The only exception is the `timeout` which is passed to `Popen.wait()`, if the timeout expires, the child process will be killed and then waited for again. The TimeoutExpired exception will be re-raised after the child process has terminated.

This function runs the command described by `args`, waits for the command to complete and returns the exit code attribute. Let's see a trivial example:

```
$ python3
>>> import subprocess
>>> subprocess.call(["ls", "-l"])
0
```

As you see, it's returning 0, as it's the return code of the `ls -l` command. You will probably see the output of the command in the console as it's sent to standard output, but it won't be available within your code using this command this way.

Another thing to notice is that the command to run is being passed as an array, as the `Popen` interface will then concatenate and process the elements saving us the trouble of escaping quote marks and other special characters. You can avoid this notation by setting the `shell` flag to a `True` value, which will spawn an intermediate shell process and tell it to run the command, like this:

```
$ python3
>>> import subprocess
>>> subprocess.call("ls -l", shell=True)
0
```

Although the `shell=True` option might be useful to better exploit the most powerful tools the shell provides us, it can be a security risk, and you should keep this in mind. To use this tool in a safer way you can use `shlex.quote()` to properly escape whitespace and shell metacharacters in strings that are going to be used to construct shell commands. Let's see and example of a security issue this can cause:

security-issues.py

```
import subprocess
import shlex


def unsafe(directory):
    command = "du -sh {}".format(directory)
    subprocess.call(command, shell=True)


def safe(directory):
    sanitized_directory = shlex.quote(directory)
    command = "du -sh {}".format(sanitized_directory)
    subprocess.call(command, shell=True)

if __name__ == "__main__":
    directory = "/dev/null; ls -l /tmp"
    unsafe(directory)
    safe(directory)
```

Here we have two functions. The first one, `unsafe`, will run a `du -sh` over a directory provided, without validating or cleaning it. The second one, `safe`, will apply the same command to a sanitized version of the provided directory. By providing a directory like `"/dev/null;ls -l /tmp"`, we are trying to exploit the vulnerability to see what the contents of `/tmp` are, of course it can get a lot uglier if a `rm` would intervene here.

The `unsafe` function will output to stdout a line like `0 /dev/null` as the output of the `du -sh`, and then the contents of `/tmp`. The `safe` method will fail, as no directory called `"/dev/null;ls -l /tmp"` exists in the system.

Now, as we said before, this function returns the exit code of the executed command, which give us the responsibility of interpreting it as a successful or an error code. Here comes the `check_call` function.

9.1.2 subprocess.check_call

This wrapper function takes the same arguments as `subprocess.call`, and behaves almost the same. The only difference is that it interprets the exit code for us.

It runs the described command, waits for it to complete and, if the return code is zero, it returns, otherwise it will raise `CalledProcessError`. Let's see:

```
$ python3
>>> import subprocess
>>> subprocess.check_call("exit 1", shell=True)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/usr/lib/python3.4/subprocess.py", line 561, in check_call
    raise CalledProcessError(retcode, cmd)
subprocess.CalledProcessError: Command 'exit 1' returned non-zero exit status 1
```

The command `exit` will ask the shell to finish with the given exit code. As it is not zero, `CalledProcessError` is raised.

One way of gaining access to the output of the executed command would be to use `PIPE` in the arguments `stdout` or `stderr`, but the child process will block if it generates enough output to a pipe to fill up the OS pipe buffer as the pipes are not being read from. So, having said that this method is not safe, let's skip to the safe and easy way.

9.1.3 subprocess.check_output

This function receives almost the same arguments as the previous ones, but it adds `input` and `universal_newlines`. The `input` argument is passed to `Popen.communicate()` and thus to the command stdin. When it is uses, it should be a byte sequence, or a string if `universal_newlines=True`.

Another change is in its behaviour, as it returns the output of the command. It will check the exit code of the command for you raising `CalledProcessError` if it's not 0. Let's see an example:

```
$ python3
>>> import subprocess
>>> subprocess.check_output(["echo", "Hello world!"])
b'Hello world!\n'
>>> subprocess.check_output(["echo", "Hello World!"], universal_newlines=True)
'Hello World!\n'
>>> subprocess.check_output(["sed", "-e", "s/foo/bar/"], input=b"when in the course of  ↩
    fooman events\n")
b'when in the course of barman events\n'
>>> subprocess.check_output(["sed", "-e", "s/foo/bar/"], input="when in the course of  ↩
    fooman events\n", universal_newlines=True)
'when in the course of barman events\n'
```

Here you can see the variations of the two new arguments. The `input` argument is being passed to the `sed` command in the third and fourth statements, and the `universal_newlines` decodes the byte sequences into strings or not, depending on its value.

Also, you can capture the standard error output by setting `stderr=subprocess.STDOUT`, but keep in mind that this will merge stdout and stderr, so you really need to know how to parse all that information into something you can use.

9.2 Popen

Now, let's see how can we use `Popen` directly. I think with the explanation made before, you get an idea. Let's see a couple of trivial use cases. We'll start by rewriting the already written examples using `Popen` directly.

```
$ python3
>>> from subprocess import Popen
>>> proc = Popen(["ls", "-l"])
>>> proc.wait()
0
```

Here, as your see, `Popen` is being instantiated passing only the command to run as an array of strings (just as we did before). `Popen`'s constructor also accepts the argument `shell`, but it's still unsafe. This constructor starts the command execution the moment it is instantiated, but calling `Popen.wait()` will wait for the process to finish and return the exit code.

Now, let's see an example of `shell=True` just to see that it's just the same:

```
$ python3
>>> from subprocess import Popen
>>> proc = Popen("ls -l", shell=True)
>>> proc.wait()
0
```

See? It behaves the same as the wrapper functions we saw before. It doesn't seem so complicated now, does it? Let's rewrite the `security-issues.py` example with it:

security-issues-popen.py

```
from subprocess import Popen
import shlex


def unsafe(directory):
    command = "du -sh {}".format(directory)
    proc = Popen(command, shell=True)
    proc.wait()


def safe(directory):
    sanitized_directory = shlex.quote(directory)
    command = "du -sh {}".format(sanitized_directory)
    proc = Popen(command, shell=True)
    proc.wait()

if __name__ == "__main__":
    directory = "/dev/null; ls -l /tmp"
    unsafe(directory)
    safe(directory)
```

Now, if we execute this example, we'll see it behaves the same as the one written with the `subprocess.call`.

Now, `Popen`'s constructor will not raise any errors if the command exit code is not zero, its `wait` function will return the exit code and, again, it's your responsibility to check it and perform any fallback.

So, what if we want to gain access to the output? Well, `Popen` will receive the `stdout` and `stderr` arguments, which now are safe to use, as we are going to read from them. Let's write a little example which let's us see the size of a file/directory.

size.py

```
from subprocess import Popen, PIPE


def command(directory):
    return ["du", "-sh", directory]


def check_size(directory):
    proc = Popen(command(directory), stdout=PIPE, stderr=PIPE, universal_newlines=True)
    result = ""
    exit_code = proc.wait()
    if exit_code != 0:
        for line in proc.stderr:
            result = result + line
    else:
        for line in proc.stdout:
            result = result + line
    return result


def main():
    arg = input("directory to check: ")
    result = check_size(directory=arg)
    print(result)

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

Here we are instantiating a `Popen` and setting `stdout` and `stderr` to `subprocess.PIPE` to be able to read the output of our command, we are also setting `universal_newline` to `True` so we can work with strings. Then we are calling `Popen`.

`wait()` and saving the exit code (notice that you can pass a timeout to the `wait` function) and, then printing the standard output of our command if it ran successfully, or the standard error output if else.

Now, the last case scenario we are missing is sending stuff to the standard input of our command. Let's write a script that interacts with our `size.py` sending a hardcoded directory to its stdin:

size-of-tmp.py

```
from subprocess import Popen, PIPE


def size_of_tmp():
    proc = Popen(["python3", "size.py"], stdin=PIPE, stdout=PIPE, stderr=PIPE,  ↩
        universal_newlines=True)
    (stdout, stderr) = proc.communicate("/tmp")
    exit_code = proc.wait()
    if exit_code != 0:
        return stderr
    else:
        return stdout

if __name__ == "__main__":
    print(size_of_tmp())
```

Now, this last script will execute the first one, and as it asks for a directory to check, the new script will send `"/tmp"` to its standard input.

9.3 Download the Code Project

This was a basic example on Python subprocesses.

Download You can download the full source code of this example here: **python-subprocess**

Chapter 10

Send Email Example

In this example, by using Python 3.4, we'll learn how to send mails using Python's `smtplib` module.

SMTP stands for Simple Mail Transfer Protocol. It's a protocol which handles sending e-mails and routing them between mail servers.

To send a mail, we need the host and port of a server, then we send to this server the message with a sender and a list of receivers.

The message is not just any string. This protocol expects the message in a certain format, which defines three headers (**From**, **To** and **Subject**) and the actual message, these separated with a blank line as in:

```
message = """From: Example Sender <sender@example.com>
To: Example Receiver <receiver@example.com>
Subject: Example Message

This is an example message. Sorry if you receive this by mistake."""
```

10.1 The Basics of smtplib

The `smtplib` module provides an SMTP client session object that can be used to send mails to any Internet machine with an SMTP or ESMTP listener daemon. Its constructor receives five optional parameters:

- **host**: This is the host running the SMTP server. You can specifiy IP address of the host or a domain name like web-codegeeks.com.
- **port**: When you provide a host argument, you need to specify the port your server is listening.
- **local_hostname**: If your SMTP server is running locally, you can specify "localhost" here.
- **timeout**: Specifies a timeout in seconds for blocking operations like the connection attempt.
- **source_address**: Allows to bind to some specific source address in a machine with multiple network interfaces, and/or to some specific source TCP port.

An instance of this object encapsulates an SMTP connection. If the optional host and port parameters are given, the SMTP `connect()` method is called with those parameters during initialization. If specified, `local_hostname` is used as the FQDN of the local host in the HELO/EHLO command. Otherwise, the local hostname is found using `socket.getfqdn()`. If the `connect()` call returns anything other than a success code, an `SMTPConnectError` is raised. If the timeout expires, `socket.timeout` is raised. The optional `source_address` parameter takes a 2-tuple (host, port), for the socket to bind to as its source address before connecting. If omitted (or if host or port are `''` and/or `0` respectively) the OS default behavior will be used.

Once this session object is created, it will provide a function called `sendmail` which receives three arguments (**sender**, **receivers** and **message**). `sender` is the e-mail address from which this mail will be sent, `receivers` is an array of e-mail addresses which will be the recipients, and `message` is the body of the mail, it should be formatted as we talked before.

Let's see an example of how to apply the knowledge we acquired so far.

simple.py

```
import smtplib

sender = "sender@example.com"
receivers = ["receiver1@example.com", "receiver2@example.com"]


def format_mail(mail):
    return "{} ".format(mail.split("@")[0], mail)

message = """From: {}
To: {}
Subject: Example Subject

This is a test mail example
""".format("{} ".format(sender.split("@")[0], sender), ", ".join(map(format_mail, receivers ↩
    )))

try:
    print("sending message: " + message)
    with smtplib.SMTP('example-smpt.com', 25) as session:
        session.sendmail(sender, receivers, message)
    print("message sent")
except smtplib.SMTPException:
    print("could not send mail")
```

Of course, the SMPT server defined in this example does not exist, so this example won't actually work. In fact, as we need passwords to send mails (we'll see this below), none of the examples in this article will actually work, unless you replace the credentials in them with ones of your own.

Having said that, let's talk about this simple example. It's just a really simple example which defines a sender, an array of recipients and a message (formatted as it should be, and dynamically inserting the addresses). Then it creates a session and calls `sendmail`. Notice the use of the `with` statement, when used like this, the SMTP `QUIT` command is issued automatically when the `with` statement exits.

10.2 SSL and Authentication

Let's talk bout login. Usually we send a mail from an account to which we have access, and it's secured (most of the times), so authentication will be required. Also, connections to an SMTP server should be done via SSL from the beginning of the connection. In the next example we'll see both login and SSL connections procedures.

Python provides a class called `SMTP_SSL`, which behaves exactly like `SMTP`. It's constructor receives almost the same arguments. If the host is not specified, it will use the local host instead. If port is zero, it will use the default for SSL SMTP connections, which is 465. The optional arguments local_hostname, timeout and source_address have the same meaning as they do in the `SMTP` class. There are also three more arguments. The first one is `context`, which is optional, and can contain a SSLContext and allows to configure various aspects of the secure connection. The other ones are `keyfile` and `certfile`, which are a legacy alternative to context, and can point to a PEM formatted private key and certificate chain file for the SSL connection.

Now, both `SMTP` and `SMTP_SSL` provide a function called `login` which receives a user and a password, and is our way through authentication. The next example will show how to use `SMTP_SSL` and the `login` function, but keep in mind that this function (and almost every other) behave the same in both `SMTP` and `SMTP_SSL`.

login_ssl.py

```
import smtplib

sender = "sender@example.com"
receivers = ["receiver@example.com"]


def format_mail(mail):
    return "{} ".format(mail.split("@")[0], mail)

message = """From: {}
To: {}
Subject: Example Subject

This is a test mail example
""".format("{} ".format(sender.split("@")[0], sender), ", ".join(map(format_mail, receivers ↩
    )))

try:
    print("sending message: " + message)
    with smtplib.SMTP_SSL('smtp.example.com', 465) as session:
        session.login("sender@example.com", "sender_password")
        session.sendmail(sender, receivers, message)
    print("message sent")
except smtplib.SMTPException:
    print("could not send mail")
```

The only thing that changed from `simple.py` is that we are now getting an instance of `SMTP_SSL` pointing to the default SMTP SSL port (465), and we added a line which invokes `SMTP_SSL.login(user, password)`.

This example was tested with the GMail SMTP server, via SSL, with my own credentials and worked just fine.

10.3 Sending HTML

Remember how I talked about the headers required for the message to be compliant with the SMTP protocol? Well, as those are the ones required, there are other headers which can be sent that can let us do some pretty awesome stuff.

By using the headers `MIME-Version` and `Content-type` we can tell the mail client how to interpret the information we are sending. We'll now see how to set these headers to send HTML in a mail, but keep in mind that this is not just as easy as writing HTML in a browser, there are so so so... so many mail clients out there, that writing HTML that is compliant to every one of them is a pretty defying task.

html_mail.py

```
import smtplib


def format_mail(mail):
    return "{} ".format(mail.split("@")[0], mail)

smtp_server_host = "smtp.example.com"
smtp_server_port = 465
sender = "sender@example.com"
pwd = "sender_password"
receivers = ["receiver@example.com"]
message = """
<h1>This is a title</h1>
<h2>This is a sub title</h2>
This is a paragraph <strong>with some bold text</strong>.
"""
```

```
formatted_message = """From: {}
To: {}
MIME-Version: 1.0
Content-type: text/html
Subject: Example Subject

{}
""".format("{} ".format(sender.split("@")[0], sender), ", ".join(map(format_mail, receivers ↩
    )), message)

try:
    print("sending message: " + message)
    with smtplib.SMTP_SSL(smtp_server_host, smtp_server_port) as session:
        session.login(sender, pwd)
        session.sendmail(sender, receivers, formatted_message)
    print("message sent")
except smtplib.SMTPException:
    print("could not send mail")
```

As you see, the actual code for connecting to the SMTP server, logging in and sending the message is the same as before. The only thing we actually changed was the headers of the message. By setting `MIME-Version:1.0` and `Content-type: text/html`, the mail client will now know that this is HTML and should be rendered as such.

10.4 Sending Attachments

To send a mail with mixed content you need to send the `Content-type` header to `multipart/mixed` and then, text and attachment sections can be specified within boundaries. A boundary is started with two hyphens followed by a unique string, which cannot appear in the message part of the e-mail. A final boundary denoting the e-mail's final section must also end with two hyphens.

We'll now send a mail with HTML in the body and an attachment. It will be a text file containing the following piece of Lorem Ipsum:

attachment.txt

```
"Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nullam eu justo vel tortor ↩
    hendrerit dignissim et sit amet arcu. Pellentesque in sapien ipsum. Donec vitae neque ↩
    blandit, placerat leo ac, tincidunt libero. Donec ac sem ut libero volutpat facilisis ↩
    sit amet ac ante. Interdum et malesuada fames ac ante ipsum primis in faucibus. Duis ↩
    quis elit porta, bibendum lacus vel, auctor sem. Nulla ut bibendum ipsum. In efficitur ↩
    mauris sed interdum commodo. Maecenas enim orci, vestibulum et dui id, pretium ↩
    vestibulum purus. Etiam semper dui ante, convallis volutpat massa convallis ut. ↩
    Pellentesque at enim quis est bibendum eleifend sit amet eget enim.

Morbi cursus ex ut orci semper viverra. Aenean ornare erat justo. Cras interdum mauris eu ↩
    mauris aliquet tincidunt. Praesent semper non tellus a vehicula. Suspendisse potenti. ↩
    Sed blandit tempus quam. In sem massa, volutpat nec augue eu, commodo tempus metus. ↩
    Fusce laoreet, nunc in bibendum placerat, sem ex malesuada est, nec dictum ex diam nec ↩
    augue. Aliquam auctor fringilla nulla, vitae laoreet eros laoreet ut. Sed consectetur ↩
    semper risus non efficitur. Nunc pharetra rhoncus consectetur.

Nam dictum porta velit sit amet ultricies. Praesent eu est vel ex pretium mattis sit amet a ↩
     ex. Mauris elit est, eleifend et interdum nec, porttitor quis turpis. Sed nisl ligula, ↩
    tempus ac eleifend nec, faucibus at massa. Nam euismod quam a diam iaculis, luctus ↩
    convallis neque sollicitudin. Etiam eget blandit magna, non posuere nisi. Aliquam ↩
    imperdiet, eros nec vestibulum pellentesque, ante dolor tempor libero, in efficitur leo ↩
    lectus eu ex. Pellentesque sed posuere justo. Etiam vestibulum, urna at varius faucibus, ↩
     odio eros aliquet tortor, nec rhoncus magna massa eu felis. Phasellus in nulla diam.
```

```
Pellentesque blandit sapien orci, sit amet facilisis elit commodo quis. Quisque euismod
    imperdiet mi eu ultricies. Nunc quis pellentesque felis, aliquam ultrices neque. Duis
    quis enim non purus viverra molestie eget porttitor orci. Morbi ligula magna, lacinia
    pulvinar dolor at, vehicula iaculis felis. Sed posuere eget purus sed pharetra. Fusce
    commodo enim sed nisl mollis, eu pulvinar ligula rutrum.

In mattis posuere fringilla. Mauris bibendum magna volutpat arcu mollis, nec semper lorem
    tincidunt. Aenean dictum feugiat justo id condimentum. Class aptent taciti sociosqu ad
    litora torquent per conubia nostra, per inceptos himenaeos. Morbi lobortis, ipsum et
    condimentum cursus, risus nunc porta enim, nec ultrices velit libero eget dui. Nulla
    consequat id mi nec hendrerit. Suspendisse et odio at mauris viverra pellentesque. Nunc
    eget congue nisi."
```

Here goes the python code to send this file attached to a mail with HTML:

attachment_html_mail.py

```
import smtplib


def format_mail(mail):
    return "{} ".format(mail.split("@")[0], mail)


def message_template(sender, receivers, subject, boundary, body, file_name, attachment):
    return """From: {}
To: {}
Subject: {}
MIME-Version: 1.0
Content-Type: multipart/mixed; boundary={}
--{}
Content-Type: text/html
Content-Transfer-Encoding:8bit

{}
--{}
Content-Type: multipart/mixed; name="{}"
Content-Transfer-Encoding:base64
Content-Disposition: attachment; filename={}

{}
--{}--
""".format(format_mail(sender), ", ".join(map(format_mail, receivers)), subject, boundary,
    boundary, body,
           boundary, file_name, file_name, attachment, boundary)


def main():
    sender = "sender@example.com"
    receivers = ["receiver@example.com"]
    subject = "Test Mail with Attachment and HTML"
    boundary = "A_BOUNDARY"
    msg_body = """
    <h1>Hello there!</h1>
    You will find <strong>attachment.txt</strong> attached to this mail. <strong>Read it
        pls!</strong>
    """
    file_name = "attachment.txt"
    attachment = open(file_name, "rb").read().decode()

    smtp_server_host = "smtp.example.com"
    smtp_server_port = 465
```

```
    pwd = "sender_password"

    message = message_template(sender, receivers, subject, boundary, msg_body, file_name,  ↩
        attachment)

    try:
        with smtplib.SMTP_SSL(smtp_server_host, smtp_server_port) as session:
            session.login(sender, pwd)
            session.sendmail(sender, receivers, message)
        print("mail was successfully sent")
    except smtplib.SMTPException:
        print("could not send mail")


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

Again, the process of connecting to the SMTP server, logging in and sending the mail is untouched. The magic is happening in the message, with its headers and content.

The first headers define `From`, `To`, `Subject`, `MIME-Version`, `Content-Type` and, with it, `boundary`. These are the headers for the whole message.

Then, after using the boundary to separate the definitions of the global headers from the body of the message, we are defining the `Content-Type` and the `Content-Transfer-Encoding` **for the body of the mail only**, and then we insert the body of the message.

Separated from that last section, we are defining the `Content-Type`, with the `name` (of the attachment), the `Content-Transfer-Encoding` and the `Content-Disposition` with the `filename`, and then we insert the file content. Then we just end the mail with two hyphens.

10.5 Download the Code Project

This was a basic example on how to send mails with Python's smtplib module.

Download You can download the full source code of this example here: **python-mail**