ROBOTIC PROCESS AUTOMATION

SUCCINCTLY

BY **ED FREITAS**

SUCCINCTLY EBOOK SERIES

Syncfusion

Robotic Process Automation Succinctly

By
Ed Freitas

Foreword by Daniel Jebaraj

Syncfusion®
Deliver innovation with ease®

Copyright © 2020 by Syncfusion, Inc.

2501 Aerial Center Parkway
Suite 200
Morrisville, NC 27560
USA
All rights reserved.

Important licensing information. Please read.

This book is available for free download from www.syncfusion.com on completion of a registration form.

If you obtained this book from any other source, please register and download a free copy from www.syncfusion.com.

This book is licensed for reading only if obtained from www.syncfusion.com.

This book is licensed strictly for personal or educational use.

Redistribution in any form is prohibited.

The authors and copyright holders provide absolutely no warranty for any information provided.

The authors and copyright holders shall not be liable for any claim, damages, or any other liability arising from, out of, or in connection with the information in this book.

Please do not use this book if the listed terms are unacceptable.

Use shall constitute acceptance of the terms listed.

SYNCFUSION, SUCCINCTLY, DELIVER INNOVATION WITH EASE, ESSENTIAL, and .NET ESSENTIALS are the registered trademarks of Syncfusion, Inc.

Technical Reviewer: James McCaffrey
Copy Editor: Courtney Wright
Acquisitions Coordinator: Tres Watkins, vice president of content, Syncfusion, Inc.
Proofreader: Jacqueline Bieringer, content producer, Syncfusion, Inc.

Table of Contents

The Story Behind the *Succinctly* Series of Books **6**

Robotic Process Automation **10**

Chapter 1 RPA Fundamentals **12**

Overview 12

Basic concepts 13

Getting started 13

Summary 15

Chapter 2 Accessing Files and Folders **16**

Quick intro 16

Automated backups config 16

Reading the config file 17

Walking a directory tree 19

Merging paths 21

Tracking file changes with SQLite 23

Summary 37

Chapter 3 File Operations **38**

Quick intro 38

Copying files 38

Moving files 42

Full copy files script 44

Full move files script 46

Deleting files 47

Deleting folders 49

Creating a zip file 49

Summary 54

Chapter 4 Backup Script 55

Quick intro 55

Putting it all together 55

File actions 56

Zipping files 59

Main functions 61

Finished backup script 62

Summary 68

Chapter 5 Assisted Data Entry Automation 69

Quick intro 69

Reading PDF forms 70

Generating the browser script 73

The AutoForm script 78

Finished code 84

Scanned or text PDFs 84

Final thoughts 88

Appendix – My Python 2.7 Backup Script 90

Background 90

Main FTP method 93

Full source code 94

The Story Behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Ed Freitas is a consultant on software development related to financial process automation, accounts payable processing, and data extraction.

He loves technology and enjoys playing soccer, running, traveling, life-hacking, learning, and spending time with his family.

You can reach him at https://edfreitas.me.

Acknowledgments

Many thanks to all the people who contributed to this book, including the amazing Syncfusion team that helped this book become a reality—especially Jacqueline Bieringer, Tres Watkins, and Graham High.

The manager and technical editor thoroughly reviewed the book's organization, code quality, and overall accuracy—Jacqueline Bieringer from Syncfusion, and James McCaffrey from Microsoft Research. Thank you.

This book is dedicated to my father—for everything you did, for everyone you loved—thank you.

Robotic Process Automation

Around the world, organizations are facing the increasing challenge of becoming more efficient by augmenting productivity with their existing resources—or sometimes even less. Resources do not always refer to people; they can refer to infrastructure, hardware, software, processes, and even time.

Robotic process automation (RPA) is a growing trend in the business world that is often misunderstood. RPA is a form of business process automation technology based on metaphorical software robots or a digital workforce, built on the notion of artificial intelligence workers.

When people hear the word “robotic,” they usually think of an intelligent artificial entity that is smart enough that it might outperform them, take their jobs, and rule the world.

There are a plethora of Hollywood science-fiction movies that have conveyed the image of a future world being dominated by robots, where humans become second-class citizens.

Although within the laws of physics, that possibility could exist in a not-so-distant future, this is still well beyond the current level of sophistication of robots, and robotics as a whole.

To quote the world-famous theoretical physicist Michio Kaku:

> *"Right now, our machines are as smart as insects. Eventually, they’ll be as smart as mice. After that, they’ll be as smart as dogs and cats. Probably by the end of the century, who knows, they’ll be as smart as monkeys. At that point, they could become potentially dangerous, because monkeys can formulate their own plans; they don’t have to listen to you. They can formulate their own strategies, their own goals and I would say, therefore, at that point, let’s put a chip in their brain to shut them off if they have murderous thoughts."*

In many ways, humans have been inventing tools to automate processes throughout history. Think of the wheel for a moment, which was invented long before modern-day written scripture.

Before the advent of the wheel, donkeys, goats, camels, horses, and human backs were the only means of transporting goods from one place to another. The invention of the wheel changed that.

The wheel didn’t fully replace horses and donkeys, but it augmented their ability to transport goods. Horses pulling wagons were able to open new routes all over the world in ancient times. Even not so long ago, just a few hundred years back, those wagons led to the expansion of the United States to the west.

We barely use horses, donkeys, and wagons anymore to transport goods, but horses and donkeys are still around—and are probably quite thankful for the invention of the wheel.

The same goes for RPA—it's not a technology that is intended to replace human workers, although in some specific tasks there might be some overlap. Instead, it is a technology that can augment the productivity of human workers by liberating them from tedious and repetitive tasks that can be performed faster and with more accuracy by a computer.

RPA opens exciting possibilities to automate time-consuming tasks. Imagine having to manually open 5,000 PDF files, individually look at each file, find relevant data, copy the data from each file, and manually enter those copied values into a CRM application, multiple times per PDF. If you were given this task, every day, how excited would you be?

If I were assigned this task, my first thought would be about how I could automate it (or at least some of it). I think most people would feel the same.

My father used to always tell me that the most precious resource in life was not money, but time—as time is finite for everyone, it should be used wisely.

RPA gives us the ability to create software robots, which are nothing more than computer programs that mimic the actions a human would take to perform that same repetitive task, over and over. This gives people the ability to use their time, energy, and intelligence on other, less repetitive tasks that cannot be performed by a computer, like watering their gardens, hosting a business meeting, or giving counseling to divorced couples.

Therefore, instead of being a threat to humans, RPA can make us more productive, and help us make smarter and more efficient use of our time. But like everything, it must be used within reason and the ethical boundaries of society.

In this book, we'll explore some ways we can use nonproprietary and open-source software to write our RPA scripts, which can help us automate tedious and repetitive tasks—so we can have time to put our energy into other endeavors and leave the computer to do what it was designed to do: compute.

Sounds exciting, right? Let's dive right in!

Chapter 1 RPA Fundamentals

Overview

RPA is all about how to use software programs and scripts to type in words, perform mouse clicks, open files, read file contents, open desktop and web applications, and enter data, so some processing can take place that is repetitive, tedious, and time-consuming.

The term *digital workforce* is sometimes used when referring to what RPA does and the scope it covers.

More than a set of specific programs, libraries, and tools, RPA is about techniques and how to apply them to automate specific tasks.

There are many proprietary software applications and frameworks on the market that claim to have a one-size-fits-all solution to all business process automation needs. Several key players in this market offer out-of-the-box RPA solutions that can be customized for different automation tasks by using a drag-and-drop (low-code) approach. The goal of this book is not to give you insights on those proprietary solutions, but instead to explain and highlight how you can use open-source resources and a few easy-to-learn techniques to achieve the same results as you would using proprietary software.

With the rise of relatively easy-to-learn dynamic programming languages such as Python, RPA is now within the reach of anyone who has intermediate programming knowledge, and who has a curious mind and desire to learn how to automate repetitive tasks.

Even if you have minimal programming knowledge, picking up a language like Python is a very straightforward process, and frankly, quite enjoyable.

The goal is not to discourage you or your organization from exploring, checking, and testing proprietary RPA software, which still offers a great level of abstraction from a Python code-centric approach to RPA. Instead, the goal is to give you insights on how you can use existing, top-performing and state-of-the-art open-source libraries to achieve the same results—possibly in even less time—at a fraction of the cost of using and deploying proprietary software.

Proprietary software offers amazing value, but it does come with a heavy price tag. Think of licenses, training, ongoing maintenance fees, professional services, and support. If your organization needs to see the immediate benefits of using RPA and has the budget, this might be the best option.

If you have a bit more time to invest in learning how to implement an RPA solution, would like to be completely in control of your RPA strategy and techniques, and do not want to have any dependencies with third-party offerings, then writing your own RPA code is probably the best long-term strategy and investment you can make.

In this book, we will explore how we can use Python and some RPA techniques to automate some common and repetitive business tasks, which in the long run can be of great benefit to your organization.

Basic concepts

The R in RPA stands for *robotic*. Now, that's not a typical Japanese robot that you might have seen in an automobile factory placing doors into car frames, or one that walks around bipedal, like a human.

Robotic, in this case, refers to a software script. It is nothing other than another computer program that contains line-by-line instructions to carry out a specific task.

The difference though, compared to a regular computer program, is that an RPA script should perform specific steps that would be similar or the same as a human would perform when doing a repetitive task on a computer.

The P in RPA stands for *process*, which is nothing other than tasks that need to be executed multiple times.

The A in RPA stands for *automation*, which consists of putting together a set of instructions that a computer can perform to automate a series of manual tasks, over and over.

In the RPA world, automation tasks are carried out by executing scripts or programs that are designed with a single, specific purpose in mind.

This means that if we intend to automate the reading and extraction of data contained within PDF files, we should not use the same script to automate another task that is not related to the first one.

Therefore, when you think about a series of manual tasks to automate, make sure that the script you will create is for a specific purpose. Every RPA script should focus on solving a single problem.

Getting started

You can download all the finished code examples that will be implemented throughout this book here.

We'll be implementing the various RPA scripts throughout this book with Python. Although we could use any other programming language such as Java, C#, or C++, which are faster to execute, Python offers great out-of-the-box tooling, libraries, and support for many RPA-related tasks.

Python is also easier to learn, and even though it doesn't execute as fast as Java, C#, or C++, it gets the job done with fewer lines of code.

If you don't have Python installed, make sure you install it first. All the examples laid out in this book will be done on a computer running the Windows 10 operating system.

However, as Python is an OS-independent language that can be executed on multiple operating systems, you should be able to port some of the examples that are non-Windows specific to other operating systems with few or no changes.

In your browser, navigate to the official Python website to download the right Python executable for your operating system.

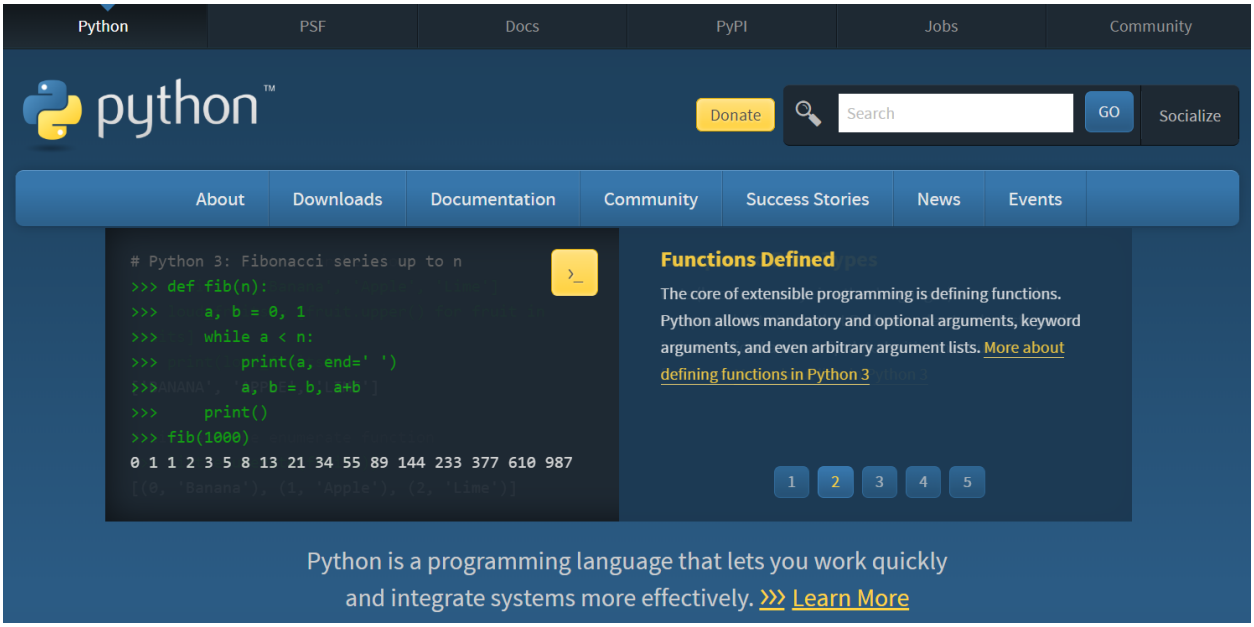


Figure 1-a: The Python Website

Go to the **Downloads** section and select the appropriate installer for your platform. If you are using Windows, then choose the latest option available.

Figure 1-b: The Python Downloads Section

All the code that we will be writing throughout this book is compatible with Python 3.6 and later, so if you already have Python 3.6.X installed, you don't necessarily need to install the latest version.

If you do not have Python installed, once you have downloaded the installer or installation files, simply follow the installation instructions or steps, which for Windows consist of a few clicks using an intuitive setup wizard.

You can choose the default installation or upgrade option, or do a custom install, where you can choose the location and what features to install. In my case, I've chosen the upgrade option. So, follow the steps and you should be good to go. I use a 32-bit version of Python for compatibilty reasons, but you can use a 64-bit version if you wish.

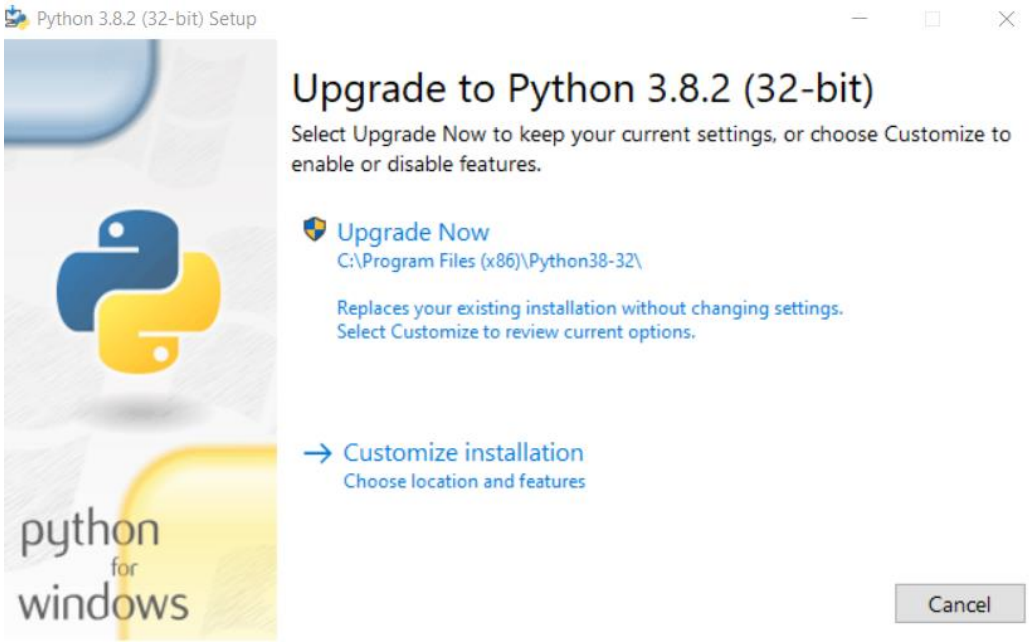


Figure 1-c: The Python Installation Wizard (Windows)

With Python installed and up to date, we are ready to start exploring the world of RPA.

Summary

At this stage, you should have a fundamental understanding of what RPA is, what it is intended for, and what benefits it offers.

You should also have Python installed on your computer, which is going to be the programming language for the RPA scripts we will be writing throughout this book.

Next, we are going to dive right into file automation, which is perhaps the most important aspect of RPA—without it, we can't automate any processes.

File automation is an essential aspect of RPA. We need to understand how to traverse directories and subfolders, and be able to find files, read their contents, and extract data, which can then be used to automate a manual data entry process on another web or software application.

Chapter 2 Accessing Files and Folders

Quick intro

Manual, repetitive processes that humans carry out with computers mostly involve gathering information from one place, validating the data gathered, and then using part of the data in other systems.

Most information is stored within files or databases. To be able to extract data to automate certain processes, we need to know how to access files and understand how to automate frequent operations with them, which include: copying, moving, reading, and writing files. This is what we will be covering throughout this chapter.

Automated backups config

As automating specific business processes requires changing data, it is always a good idea to make backups of that data before using it to automate other processes. There's always a possibility that some of that information might be altered due to file manipulation.

Before we explore how to extract data to automate specific business processes, it's good that we understand how we can perform specific operations with files and folders, such as reading configuration files, walking a directory tree, merging file paths, and detecting file changes.

The goal for this chapter is to understand how files and folders can be accessed. In the next chapter, we can see how to perform file and folder operations, which will later allow us to create a backup Python script that can be configured to copy, move, zip, and delete files within any folder.

The script would be configured through a .ini file that would have the following structure.

Code Listing 2-a: Python Backup Script Configuration .ini File (readcfg.ini)

```
C:\Projects\RPA Succinctly\Temp;.py,.ini|C:\Temp\RPASuccinctly_%%.zip|z
C:\Projects\RPA Succinctly\Temp|/foo/RPASuccinctly|f
C:\Projects\RPA Succinctly\Temp|C:\Temp\RPASuccinctly|c
C:\Projects\RPA Succinctly\Temp|C:\Temp\RPASuccinctly|m
C:\Temp\RPASuccinctly|C:\Temp\RPASuccinctly2|c
C:\Temp\RPASuccinctly2||d
```

Let's explore the structure of this .ini file and the operation each line represents. Each line has three parts, and each part is separated by a pipe character (**|**). Let's explore the first line within the .ini file.

The first part indicates the origin of the files: **`C:\Projects\RPA Succinctly\Temp;.py,.ini`**.

The second part indicates the destination of the files: **C:\Temp\RPASuccinctly_%%.zip**.

The third part indicates the type of operation that will take place between the first and second parts: **z**, which in this case, represents a zip (compression) operation.

So in essence, in this first line, we are instructing the Python script to zip all the files contained within the **C:\Projects\RPA Succinctly\Temp** folder, except for files with the **.py** and **.ini** extensions, and archive them into a file called **C:\Temp\RPASuccinctly_%%.zip**, where the **%%** chars will be replaced with the date and time the operation takes place.

On the second line, the operation taking place is an FTP upload of the files located within **C:\Projects\RPA Succinctly\Temp** to **\foo\RPASuccinctly**. This operation is represented by the **f** char.

The third and fourth lines indicate copy (**c**) and move (**m**) operations, respectively, from **C:\Projects\RPA Succinctly\Temp** to **C:\Temp\RPASuccinctly**.

The fifth line also indicates a copy (**c**) operation from **C:\Temp\RPASuccinctly** to **C:\Temp\RPASuccinctly2**.

The sixth line of the config file indicates a delete operation for the files contained within the **C:\Temp\RPASuccinctly2** folder.

This configuration file may contain any number of lines. An operation may only be described in one line.

Reading the config file

Now that we have defined what the config file looks like and what operations can be configured, it's now time to write code, read its content, and parse each line. Let's explore that code.

Code Listing 2-b: Reading the .ini File (readcfg.py)

```
import os

def readcfg(config):
    items = []
    if os.path.isfile(config):
        cfile = open(config, 'r')
        for line in cfile.readlines():
            items.append(parsecfgline(line))
        cfile.close()
    return items

def parsecfgline(line):
    option = {}
    if '|' in line:
```

```
        opts = line.split('|')
        if len(opts) == 3:
            option['origin'] = extcheck(opts[0], 0)
            option['exclude'] = extcheck(opts[0], 1)
            option['dest'] = opts[1]
            option['type'] = opts[2].replace('\n', '')
    return option

def extcheck(opt, idx):
    res = ''
    if ';' in opt:
        opts = opt.split(';')
        if len(opts) == 2:
            res = opts[0] if idx == 0 else opts[1]
    elif idx == 0:
        res = opt
    return res

opts = readcfg(os.path.splitext(os.path.basename(__file__))[0] + '.ini')

for opt in opts:
    print(opt)
```

This Python script starts by invoking the **`readcfg`** function, which is responsible for reading the **readcfg.ini** file. The name of the .ini file is retrieved by executing the following instruction.

```
os.path.splitext(os.path.basename(__file__))[0] + '.ini'
```

This instruction extracts the base name of the current file (**`__file__`**). In other words, it returns the **`<folder-name>\readcfg`** string, and by passing that value to the **`os.path.splitext()[0]`** method, it strips out the folder name from the file name. The file name is appended the .ini extension.

The configuration file name is passed to the **`readcfg`** function, which loops through each of the lines contained within the file and returns a list of the lines read.

For each line that is read, the **`parsecfgline`** function is invoked, which is responsible for parsing the origin (**`origin`**), destination (**`dest`**), what files types to exclude (**`exclude`**), and the type of operation taking place (**`type`**). This is done by splitting each line based on the occurrence of the pipe **(|)** character.

The **`parsecfgline`** function invokes the **`extcheck`** function, which determines if any file types should be excluded from the operation.

The results returned by the **`readcfg`** function are printed out to the console.

Let's now execute the script from the command line or the built-in terminal within Visual Studio Code (VS Code), which is my editor of choice—but you are free to use any other editor of your choice.

```
PROBLEMS   OUTPUT   DEBUG CONSOLE   TERMINAL                                   1: powershell

PS C:\Projects\Projects\Syncfusion\RPA Succinctly\demo> python3 readcfg.py
{'origin': 'C:\\Projects\\RPA Succinctly\\Temp', 'exclude': '.py,.ini', 'dest': 'C:\\Temp\\RPASuccinctly_%%.zip', 'type': 'z'}
{'origin': 'C:\\Projects\\RPA Succinctly\\Temp', 'exclude': '', 'dest': '/foo/RPASuccinctly', 'type': 'f'}
{'origin': 'C:\\Projects\\RPA Succinctly\\Temp', 'exclude': '', 'dest': 'C:\\Temp\\RPASuccinctly', 'type': 'c'}
{'origin': 'C:\\Projects\\RPA Succinctly\\Temp', 'exclude': '', 'dest': 'C:\\Temp\\RPASuccinctly', 'type': 'm'}
{'origin': 'C:\\Temp\\RPASuccinctly', 'exclude': '', 'dest': 'C:\\Temp\\RPASuccinctly2', 'type': 'c'}
{'origin': 'C:\\Temp\\RPASuccinctly2', 'exclude': '', 'dest': '', 'type': 'd'}
PS C:\Projects\Projects\Syncfusion\RPA Succinctly\demo>
```

Figure 2-a: Console Output – readcfg.py Execution

Notice how the script has returned the lines parsed from the .ini file, split into different sections, which we will use later to automate a backup process.

The script was executed by running the following command.

Code Listing 2-c: Command to Execute the Python Script (readcfg.py)

```
python3 readcfg.py
```

Note: On some Python Windows installations, the name of the Python executable might be python3.exe instead of simply python.exe (which is true in my case).

If **python3.exe** is not available for you, simply use **python.exe**. In that case, the command will be as follows.

Code Listing 2-d: Alternate Command to Execute the Python Script (readcfg.py)

```
python readcfg.py
```

For simplicity, going forward I'll always use **python** instead of **python3**, so it's less confusing and easier to follow along.

An important consideration is that the Python executable needs to be in your system path so you can execute it from any folder.

Normally, this process is taken care of by the installer. In any case, it's worthwhile to check that Python is within your system path; for Windows, you can do this using these instructions.

Walking a directory tree

A fundamental aspect of extracting data from files is the ability to traverse a directory tree and find all the files we want to process. In Python, this is known as walking a directory tree.

This means that we can iterate through all the folders and subfolders of a specific directory and get a list of all the files contained within.

In most programming languages, this can only be done with a recursive function, which is one that invokes itself, but in Python this can be achieved quite easily without the need to use recursion.

Let's explore how we can easily walk a directory tree with Python.

Code Listing 2-e: Walking a Directory Tree with Python (walkdir.py)

```
import os

for fn, sflds, fnames in os.walk('.\\'):
    print('Current folder is ' + fn)
    for sf in sflds:
        print(sf + ' is a subfolder of ' + fn)
    for fname in fnames:
        print(fname + ' is a file of ' + fn)
    print('')
```

Note: It's common in Windows Python code for the backslash character (\) to be escaped; thus it can be written as (\\). Going forward, I'll be using mostly Windows folder notation.

The directory we want to walk across, including all its subdirectories and files, is **`.\`**, which is the parent folder of where the script resides. You may choose any other folder.

The **`os.walk`** method returns the main folder name (**`fn`**), the subfolders (**`sflds`**), and all the files (**`fnames`**) contained within the main directory and subfolders.

Let's now execute this script from the command line or built-in terminal within VS Code as follows.

Code Listing 2-f: Command to Execute the Python Script (walkdir.py)

```
python walkdir.py
```

Here are the results of the execution of the script on my machine.

```
PS C:\Projects\Projects\Syncfusion\RPA Succinctly\demo> python3 walkdir.py
Current folder is .\
.vscode is a subfolder of .\
readcfg.ini is a file of .\
readcfg.py is a file of .\
walkdir.py is a file of .\

Current folder is .\.vscode
settings.json is a file of .\.vscode
```

Figure 2-b: Console Output – walkdir.py Execution

Notice how the execution of this script returns all the subfolders and files contained within the folder that is passed to the **`os.walk`** method.

Depending on the number of files in that specific directory, the output will vary.

Merging paths

When we copy or move files from one folder to another, sometimes we want to preserve the original path and replicate that same file path to the new destination by recreating the same folder structure where the original files exist, or once were. This is what the concept of merging paths is all about.

Say, for instance, we have the origin path **`C:\Projects\Test`** and the destination path **`D:\Backup`**.

The merged path would be **`D:\Backup\Projects\Test`**. All the files contained with the origin would be copied or moved to the destination, but keep the same folder structure of their origin.

Merging the origin path into the destination path can be quite handy if you want to retain evidence of where the files originally existed, without having to use a database to store the original file locations.

To see how this works, let's create a new Python script called **mergepaths.py** and add the following code to it.

Coding Listing 2-g: Merging Paths (mergepaths.py)

```
import os

def mergepaths(path1, path2):
    pieces = []
    parts1, tail1 = os.path.splitdrive(path1)
    parts2, tail2 = os.path.splitdrive(path2)
    result = path2
    parts1 = tail1.split('\\') if '\\' in tail1 else tail1.split('/')
```

```
    parts2 = tail2.split('\\') if '\\' in tail2 else tail2.split('/')
    for pitem in parts1:
        if pitem != '':
            if not pitem in parts2:
                pieces.append(pitem)
    for piece in pieces:
        result = os.path.join(result, piece)
    return result

print(mergepaths("C:\\Projects\\Test", "C:\\Temp\\RPASuccinctly"))
```

We can execute this script using the following command from the prompt or built-in terminal within VS Code.

Code Listing 2-h: Command to Execute the Python Script (mergepaths.py)

```
python mergepaths.py
```

The execution of the script would return the following result.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Projects\Projects\Syncfusion\RPA Succinctly\demo> python3 mergepaths.py
C:\Temp\RPASuccinctly\Projects\Test
```

Figure 2-c: Console Output – mergepaths.py Execution

Notice how the origin location **`C:\Projects\Test`** was merged with the destination path **`C:\Temp\RPASuccinctly`** into one path.

To understand how this works, let's explore what the code of the **`mergepaths`** function does.

Both file paths, origin and destination, are split into their various subparts, which is done as follows.

```
parts1, tail1 = os.path.splitdrive(path1)

parts2, tail2 = os.path.splitdrive(path2)
```

Depending on the operating system in use, the splitting of each of the paths is done looking for the backslash (****) or forward slash (**/**) characters.

```
parts1 = tail1.split('\\') if '\\' in tail1 else tail1.split('/')

parts2 = tail2.split('\\') if '\\' in tail2 else tail2.split('/')
```

Then, for each subpart within the origin location (**pitem**)—each subpart represents a folder within the original location (**parts1**)—we check if that subpart is not part of the destination location (**parts2**). If not, we add it to a list called **pieces**, which contains the subparts that will be merged into the destination location.

```
for pitem in parts1:
    if pitem != '':
        if not pitem in parts2:
            pieces.append(pitem)
```

Then, for each of the pieces (**piece**) contained within the list of **pieces** to be merged, we concatenate (by invoking the **os.path.join** method) each of those subparts to the destination location.

```
for piece in pieces:
    result = os.path.join(result, piece)
```

The resultant merged path is assigned to the **result** variable that the **mergepaths** function returns**.** By using the **mergepaths** function, we can preserve the original file locations during a backup operation.

Tracking file changes with SQLite

Sometimes when working with files, it is important to keep track of file changes. This can be quite useful if multiple people are working with a specific set of files, and when these files change, an action could be performed on them.

Keeping track of files is not complicated, but it does require having a small database of the each of the MD5 hash values of the files that we want to monitor, and checking if the hash values stored are different than the current file hash values.

You can think of a hash value as a file's unique signature—something that can identify the file during a specific period.

For instance, the following files, based on their file size, file name, and modified date, will each have their unique MD5 hash value.

dfilechanges.py	4/13/2020 12:06 PM	Python File	6 KB
mergepaths.py	4/12/2020 11:14 PM	Python File	1 KB
readcfg.ini	4/12/2020 4:38 PM	Configuration setti...	1 KB
readcfg.py	4/8/2019 9:10 PM	Python File	1 KB

Figure 2-d: List of Files to Monitor

These hash values can be initially calculated and stored within a database, along with the file name and file path.

File Name (fname)	Hashtag (md5)
\..\demo\dfilechanges.py	1234567890
\..\demo\mergepaths.py	2234567891
\..\demo\readcfg.ini	3234567892
\..\demo\readcfg.py	4234567893

Figure 2-e: Database of Files and MD5 Hash Values

If a file is modified, the file's hash value will change, so by keeping a database of files name, file paths, and file hash values, we can check whether a file has changed recently.

SQLite is a compact and embeddable version of a SQL database that is particularly useful and well suited for this type of scenario.

What we are going to do is to create a Python script called **dfilechanges.py**, which stands for "detect file changes."

You can see the finished code for this script in Code Listing 2-i. Let's have a look at the full code of this script and analyze each of its parts.

To make some parts of the code easier to follow, I've added a few helpful comments.

Code Listing 2-i: Finished dfilechanges.py Script

```
import os
import sqlite3
import hashlib

def getmoddate(fname):
    """Get file modified date"""
    try:
        mtime = os.path.getmtime(fname)
    except OSError as emsg:
        print(str(emsg))
        mtime = 0
    return mtime

def md5short(fname):
    """Get md5 file hast tag..."""
    enc = fname + '|' + str(getmoddate(fname))
    return hashlib.md5(enc.encode('utf-8')).hexdigest()

def getbasefile():
    """Name of the SQLite DB file"""
```

```
    return os.path.splitext(os.path.basename(__file__))[0]

def connectdb():
    """Connect to the SQLite DB"""
    try:
        dbfile = getbasefile() + '.db'
        conn = sqlite3.connect(dbfile, timeout=2)
    except BaseException as err:
        print(str(err))
        conn = None
    return conn

def corecursor(conn, query):
    """Opens a SQLite DB cursor"""
    result = False
    try:
        cursor = conn.cursor()
        cursor.execute(query)
        rows = cursor.fetchall()
        numrows = len(list(rows))
        if numrows > 0:
            result = True
    except sqlite3.OperationalError as err:
        print(str(err))
        cursor.close()
    finally:
        cursor.close()
    return result

def tableexists(table):
    """Checks if a SQLite DB Table exists"""
    result = False
    core = "SELECT name FROM sqlite_master WHERE type='table' AND name='"
    try:
        conn = connectdb()
        if not conn is None:
            query = core + table + "'"
            result = corecursor(conn, query)
            conn.close()
    except sqlite3.OperationalError as err:
        print(str(err))
        conn.close()
    return result
```

```
def createhashtableidx():
    """Creates a SQLite DB Table Index"""
    table = 'files'
    query = 'CREATE INDEX idxfile ON files (file, md5)'
    try:
        conn = connectdb()
        if not conn is None:
            if not tableexists(table):
                try:
                    cursor = conn.cursor()
                    cursor.execute(query)
                except sqlite3.OperationalError:
                    cursor.close()
                finally:
                    conn.commit()
                    cursor.close()
    except sqlite3.OperationalError as err:
        print(str(err))
        conn.close()
    finally:
        conn.close()

def createhashtable():
    """Creates a SQLite DB Table"""
    result = False
    query = "CREATE TABLE files ({file} {ft} PRIMARY KEY, {md5} {ft})"\
                .format(file='file', md5='md5', ft='TEXT')
    try:
        conn = connectdb()
        if not conn is None:
            if not tableexists('files'):
                try:
                    cursor = conn.cursor()
                    cursor.execute(query)
                except sqlite3.OperationalError:
                    cursor.close()
                finally:
                    conn.commit()
                    cursor.close()
                    result = True
    except sqlite3.OperationalError as err:
        print(str(err))
        conn.close()
    finally:
```

```
        conn.close()
    return result

def runcmd(qry):
    """Run a specific command on the SQLite DB"""
    try:
        conn = connectdb()
        if not conn is None:
            if tableexists('files'):
                try:
                    cursor = conn.cursor()
                    cursor.execute(qry)
                except sqlite3.OperationalError:
                    cursor.close()
                finally:
                    conn.commit()
                    cursor.close()
    except sqlite3.OperationalError as err:
        print(str(err))
        conn.close()
    finally:
        conn.close()

def updatehashtable(fname, md5):
    """Update the SQLite File Table"""
    qry = "UPDATE files SET md5='{md5}' WHERE file='{fname}'"\
        .format(fname=fname, md5=md5)
    runcmd(qry)

def inserthashtable(fname, md5):
    """Insert into the SQLite File Table"""
    qry = "INSERT INTO files (file, md5) VALUES ('{fname}', '{md5}')"\
        .format(fname=fname, md5=md5)
    runcmd(qry)

def setuphashtable(fname, md5):
    """Sets Up the Hash Table"""
    createhashtable()
    createhashtableidx()
    inserthashtable(fname, md5)

def md5indb(fname):
    """Checks if md5 hash tag exists in the SQLite DB"""
    items = []
```

```
    qry = "SELECT md5 FROM files WHERE file = '" + fname + "'"
    try:
        conn = connectdb()
        if not conn is None:
            if tableexists('files'):
                try:
                    cursor = conn.cursor()
                    cursor.execute(qry)
                    for row in cursor:
                        items.append(row[0])
                except sqlite3.OperationalError as err:
                    print(str(err))
                    cursor.close()
                finally:
                    cursor.close()
    except sqlite3.OperationalError as err:
        print(str(err))
        conn.close()
    finally:
        conn.close()
    return items

def haschanged(fname, md5):
    """Checks if a file has changed"""
    result = False
    oldmd5 = md5indb(fname)
    numits = len(oldmd5)
    if numits > 0:
        if oldmd5[0] != md5:
            result = True
            updatehashtable(fname, md5)
    else:
        setuphashtable(fname, md5)
    return result

def getfileext(fname):
    """Get the file name extension"""
    return os.path.splitext(fname)[1]

def checkfilechanges(folder, exclude):
    """Checks for files changes"""
    for subdir, dirs, files in os.walk(folder):
        for fname in files:
            origin = os.path.join(subdir, fname)
```

```
            if os.path.isfile(origin):
                fext = getfileext(origin)
                if not fext in exclude:
                    md5 = md5short(origin)
                    if haschanged(origin, md5):
                        print(origin + ' has changed...')
                    else:
                        print(origin + ' has NOT changed...')

checkfilechanges("./", ['.db'])
```

Before we run the code, let's analyze what is going on and understand each of its parts and functions.

First, we import the following built-in Python libraries, which will allow us to work with SQLite and MD5 hash values, respectively.

```
import sqlite3

import hashlib
```

Then, we have the **`getmoddate`** function, which is used for retrieving the last modified date of a specific file.

```
def getmoddate(fname):
    """Get file modified date"""
    try:
        mtime = os.path.getmtime(fname)
    except OSError as emsg:
        print(str(emsg))
        mtime = 0

    return mtime
```

Next, we have the **`md5short`** function, which is used for calculating the MD5 hash values of a specific file.

```
def md5short(fname):
    """Get md5 file hash value..."""
    enc = fname + '|' + str(getmoddate(fname))

    return hashlib.md5(enc.encode('utf-8')).hexdigest()
```

As you can see, the MD5 hash value is generated as a combination of the encoded UTF-8 file name and the last modified date of the file.

Next we have the **getbasefile** function, which is responsible for determining the name of the SQLite database that will be used to store the MD4 hash values of the files we want to monitor for changes.

```
def getbasefile():
    """Name of the SQLite DB file"""

    return os.path.splitext(os.path.basename(__file__))[0]
```

The name of this database file will be the same as the Python script, but with the .db file extension, which we can see as follows.

dfilechanges.db	4/13/2020 12:55 PM	Data Base File	12 KB
dfilechanges.py	4/13/2020 12:55 PM	Python File	7 KB

Figure 2-f: The Database File (dfilechanges.db) and Python Script (dfilechanges.py)

The database file gets automatically created by the script—within the same folder the script resides—the first time the script is executed.

Next, we have the **connectdb** function, which is responsible for establishing the connection to the database.

```
def connectdb():
    """Connect to the SQLite DB"""
    try:
        dbfile = getbasefile() + '.db'
        conn = sqlite3.connect(dbfile, timeout=2)
    except BaseException as err:
        print(str(err))
        conn = None

    return conn
```

As you can see, this function simply invokes the **sqlite3.connect** method so a connection can be established to the database.

Next, we have the **corecursor** function, which creates a SQL cursor on the database. A cursor helps with the retrieval of the records stored in the database.

```
def corecursor(conn, query):
    """Opens a SQLite DB cursor"""
    result = False
    try:
        cursor = conn.cursor()
        cursor.execute(query)
        rows = cursor.fetchall()
        numrows = len(list(rows))
```

```
        if numrows > 0:
            result = True
    except sqlite3.OperationalError as err:
        print(str(err))
        cursor.close()
    finally:
        cursor.close()

    return result
```

This function provides the script with a simple and fast way to access all the records that exist in the database by prefetching all the records stored. This function is also useful when checking whether a particular table exists within the database, or whether one has any records.

In our case, we are using an SQLite database with one table only. However, if our database has multiple tables, then using this function would be a good way to know whether an existing table has any records.

Next, we have the **`tableexists`** function, which queries the SQLite master database to check if the table to be used actually exists in the database.

```
def tableexists(table):
    """Checks if a SQLite DB Table exists"""
    result = False
    core = "SELECT name FROM sqlite_master WHERE type='table' AND name='"
    try:
        conn = connectdb()
        if not conn is None:
            query = core + table + "'"
            result = corecursor(conn, query)
            conn.close()
    except sqlite3.OperationalError as err:
        print(str(err))
        conn.close()

  return result
```

This is a failsafe function that is used for verifying that we don't execute operations on a table that doesn't exist within our database file.

Next, we have the **`createhashtableidx`** function, which is responsible for creating a database index on the **`files`** table, which is used to keep the hash values of the files we are interested in monitoring.

```
def createhashtableidx():
    """Creates a SQLite DB Table Index"""
    table = 'files'
    query = 'CREATE INDEX idxfile ON files (file, md5)'
```

```
    try:
        conn = connectdb()
        if not conn is None:
            if not tableexists(table):
                try:
                    cursor = conn.cursor()
                    cursor.execute(query)
                except sqlite3.OperationalError:
                    cursor.close()
                finally:
                    conn.commit()
                    cursor.close()
    except sqlite3.OperationalError as err:
        print(str(err))
        conn.close()
    finally:

        conn.close()
```

From a database development perspective, it's usually a good idea to have indexes, as this helps to speed up and improve the retrieval and querying of records, especially when the number of records on a table grows.

Next, we have the **`createhashtable`** function, which creates the **`files`** table within the database. This table is used for storing the file names and MD5 hash values of the files we want to monitor for changes.

```
def createhashtable():
    """Creates a SQLite DB Table"""
    result = False
    query = "CREATE TABLE files ({file} {ft} PRIMARY KEY, {md5} {ft})"\
                .format(file='file', md5='md5', ft='TEXT')
    try:
        conn = connectdb()
        if not conn is None:
            if not tableexists('files'):
                try:
                    cursor = conn.cursor()
                    cursor.execute(query)
                except sqlite3.OperationalError:
                    cursor.close()
                finally:
                    conn.commit()
                    cursor.close()
                    result = True
    except sqlite3.OperationalError as err:
```

```
        print(str(err))
        conn.close()
    finally:
        conn.close()

    return result
```

Both the **`createhashtable`** and **`createhashtableidx`** functions only get executed when the **`files`** table does not exist yet in the database. This means that the **`createhashtable`** and **`createhashtableidx`** functions are only used the first time the script runs, or when the script executes, but the database file does not yet exist.

Next, we have the **`runcmd`** function, which, as its name implies, is used for running a specific command against the database if the **`files`** table exists.

```
def runcmd(qry):
    """Run a specific command on the SQLite DB"""
    try:
        conn = connectdb()
        if not conn is None:
            if tableexists('files'):
                try:
                    cursor = conn.cursor()
                    cursor.execute(qry)
                except sqlite3.OperationalError:
                    cursor.close()
                finally:
                    conn.commit()
                    cursor.close()
    except sqlite3.OperationalError as err:
        print(str(err))
        conn.close()
    finally:

        conn.close()
```

These commands can be: inserting a new record, updating an existing record, deleting a record, or querying the **`files`** table to retrieve one or more specific records.

As we will see shortly, the **`runcmd`** function is one of the pillars of this script, and it used by most of the other functions within this Python script.

The **`updatehashtable`** function is responsible for updating specific records on the **`files`** table.

```
def updatehashtable(fname, md5):
    """Update the SQLite File Table"""
    qry = "UPDATE files SET md5='{md5}' WHERE file='{fname}'"\
        .format(fname=fname, md5=md5)
```

```
    runcmd(qry)
```

As you can see, it simply invokes the **runcmd** function by passing a SQL UPDATE statement as a parameter, which updates a specific record on the table.

The **inserthashtable** function is responsible for inserting new records into the **files** table.

```
def inserthashtable(fname, md5):
    """Insert into the SQLite File Table"""
    qry = "INSERT INTO files (file, md5) VALUES ('{fname}', '{md5}')"\
        .format(fname=fname, md5=md5)
    runcmd(qry)
```

As you can see, it simply invokes the **runcmd** function by passing a SQL INSERT statement as a parameter, which inserts a new record into the table.

Next, we have the **setuphashtable** function, which is invoked the first time the script runs, or when the script executes, but the database file does not yet exist.

```
def setuphashtable(fname, md5):
    """Sets Up the Hash Table"""
    createhashtable()
    createhashtableidx()
    inserthashtable(fname, md5)
```

This function is responsible for creating the files table (**createhashtable**), creating the index on the files table (**createhashtableidx**), and inserting the first record into the files table for the first time (**inserthashtable**).

Next, we have the **md5indb** function, which checks whether a specific MD5 hash value exists within the **files** table in the database.

```
def md5indb(fname):
    """Checks if md5 hash value exists in the SQLite DB"""
    items = []
    qry = "SELECT md5 FROM files WHERE file = '" + fname + "'"
    try:
        conn = connectdb()
        if not conn is None:
            if tableexists('files'):
                try:
                    cursor = conn.cursor()
                    cursor.execute(qry)
                    for row in cursor:
                        items.append(row[0])
                except sqlite3.OperationalError as err:
                    print(str(err))
```

```
                cursor.close()
            finally:
                cursor.close()
    except sqlite3.OperationalError as err:
        print(str(err))
        conn.close()
    finally:
        conn.close()
    return items
```

This function is used for checking and determining whether a specific file record in the database needs to be updated, if the file has changed.

Then, the **haschanged** function checks whether a file has changed and invokes the **md5indb** function to see whether the database record corresponding to the changed file needs to be updated in the database with the file's new MD5 hash value.

```
def haschanged(fname, md5):
    """Checks if a file has changed"""
    result = False
    oldmd5 = md5indb(fname)
    numits = len(oldmd5)
    if numits > 0:
        if oldmd5[0] != md5:
            result = True
            updatehashtable(fname, md5)
    else:
        setuphashtable(fname, md5)
    return result
```

When a file has been updated or changed, its MD5 hash value will change. Therefore, it will differ from the MD5 hash value originally stored within the files table in the database. In that case, the existing record in the database table would have to be updated with the new MD5 hash value.

Next, we have the **getfileext** function, which is used for getting the file extension of a specific file.

```
def getfileext(fname):
    """Get the file name extension"""
    return os.path.splitext(fname)[1]
```

This function is useful when there are files with specific extensions that we are not interested in monitoring for changes, such as the database file itself.

Next, we have the **checkfilechanges** function, which is the script's main function.

```python
def checkfilechanges(folder, exclude):
    """Checks for files changes"""
    for subdir, dirs, files in os.walk(folder):
        for fname in files:
            origin = os.path.join(subdir, fname)
            if os.path.isfile(origin):
                fext = getfileext(origin)
                if not fext in exclude:
                    md5 = md5short(origin)
                    if haschanged(origin, md5):
                        print(origin + ' has changed...')
                    else:
                        print(origin + ' has NOT changed...')
```

This function walks through the directory tree indicated by the **`folder`** parameter, and retrieves all the files contained within that folder.

The function checks that each file within the directory does not have an extension that is to be excluded from monitoring. If so, then the file's current MD5 hash value is calculated by calling the **`md5short`** function. That hash value is checked against the hash value stored in the database table. Whether the file has changed or not, this information is displayed and printed out to the console.

The **`checkfilechanges`** function is invoked with two parameters. The first is the **`folder`** to inspect, and the second is an array of file extensions to exclude—files types that we are not interested in monitoring. In this particular example, we are interested in monitoring the folder where the Python script resides, and checking for all file changes, except for the database file itself.

```python
checkfilechanges("./", ['.db'])
```

Let's now run this script through the command prompt or built-in terminal within VS Code by using the following command.

Code Listing 2-j: Command to Execute the Python Script (dfilechanges.py)

```
python dfilechanges.py
```

In my case, I see the following results.

```
PROBLEMS 1   OUTPUT   DEBUG CONSOLE   TERMINAL

PS C:\Projects\Projects\Syncfusion\RPA Succinctly\demo> python3 dfilechanges.py
./dfilechanges.py has NOT changed...
./mergepaths.py has NOT changed...
./readcfg.ini has NOT changed...
./readcfg.py has NOT changed...
./walkdir.py has NOT changed...
./.vscode\settings.json has NOT changed...
PS C:\Projects\Projects\Syncfusion\RPA Succinctly\demo>
```

Figure 2-g: The Execution of the dfilechanges.py Script

Let's now make a small change to the **dfilechanges.py** file, which we can see in the following code as an empty comment (**#**).

```
checkfilechanges("./", ['.db']) #
```

If we run the script again, we should see the following result.

```
PS C:\Projects\Projects\Syncfusion\RPA Succinctly\demo> python3 dfilechanges.py
./dfilechanges.py has changed...
./mergepaths.py has NOT changed...
./readcfg.ini has NOT changed...
./readcfg.py has NOT changed...
./walkdir.py has NOT changed...
./.vscode\settings.json has NOT changed...
PS C:\Projects\Projects\Syncfusion\RPA Succinctly\demo>
```

Figure 2-h: The Execution of the dfilechanges.py Script (File Change Detected)

Notice how the script was able to detect that the **dfilechanges.py** file (which is the script itself) has changed, following that small change made.

Summary

Awesome—we have now explored the foundations of how to perform specific file- and folder-accessing operations, such as reading config files, walking a directory tree, merging file paths, and detecting file changes.

With these scripts and notions acquired, we are now in a good position to explore how to perform more specific file and folder operations, such as copying, moving, deleting, and zipping files and folders, which we will later use to create a configurable backup script.

You'll be able to use that script to automate real-world file backup scenarios for yourself, your organization, and your customers.

Chapter 3 File Operations

Quick intro

Now that we know how to access files and folders, read their contents, get all the files within a directory tree, and find out what files have changed, we can focus on understanding how to perform specific file and folder operations.

We can then use these file and folder operations to design our configurable backup script. Let's jump right in!

Copying files

An essential aspect of any backup script is the ability to copy files. Let's explore how we can copy files in Python.

To do that, let's create a new Python file called **filecopy.py** and add the following code to it.

Code Listing 3-a: Copying Files with Python (filecopy.py)

```
import os
import shutil

def mergepaths(path1, path2):
    pieces = []
    parts1, tail1 = os.path.splitdrive(path1)
    parts2, tail2 = os.path.splitdrive(path2)
    result = path2
    parts1 = tail1.split('\\') if '\\' in tail1 else tail1.split('/')
    parts2 = tail2.split('\\') if '\\' in tail2 else tail2.split('/')
    for pitem in parts1:
        if pitem != '':
            if not pitem in parts2:
                pieces.append(pitem)
    for piece in pieces:
        result = os.path.join(result, piece)
    return result

def filecopy(fname, fld, dest):
    fn = os.path.join(fld, fname)
    d = mergepaths(fld, dest)
    try:
```

```
        if not os.path.exists(d):
            os.makedirs(d)
        shutil.copy(fn, d)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Copied file: ' + fname + ' in ' + fld)

def getallfilesfld(path):
    files = []
    # r=root, d=directories, f=files
    for r, d, f in os.walk(path):
        for file in f:
            if not '.db' in file:
                files.append(os.path.join(r, file))
    return files

def copyall(origin, dest):
    files = getallfilesfld(origin)
    for f in files:
        if os.path.isfile(f):
            o = os.path.dirname(os.path.abspath(f))
            filecopy(os.path.basename(f), o, dest)

origin = os.path.dirname(os.path.abspath(__file__))
copyall(origin, origin + '\\Copied')
```

Let's break this code into smaller parts to understand what is going on.

We first reference the **`shutil`** built-in library, which contains some useful utility functions for working with files in Python, such as copying, moving, and deleting files. We can see that the **`mergepaths`** functions is used, which we explored in the previous chapter.

Next, we find the **`filecopy`** function, which is responsible for calling the **`mergepaths`** function and copying a specific file (**`fname`**) from the origin folder (**`fld`**) to the destination folder (**`dest`**).

```
def filecopy(fname, fld, dest):
    fn = os.path.join(fld, fname)
    d = mergepaths(fld, dest)
    try:
        if not os.path.exists(d):
            os.makedirs(d)
        shutil.copy(fn, d)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
```

```
    finally:

    print('Copied file: ' + fname + ' in ' + fld)
```

If the destination folder does not exist, the **filecopy** function will create it.

Then, we have a function called **getallfilesfld**, which is responsible for returning a list of all the files present within the origin folder.

```
def getallfilesfld(path):
    files = []
    # r=root, d=directories, f=files
    for r, d, f in os.walk(path):
        for file in f:
            if not '.db' in file:
                files.append(os.path.join(r, file))

    return files
```

The **getallfilesfld** function loops through the directory tree of the origin folder (**path**) to get a list of all the files contained within each of the subfolders found under that **path**.

Next, the **copyall** function loops through all the files returned by the **getallfilesfld** function and calls the **filecopy** function for every file (**f**) within that list of **files**.

```
def copyall(origin, dest):
    files = getallfilesfld(origin)
    for f in files:
        if os.path.isfile(f):
            o = os.path.dirname(os.path.abspath(f))

            filecopy(os.path.basename(f), o, dest)
```

The origin for each file (**o**) is determined by returning the directory name (**os.path.dirname**) of the file's (**f**) absolute path (**os.path.abspath**).

The copying process is initiated by calling the **copyall** function as follows.

```
origin = os.path.dirname(os.path.abspath(__file__))

copyall(origin, origin + '\\Copied')
```

The **origin** folder is determined by returning the directory name (**os.path.dirname**) of the absolute path (**os.path.abspath**) of Python script (**__file__**).

The destination folder is the **origin** concatenated with the following string: **'\\Copied'**.

Let's execute the script from the command prompt or built-in terminal within VS Code with the following command.

Code Listing 3-b: Command to Execute the Python Script (filecopy.py)

```
python filecopy.py
```

In my case, the results look as follows.

```
PROBLEMS 2    OUTPUT    DEBUG CONSOLE    TERMINAL

PS C:\Projects\Projects\Syncfusion\RPA Succinctly\demo> python3 filecopy.py
Copied file: dfilechanges.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo
Copied file: filecopy.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo
Copied file: mergepaths.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo
Copied file: readcfg.ini in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo
Copied file: readcfg.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo
Copied file: walkdir.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo
Copied file: settings.json in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\.vscode
```

Figure 3-a: The Execution of the filecopy.py Script (Files Copied)

If I open the folder where the **filecopy.py** file is located on my machine, I can see a **Copied** folder as follows.

Name	Date modified	Type	Size
.vscode	4/13/2020 6:31 PM	File folder	
Copied	4/13/2020 7:14 PM	File folder	
dfilechanges.db	4/13/2020 3:21 PM	Data Base File	12 KB
dfilechanges.py	4/13/2020 3:18 PM	Python File	7 KB
filecopy.py	4/13/2020 6:33 PM	Python File	2 KB
mergepaths.py	4/12/2020 11:14 PM	Python File	1 KB
readcfg.ini	4/12/2020 4:38 PM	Configuration setti...	1 KB
readcfg.py	4/8/2019 9:10 PM	Python File	1 KB
walkdir.py	4/12/2020 10:45 PM	Python File	1 KB

Figure 3-b: The Copied Folder within the Origin Folder

If I now look inside the **Copied** folder, I can see all files copied. If you have followed these exact steps, you should see the same or similar results.

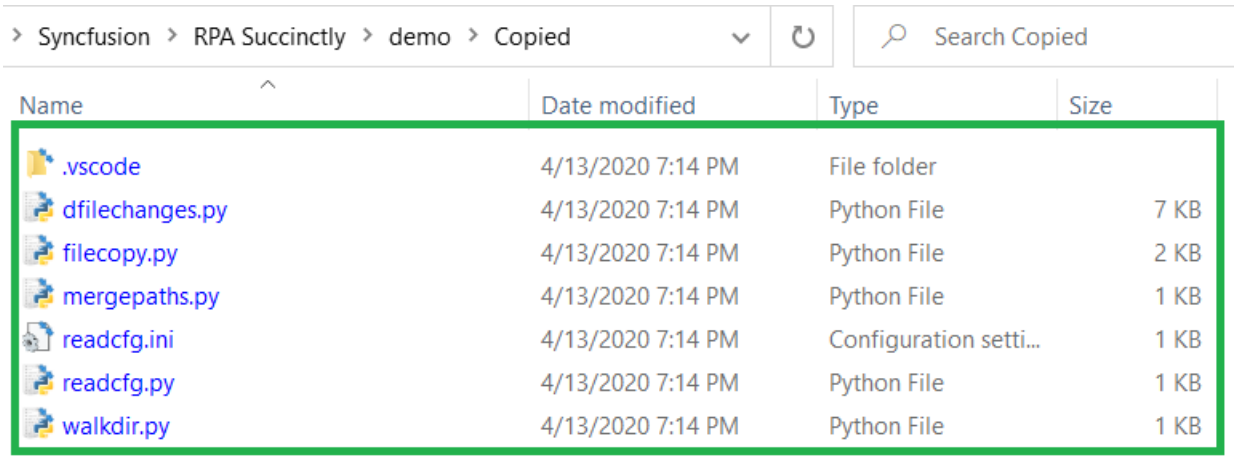


Figure 3-c: The Contents of the Copied Folder

Awesome—we now know how to copy files within any folder. Let's apply this same concept, but for moving files.

Moving files

Moving files is almost identical to the process of copying files. Let's expand the **filecopy.py** script to include this functionality.

We need to include a **`filemove`** function, which looks almost identical to the **`filecopy`** function—the only difference is that we'll use **`shutil.move`** instead of **`shutil.copy`**.

```
def filemove(fname, fld, dest):
    fn = os.path.join(fld, fname)
    d = mergepaths(fld, dest)
    try:
        if not os.path.exists(d):
            os.makedirs(d)
        shutil.move(fn, d)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Moved file: ' + fname + ' in ' + fld)
```

The **`filemove`** function is invoked by the **`moveall`** function, which is almost identical to the **`copyall`** function.

```
def moveall(origin, dest):
    files = getallfilesfld(origin)
    for f in files:
        if os.path.isfile(f):
            o = os.path.dirname(os.path.abspath(f))
```

```
            filemove(os.path.basename(f), o, dest)
```

The rest of the code remains the same. Both the **`filemove`** and **`moveall`** functions can be added to the existing **filecopy.py** codebase.

The **`moveall`** function can be invoked as follows—we can comment out the call to the **`copyall`** function.

```
origin = os.path.dirname(os.path.abspath(__file__))
#copyall(origin, origin + '\\Copied')
moveall(origin, origin + '\\Moved')
```

Let's execute the code and see what results we get. Here are the results from the execution of the code on my machine.

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL

Moved file: readcfg.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo
Moved file: walkdir.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo
Moved file: settings.json in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\.vscode
Moved file: dfilechanges.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Moved file: filecopy.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Moved file: mergepaths.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Moved file: readcfg.ini in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Moved file: readcfg.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Moved file: walkdir.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Moved file: settings.json in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied\.vscode
```

Figure 3-d: The Execution of the filecopy.py Script (Files Moved)

Let's check the origin and **Moved** folders. In the following figure, we can see that the files have been moved from the original folder.

Name	Date modified	Type	Size
.vscode	4/13/2020 7:46 PM	File folder	
Moved	4/13/2020 7:46 PM	File folder	
dfilechanges.db	4/13/2020 3:21 PM	Data Base File	12 KB

Figure 3-e: Files Moved from the Original Location

We can also inspect the content of the **Moved** folder and see the files inside.

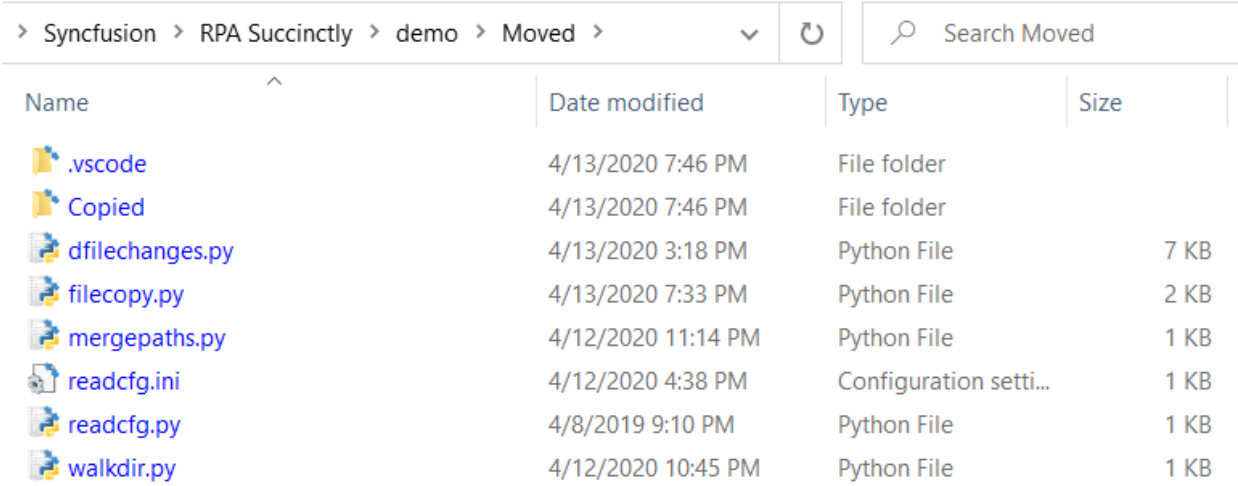


Figure 3-f: Files Found within the Moved Folder

Note: To continue to work and execute the scripts we are working with, please ensure you manually move back the files from the Moved folder to the original directory.

Uncomment the call to the **`copyall`** function and comment out the call to the **`moveall`** function, as follows.

```
origin = os.path.dirname(os.path.abspath(__file__))
copyall(origin, origin + '\\Copied')
#moveall(origin, origin + '\\Moved')
```

Another option is to save all the changes related to moving files onto another Python script called **filemove.py**.

Full copy files script

Here is the code for the full **filecopy.py** Python script.

Code Listing 3-c: Full filecopy.py Script

```
import os
import shutil

def mergepaths(path1, path2):
    pieces = []
    parts1, tail1 = os.path.splitdrive(path1)
    parts2, tail2 = os.path.splitdrive(path2)
    result = path2
    parts1 = tail1.split('\\') if '\\' in tail1 else tail1.split('/')
```

```
    parts2 = tail2.split('\\') if '\\' in tail2 else tail2.split('/')
    for pitem in parts1:
        if pitem != '':
            if not pitem in parts2:
                pieces.append(pitem)
    for piece in pieces:
        result = os.path.join(result, piece)
    return result

def filecopy(fname, fld, dest):
    fn = os.path.join(fld, fname)
    d = mergepaths(fld, dest)
    try:
        if not os.path.exists(d):
            os.makedirs(d)
        shutil.copy(fn, d)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Copied file: ' + fname + ' in ' + fld)

def getallfilesfld(path):
    files = []
    # r=root, d=directories, f=files
    for r, d, f in os.walk(path):
        for file in f:
            if not '.db' in file:
                files.append(os.path.join(r, file))
    return files

def copyall(origin, dest):
    files = getallfilesfld(origin)
    for f in files:
        if os.path.isfile(f):
            o = os.path.dirname(os.path.abspath(f))
            filecopy(os.path.basename(f), o, dest)

origin = os.path.dirname(os.path.abspath(__file__))
copyall(origin, origin + '\\Copied')
```

Full move files script

Here is the code for the full **filemove.py** Python script.

Code Listing 3-d: Full filemove.py Script

```
import os
import shutil

def mergepaths(path1, path2):
    pieces = []
    parts1, tail1 = os.path.splitdrive(path1)
    parts2, tail2 = os.path.splitdrive(path2)
    result = path2
    parts1 = tail1.split('\\') if '\\' in tail1 else tail1.split('/')
    parts2 = tail2.split('\\') if '\\' in tail2 else tail2.split('/')
    for pitem in parts1:
        if pitem != '':
            if not pitem in parts2:
                pieces.append(pitem)
    for piece in pieces:
        result = os.path.join(result, piece)
    return result

def filemove(fname, fld, dest):
    fn = os.path.join(fld, fname)
    d = mergepaths(fld, dest)
    try:
        if not os.path.exists(d):
            os.makedirs(d)
        shutil.move(fn, d)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Moved file: ' + fname + ' in ' + fld)

def getallfilesfld(path):
    files = []
    # r=root, d=directories, f=files
    for r, d, f in os.walk(path):
        for file in f:
            if not '.db' in file:
                files.append(os.path.join(r, file))
    return files
```

```
def moveall(origin, dest):
    files = getallfilesfld(origin)
    for f in files:
        if os.path.isfile(f):
            o = os.path.dirname(os.path.abspath(f))
            filemove(os.path.basename(f), o, dest)

origin = os.path.dirname(os.path.abspath(__file__))
moveall(origin, origin + '\\Moved')
```

Now that we know how to copy and moves files, let's have a look at how we can delete files from folders.

Deleting files

Deleting files is quite similar to copying or moving files. The biggest difference from those two operations is that, when deleting files, there is no need to merge paths, as the deletion operation takes place within the origin folder—there is no destination folder.

Let's see what a deletion script would look like. The logic behind this script is quite straightforward, as you can see.

Code Listing 3-e: Full filedelete.py Script

```
import os
import shutil

def filedelete(fname, fld):
    fn = os.path.join(fld, fname)
    try:
        os.unlink(fn)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Deleted file: ' + fname + ' in ' + fld)

def getallfilesfld(path):
    files = []
    # r=root, d=directories, f=files
    for r, d, f in os.walk(path):
        for file in f:
            if not '.db' in file:
                files.append(os.path.join(r, file))
    return files
```

```
def deleteall(origin):
    files = getallfilesfld(origin)
    for f in files:
        if os.path.isfile(f):
            o = os.path.dirname(os.path.abspath(f))
            filedelete(os.path.basename(f), o)

origin = os.path.dirname(os.path.abspath(__file__))
deleteall(origin + '\\Copied')
```

We have a **`filedelete`** function that invokes the **`os.unlink`** method, which performs the deletion of the file.

The **`getallfilesfld`** function is the same one we used in the **filecopy.py** and **filemove.py** scripts.

The **`deleteall`** function is almost identical to the **`copyall`** or **`moveall`** functions. The major difference is that **`filedelete`** is invoked instead. Another difference is that the **`filedelete`** or **`deleteall`** functions don't require a destination location (**`dest`**) parameter.

Notice that on the call to **`deleteall`**, I'm passing the location of the **Copied** folder as the **`origin`**—which means that the files contained within this folder will be deleted once the script is executed.

To run the script, go to the command prompt or built-in terminal within VS Code and enter the following command.

Code Listing 3-f: Command to Execute the Python Script (filedelete.py)

```
python filedelete.py
```

The execution of the script returns the following output on my machine.

```
PROBLEMS 3    OUTPUT    DEBUG CONSOLE    TERMINAL                                  1: powe

Deleted file: dfilechanges.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Deleted file: filecopy.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Deleted file: filedelete.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Deleted file: filemove.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Deleted file: mergepaths.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Deleted file: readcfg.ini in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Deleted file: readcfg.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Deleted file: walkdir.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Deleted file: settings.json in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied\.vscode
```

Figure 3-g: Files Deleted within the Copied Folder

If I inspect the content of the **Copied** folder, I can see that the files have been deleted.

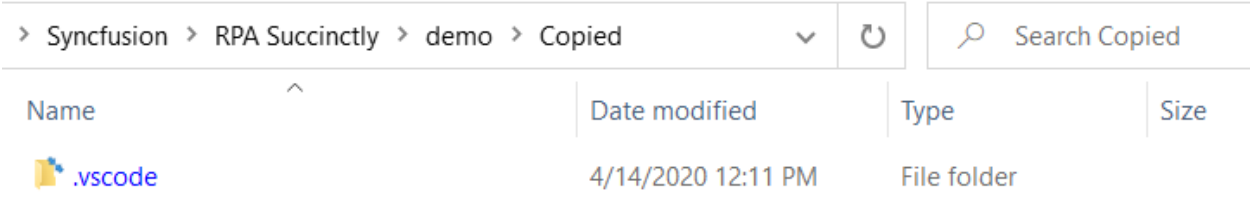


Figure 3-h: Empty Copied Folder (No Files)

Notice that the folder still contains a subfolder, which means that our script was able to delete all the files contained within the **Copied** folder, but it was not able to remove the subfolder.

This is because the **`filedelete`** function only removes files (by calling the **`os.unlink`** method), it doesn't execute any instructions for removing folders.

Deleting folders

Deleting folders with Python is a relatively easy and straightforward process. This is how it can be done.

Code Listing 3-g: Deleting Nested Subfolders with Python - folderdelete.py

```
import os
import shutil

origin = os.path.dirname(os.path.abspath(__file__))
shutil.rmtree(origin + '\\Copied', ignore_errors=False, onerror=None)
```

This script can easily delete any nested subfolder contained within the **Copied** folder and the **Copied** folder itself. Feel free to execute this script to see the result, either from the command prompt or built-in terminal within VS Code, by using the following command.

Code Listing 3-h: Command to Execute the Python Script (folderdelete.py)

```
python folderdelete.py
```

Creating a zip file

When there are many files to back up, it's usually a good idea to create one or more zip files to save space on disk.

In Python, creating archives is a relatively easy and straightforward process. There is a built-in library called zipfile that provides developers with an easy way of working with zip files.

When working with zip files in general, it is recommended to work with file systems that allow for big archives. So in the Windows world, if you are going to be programmatically creating very large zip files with Python, use an NTFS file system drive, rather than a drive with a FAT file system, to avoid zip file creation errors.

Let's see how we can use Python to easily create a zip file with just a few lines of code. Here is the **createzip.py** script.

Code Listing 3-i: Creating a Zip File with Python - createzip.py

```
import os
import time
import zipfile

def filezip(zipf, fname, fld):
    fn = os.path.join(fld, fname)
    try:
        zipf.write(fn)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Zipped file: ' + fname + ' in ' + fld)

def getallfilesfld(path):
    files = []
    # r=root, d=directories, f = files
    for r, d, f in os.walk(path):
        for file in f:
            if not '.db' in file:
                files.append(os.path.join(r, file))
    return files

def addtofilename(fname):
    datet = str(time.strftime("%Y%m%d-%H%M%S"))
    if '%%' in fname:
        fname = fname.replace('%%', datet)
    return fname

def zipall(origin, dest):
    zipf = zipfile.ZipFile(addtofilename(dest), 'w', allowZip64=True)
    files = getallfilesfld(origin)
    for f in files:
        if os.path.isfile(f):
```

```
            o = os.path.dirname(os.path.abspath(f))
            filezip(zipf, os.path.basename(f), o)
    if not zipf is None:
        zipf.close()

origin = os.path.dirname(os.path.abspath(__file__))
zipall(origin + '\\Copied', origin + '\\Copied_%%.zip')
```

We can run this script from the command prompt or built-in terminal within VS Code by running the following command.

Code Listing 3-j: Command to Execute the Python Script (createzip.py)

```
python createzip.py
```

Here are the results from running the script on my machine.

```
PROBLEMS 5    OUTPUT    DEBUG CONSOLE    TERMINAL

Zipped file: dfilechanges.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Zipped file: filecopy.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Zipped file: filedelete.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Zipped file: filemove.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Zipped file: folderdelete.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Zipped file: mergepaths.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Zipped file: readcfg.ini in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Zipped file: readcfg.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
Zipped file: walkdir.py in C:\Projects\Projects\Syncfusion\RPA Succinctly\demo\Copied
```

Figure 3-i: Files Zipped within the Copied Folder

If I now check my origin folder, which is the directory where the Python scripts reside, I can see that a zip file has been created.

Name	Date modified	Type	Size
.vscode	4/14/2020 3:21 PM	File folder	
Copied	4/14/2020 2:51 PM	File folder	
Copied_20200414-152304.zip	4/14/2020 3:23 PM	WinRAR ZIP archive	14 KB
createzip.py	4/14/2020 3:22 PM	Python File	2 KB

Figure 3-j: Zip Archive Created

By checking the contents of the created zip file, I'm able to see the following files—which are the same ones that exist within the **Copied** folder.

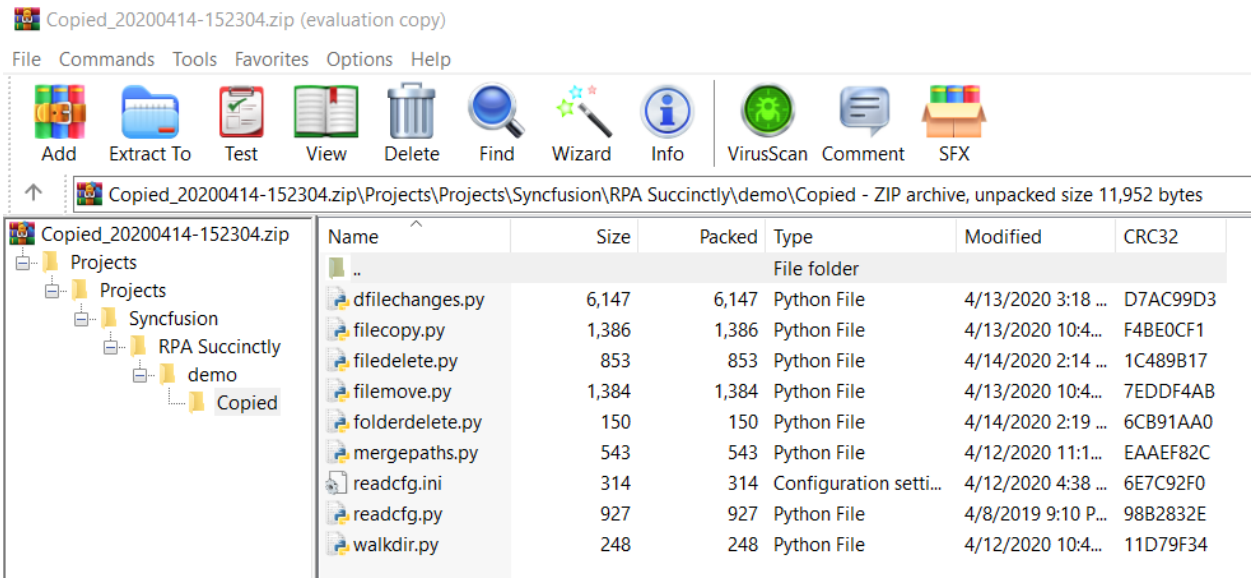


Figure 3-k: Contents of the Zip File Created

Notice that the archive file name contains the date and time when it was created, which in my case is **20200414-152304**.

Another interesting fact is that the original full path for each file zipped is preserved—which is quite useful in case you need to unzip the archive and restore the files to their original folders.

Now that we have verified that this worked, let's have a look at the code in detail, which we can do by exploring each function.

We started by importing the following two Python libraries, which we have not used in previous scripts.

```
import time

import zipfile
```

The **`time`** library was used for generating the date and time that is part of the resultant archive name, whereas the **`zipfile`** library was used for creating the archive itself.

Next, the **`filezip`** function is responsible for adding each of the files from the origin folder (**`fld`**) to the zip archive (**`zipf`**)—one file (**`fname`**) at a time.

```
def filezip(zipf, fname, fld):
    fn = os.path.join(fld, fname)
    try:
        zipf.write(fn)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
```

```
        print('Zipped file: ' + fname + ' in ' + fld)
```

The **filezip** function can do this by calling the **zipf.write** method. The **zipf** object represents the archive being created.

The **getallfilesfld** function, as we have seen from previous examples, just returns a list of the files to process within a specific directory.

Next we have the **addtofilename** function, which adds the date and time when the archive is created by replacing the string **%%** placeholder from the file name.

```
def addtofilename(fname):
    datet = str(time.strftime("%Y%m%d-%H%M%S"))
    if '%%' in fname:
        fname = fname.replace('%%', datet)

    return fname
```

The generated string that replaces the **%%** placeholder has the format **20200414-152304**, which means:

2020 – Year

04 – Month

14 – Day

15 – Hour

23 – Minute

04 – Second

Then we have the **zipall** function, which gets the list of all the files we want to archive (**getallfilesfld**) and adds them to the archive (**dest**).

```
def zipall(origin, dest):
    zipf = zipfile.ZipFile(addtofilename(dest), 'w', allowZip64=True)
    files = getallfilesfld(origin)
    for f in files:
        if os.path.isfile(f):
            o = os.path.dirname(os.path.abspath(f))
            filezip(zipf, os.path.basename(f), o)
    if not zipf is None:

        zipf.close()
```

The archive is created by invoking the method **zipfile.ZipFile**, which returns a reference to the resultant zip file (**zipf**).

The first parameter passed to the zip file is the file name (**`addtofilename(dest)`**). The second parameter is **`'w'`**, which indicates that the zip file has been opened in write mode.

The third parameter, **`allowZip64=True`**, indicates that the zip file will be created using the ZIP64 extension, which is useful when the zip file is larger than 2GB to prevent runtime errors.

For each of the files to process, the **`filezip`** function is invoked—which adds each file to the resultant archive.

Finally, when all the files have been added to the zip file, the archive can be closed, which is done by invoking the **`zipf.close`** method.

As you have seen, zipping files is quite easy to achieve with Python.

Summary

We've now covered all the essential operations to perform on files and folders. Now that we know how to access files and folders, and perform common operations with them, we are ready to create a configurable backup script with Python.

That is what we are going to do in the next chapter. Sounds exciting!

Chapter 4 Backup Script

Quick intro

We've reached a turning point in our RPA journey. We've explored many interesting individual Python scripts that do specific things, but we haven't used them all together as one solution.

In this chapter, we are going to pull together all these concepts and functionalities to build a configurable script that can perform backups on our behalf.

Let's see how this all works.

Putting it all together

Let's create a new Python file called **backup.py**, which will include most of the solution's functionality.

This **backup.py** script will include a reference to the **readcfg.py** script we previously wrote, which can read the configuration details from the **readcfg.ini** file. The **readcfg.ini** file will contain the details of what operations the backup script can execute. The list of operations that the script can execute is variable and may increase or decrease as needed.

First, let's define what libraries our script will use and also reference the **readcfg.ini** file, which we can do as follows.

Code Listing 4-a: The Backup Python Script (backup.py) – Importing Libraries

```
import os
import shutil
import zipfile
import time

from readcfg import readcfg
```

Next, let's add the **`mergepaths`** function, which will come in handy to preserve the original folder locations when we're copying and moving files.

Code Listing 4-b: The Backup Python Script (backup.py) – Function: mergepaths

```
def mergepaths(path1, path2):
    pieces = []
    parts1, tail1 = os.path.splitdrive(path1)
    parts2, tail2 = os.path.splitdrive(path2)
```

```
    result = path2
    parts1 = tail1.split('\\') if '\\' in tail1 else tail1.split('/')
    parts2 = tail2.split('\\') if '\\' in tail2 else tail2.split('/')
    for pitem in parts1:
        if pitem != '':
            if not pitem in parts2:
                pieces.append(pitem)
    for piece in pieces:
        result = os.path.join(result, piece)
    return result
```

We don't need to go over the **mergepaths** function, as we explored it in detail in Chapter 2.

File actions

Next, we need to define a **fileaction** function, which will be responsible for executing the specific file operation (**op['type']**) taking place—this is done by invoking the **cmdfolder** function.

Code Listing 4-c: The Backup Python Script (backup.py) – Function: fileaction

```
def fileaction(op):
    try:
        zipf = startzip(op['dest'], op['type'])
        cmdfolder(zipf, op['origin'], op['dest'], op['type'], op['exclude'])
        if op['type'] == 'd':
            shutil.rmtree(op['origin'])
    except BaseException as err:
        print('ERROR: ' + str(err))
    finally:
        endzip(zipf, op['type'])
```

The **startzip** function only applies when the operation taking place is archival (creating a zip file)—if **op['type'] == 'z'**.

The same applies to the **endzip** function—when the operation taking place is archival—if **op['type'] == 'z'**.

When the operation taking place is a deletion operation—if **op['type'] == 'd'**, then the **shutil.rmtree** method is invoked by passing the origin folder: **op['origin']**.

The **op['dest']** parameter indicates the destination folder, and the **op['exclude']** parameter indicates what files should be excluded from the operation taking place.

Next, let's define the **cmdfolder** function, which will loop through the **origin** folder and get the list of files to process.

Code Listing 4-d: The Backup Python Script (backup.py) – Function: cmdfolder

```
def cmdfolder(zipf, origin, dest, opt, exc):
    for fld, sflds, fnames in os.walk(origin):
        cmdfoldercore(zipf, fld, dest, opt, sflds, fnames, exc)
```

For each file within the origin folder, the **cmdfoldercore** function is invoked, which carries out the work and executes the corresponding file operation, as specified by the **opt** parameter.

Next, we define the **cmdfoldercore** function, which contains two distinct parts: the **cmdsubfolder** and **cmdfiles** functions.

Code Listing 4-e: The Backup Python Script (backup.py) – Function: cmdfoldercore

```
def cmdfoldercore(zipf, fld, dest, opt, sflds, fnames, exc):
    print('Processing folder: ' + fld)
    cmdsubfolder(fld, opt, sflds)
    cmdfiles(zipf, fld, dest, opt, fnames, exc)
```

The **cmdsubfolder** function loops through all the subfolders contained within the origin folder (**fld**). For now, we just print out to the console that we are processing each subfolder (**sf**), but we are not performing any folder operations on any of these subfolders.

This function is useful if, in the future, we want to perform folder operations on each of the subfolders (**sflds**), before performing any file operations on the files contained within those subfolders.

Code Listing 4-f: The Backup Python Script (backup.py) – Function: cmdsubfolder

```
def cmdsubfolder(fld, opt, sflds):
    for sf in sflds:
        print('Processing subfolder: ' + sf + ' in ' + fld)
```

Next, we define the **cmdfiles** function, which loops through all the files to be processed within the origin folder (**fld**) and checks if each file does not have a file extension that is excluded from being processed.

This is done by calling the **isexcluded** function, passing the file name (**fname**) and the excluded file extensions (**exc**). If the file to be processed does not have a file extension that is excluded, then the file (**fname**) is processed according to the type of operation requested (**opt**).

Code Listing 4-g: The Backup Python Script (backup.py) – Function: cmdfiles

```
def cmdfiles(zipf, fld, dest, opt, fnames, exc):
```

```
    for fname in fnames:
        if not isexcluded(fname, exc):
            if opt == 'c':
                filecopy(fname, fld, dest)
            elif opt == 'm':
                filemove(fname, fld, dest)
            elif opt == 'd':
                filedelete(fname, fld, dest)
            elif opt == 'z':
                filezip(zipf, fname, fld)
```

As you can see, depending on the type of operation (**opt**), the corresponding function is invoked: either **filecopy**, **filemove**, **filedelete**, or **filezip**. We'll explore each of these functions next.

Let's define the **filecopy** function, which is responsible for copying a file from the origin folder (**fld**) to the destination directory (**dest**). The copying operation is done by calling the **shutil.copy** method.

Code Listing 4-h: The Backup Python Script (backup.py) – Function: filecopy

```
def filecopy(fname, fld, dest):
    fn = os.path.join(fld, fname)
    d = mergepaths(fld, dest)
    try:
        if not os.path.exists(d):
            os.makedirs(d)
        shutil.copy(fn, d)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Copied file: ' + fname + ' in ' + fld)
```

Notice that the **filecopy** function also creates the destination directory (**d**) if the folder does not exist on disk.

Next, let's define the **filemove** function, which is almost identical to the **filecopy** function. The difference between both functions is that the **shutil.move** method is called instead of **shutil.copy**.

Code Listing 4-i: The Backup Python Script (backup.py) – Function: filemove

```
def filemove(fname, fld, dest):
    fn = os.path.join(fld, fname)
    d = mergepaths(fld, dest)
```

```
    try:
        if not os.path.exists(d):
            os.makedirs(d)
        shutil.move(fn, d)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Moved file: ' + fname + ' in ' + fld)
```

Let's define the **filedelete** function, which, as its name implies, is responsible for file deletion.

Code Listing 4-j: The Backup Python Script (backup.py) – Function: filedelete

```
def filedelete(fname, fld, dest):
    fn = os.path.join(fld, fname)
    try:
        os.unlink(fn)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Deleted file: ' + fname + ' in ' + fld)
```

As we have seen previously, a file deletion operation in Python can be executed by calling the **os.unlink** method, passing the full file name (**fn**) as a parameter.

Zipping files

Let's define the **filezip** function, which is responsible for adding a file (**fname**) to an archive (**zipf**), keeping the original folder structure (**fld**).

Code Listing 4-k: The Backup Python Script (backup.py) – Function: filezip

```
def filezip(zipf, fname, fld):
    fn = os.path.join(fld, fname)
    if fn.lower() != zipf.filename.lower():
        try:
            zipf.write(fn)
        except BaseException as err:
            print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
        finally:
            print('Zipped file: ' + fname + ' in ' + fld)
```

Note: A very important aspect to take into account, which we did not consider previously when creating zip files, is that we do want to write to the archive the zip file itself that is getting created. This is why the following condition is used: *`fn.lower() != zipf.filename.lower()`.*

Let's define the **`addtofilename`** function, which is used to generate an archive name that includes the current date and time when the zip file is created.

We explored this function previously, so no need to review it again.

Code Listing 4-l: The Backup Python Script (backup.py) – Function: addtofilename

```
def addtofilename(fname):
    datet = str(time.strftime("%Y%m%d-%H%M%S"))
    if '%%' in fname:
        fname = fname.replace('%%', datet)
    return fname
```

Next, let's define the **`startzip`** function, which initializes the archive. We have also explored this function. The only difference now is that the destination folder (**`dir`**) does not exist; it is created before the archive is initialized.

Code Listing 4-m: The Backup Python Script (backup.py) – Function: startzip

```
def startzip(dest, opt):
    zipf = None
    if opt == 'z':
        dir = os.path.dirname(dest)
        if not os.path.exists(dir):
            os.makedirs(dir)
        zipf = zipfile.ZipFile(addtofilename(dest), 'w', allowZip64=True)
    return zipf
```

The **`startzip`** function returns a reference to the archive (**`zipf`**), which will later be used by the **`filezip`** and **`endzip`** functions.

Let's define the **`endzip`** function. It closes the archive (**`zipf`**) once all the files have been added to it. We've also explored this function.

Code Listing 4-n: The Backup Python Script (backup.py) – Function: endzip

```
def endzip(zipf, opt):
    if not zipf is None and opt == 'z':
        zipf.close()
```

Main functions

Next, let's define the **`isexcluded`** function, which checks if a file (**`fname`**) has a file extension that should be excluded from a file operation. This is done by checking whether the file extension is included within the exclusion list (**`excl`**).

Code Listing 4-o: The Backup Python Script (backup.py) – Function: isexcluded

```
def isexcluded(fname, excl):
    res = False
    lexc = excl.split(',')
    if len(lexc) > 0:
        if os.path.splitext(fname)[1] in lexc:
            res = True
    return res
```

Let's define the script's main function, called **`runall`**. It's responsible for reading the configuration file (**readcfg.ini**) and then looping through each configuration line stored within the configuration file and executing each operation by invoking the **`fileaction`** or **`ftpaction`** functions.

Code Listing 4-p: The Backup Python Script (backup.py) – Function: runall

```
def runall():
    cfg = os.path.splitext(os.path.basename('readcfg'))[0] + '.ini'
    items = readcfg(cfg)
    for item in items:
        if bool(item) is True:
            if item['type'] == 'f':
                ftpaction(item)
            else:
                fileaction(item)
```

The **`ftpaction`** function I will leave as a teaser for you to finish when you have enough time. The idea behind this function is that you can send files to an FTP server.

Code Listing 4-q: The Backup Python Script (backup.py) – Function: ftpaction

```
#for ftp...
def ftpaction(opts):
    #todo...
    return
```

That's it! Here is the complete and finished source code of our backup script.

Finished backup script

This listing contains the full code of the finished backup script. It includes all the functions we have written throughout this chapter.

Code Listing 4-r: The Backup Python Finished Script (backup.py)

```
import os
import shutil
import zipfile
import time

from readcfg import readcfg

def mergepaths(path1, path2):
    pieces = []
    parts1, tail1 = os.path.splitdrive(path1)
    parts2, tail2 = os.path.splitdrive(path2)
    result = path2
    parts1 = tail1.split('\\') if '\\' in tail1 else tail1.split('/')
    parts2 = tail2.split('\\') if '\\' in tail2 else tail2.split('/')
    for pitem in parts1:
        if pitem != '':
            if not pitem in parts2:
                pieces.append(pitem)
    for piece in pieces:
        result = os.path.join(result, piece)
    return result

def startzip(dest, opt):
    zipf = None
    if opt == 'z':
        dir = os.path.dirname(dest)
        if not os.path.exists(dir):
            os.makedirs(dir)
        zipf = zipfile.ZipFile(addtofilename(dest), 'w', allowZip64=True)
    return zipf

def endzip(zipf, opt):
    if not zipf is None and opt == 'z':
        zipf.close()

def isexcluded(fname, excl):
    res = False
    lexc = excl.split(',')
```

```
    if len(lexc) > 0:
        if os.path.splitext(fname)[1] in lexc:
            res = True
    return res

def fileaction(op):
    try:
        zipf = startzip(op['dest'], op['type'])
        cmdfolder(zipf, op['origin'], op['dest'], op['type'], op['exclude'])
        if op['type'] == 'd':
            shutil.rmtree(op['origin'])
    except BaseException as err:
        print('ERROR: ' + str(err))
    finally:
        endzip(zipf, op['type'])

def cmdfolder(zipf, origin, dest, opt, exc):
    for fld, sflds, fnames in os.walk(origin):
        cmdfoldercore(zipf, fld, dest, opt, sflds, fnames, exc)

def cmdfoldercore(zipf, fld, dest, opt, sflds, fnames, exc):
    print('Processing folder: ' + fld)
    cmdsubfolder(fld, opt, sflds)
    cmdfiles(zipf, fld, dest, opt, fnames, exc)

def cmdsubfolder(fld, opt, sflds):
    for sf in sflds:
        print('Processing subfolder: ' + sf + ' in ' + fld)

def cmdfiles(zipf, fld, dest, opt, fnames, exc):
    for fname in fnames:
        if not isexcluded(fname, exc):
            if opt == 'c':
                filecopy(fname, fld, dest)
            elif opt == 'm':
                filemove(fname, fld, dest)
            elif opt == 'd':
                filedelete(fname, fld, dest)
            elif opt == 'z':
                filezip(zipf, fname, fld)

def filecopy(fname, fld, dest):
    fn = os.path.join(fld, fname)
    d = mergepaths(fld, dest)
```

```
    try:
        if not os.path.exists(d):
            os.makedirs(d)
        shutil.copy(fn, d)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Copied file: ' + fname + ' in ' + fld)

def filemove(fname, fld, dest):
    fn = os.path.join(fld, fname)
    d = mergepaths(fld, dest)
    try:
        if not os.path.exists(d):
            os.makedirs(d)
        shutil.move(fn, d)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Moved file: ' + fname + ' in ' + fld)

def filedelete(fname, fld, dest):
    fn = os.path.join(fld, fname)
    try:
        os.unlink(fn)
    except BaseException as err:
        print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
    finally:
        print('Deleted file: ' + fname + ' in ' + fld)

def filezip(zipf, fname, fld):
    fn = os.path.join(fld, fname)
    if fn.lower() != zipf.filename.lower():
        try:
            zipf.write(fn)
        except BaseException as err:
            print('ERROR: ' + fname + ' in ' + fld + ' with: ' + str(err))
        finally:
            print('Zipped file: ' + fname + ' in ' + fld)

def addtofilename(fname):
    datet = str(time.strftime("%Y%m%d-%H%M%S"))
    if '%%' in fname:
        fname = fname.replace('%%', datet)
```

```
    return fname

def runall():
    cfg = os.path.splitext(os.path.basename('readcfg'))[0] + '.ini'
    items = readcfg(cfg)
    for item in items:
        if bool(item) is True:
            if item['type'] == 'f':
                ftpaction(item)
            else:
                fileaction(item)

#for ftp...
def ftpaction(opts):
    #todo...
    return

runall()
```

As you can see, the script gets initialized by invoking the **`runall`** function, and it's quite short for what it does.

Before you attempt to execute the script from the command prompt or built-in terminal in VS Code, make sure you have properly defined the operations you want to execute within the **readcfg.ini** file.

In my case, I'm going to work with the files I have under my **C:\Temp\ForTesting** folder, which is specifically for testing. You can use any folder on your machine that works best for you.

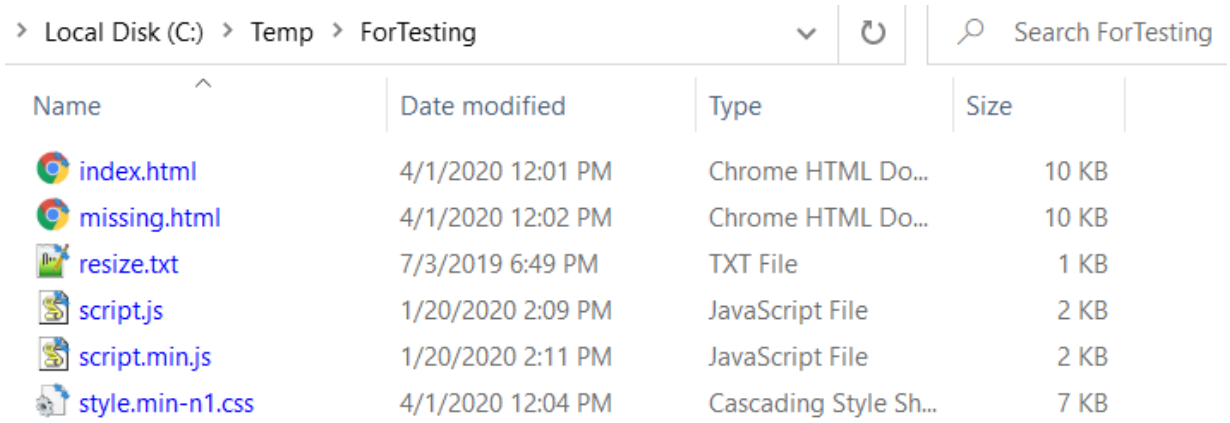


Figure 4-a: Backup Script Testing Files

Within my **readcfg.ini** file, I have defined the operations that I want my backup script to execute.

Code Listing 4-s: The Contents of the readcfg.ini File

```
C:\Temp\ForTesting|C:\Temp\ForTesting\Test_%%.zip|z
C:\Temp\ForTesting|C:\Temp\ForTesting2|c
C:\Temp\ForTesting||d
C:\Temp\ForTesting2|C:\Temp\ForTesting|c
```

Note: You will need to define the settings of your readcfg.ini according to the folders where your test files are located and will be backed up to.

As you can see, I've defined a series of operations I want to execute in order. The first operation will create a zip file from all the files contained within the **C:\Temp\ForTesting** folder.

Then, all the files found within the **C:\Temp\ForTesting** folder will be copied to another folder, in my case **C:\Temp\ForTesting2**.

After that, the files contained within the **C:\Temp\ForTesting** folder will be deleted. Finally, the files contained within the **C:\Temp\ForTesting2** folder will be copied to **C:\Temp\ForTesting**.

The backup script can be executed from the command prompt or with the built-in terminal in VS Code by using the following command.

Code Listing 4-t: Command to Execute the Python Script (backup.py)

```
python backup.py
```

After the script has executed, I can see the following results on my machine.

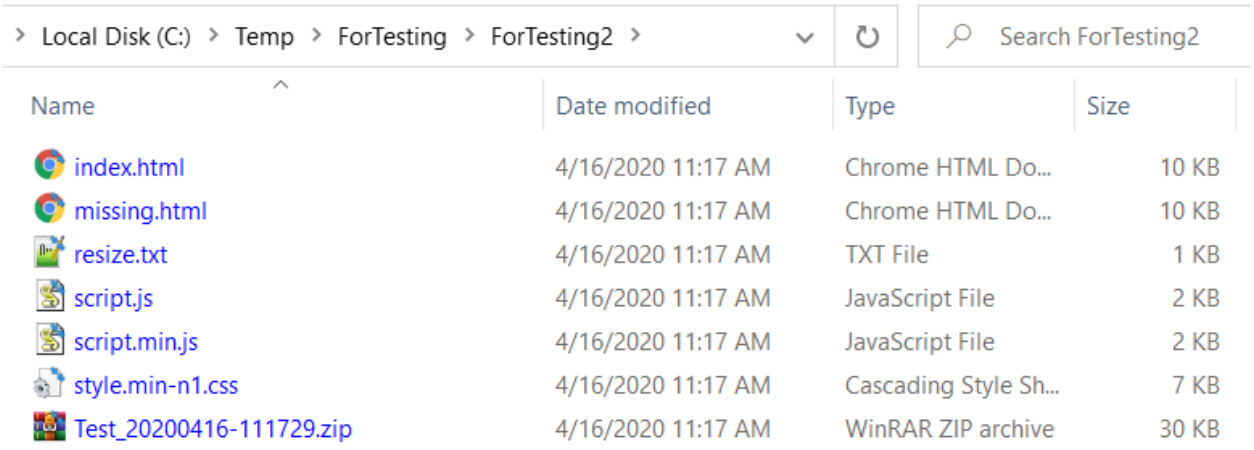


Figure 4-b: Backup Script Execution Results – Part 1

If you pay close attention, you will see that the script merged the origin folder path (**C:\Temp\ForTesting**) with the destination path (**C:\Temp\ForTesting2**) into a new path: **C:\Temp\ForTesting\ForTesting2**.

If you inspect further, you will notice that another new directory structure (merged path) also came out of that sequence of operations, which we can see in the following figure.

This new path is: **C:\Temp\ForTesting2\ForTesting**.

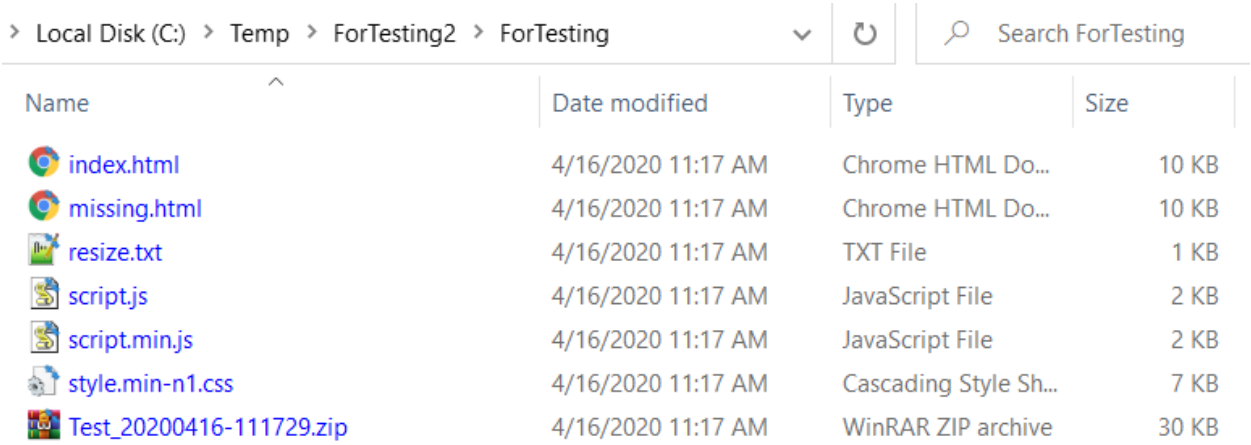


Figure 4-c: Backup Script Execution Results – Part 2

This would seem to be a bit odd, but at the same time, it is simply a consequence of the code we wrote—specifically the usage of the **`mergepaths`** function when file copying and moving operations were executed.

You might be wondering why we need to use a function like **`mergepaths`** if the file path outcome, in this case, is not what we expected. The answer is simple. Using a function like **`mergepaths`** is useful when we want to preserve within the destination folder the same original folder structure. The usefulness of this approach is only really applicable when we back up files to a different drive. Ultimately, you want to back up your valuable data to another drive, and not leave it on the same drive where it already exists. In other words, you want to have your eggs in more than one basket.

If your day-to-day drive fails, you have your information backed up on another drive. By using the **`mergepaths`** function—which keeps the original folder structure of the backed up files—you can easily know where the files originated from, in case you want to recreate that same structure again on a new drive or computer. To see this in action, I'll make some modifications to my **readcfg.ini** file.

I've plugged in an external USB drive on my machine, which will serve as my backup drive, and I'll direct my destinations paths (within **readcfg.ini)** to point to this drive instead of the C drive.

Code Listing 4-u: The Modified Contents of the readcfg.ini File

```
C:\Temp\ForTesting|D:\Backup\Test_%%.zip|z
C:\Temp\ForTesting|D:\Backup|c
```

Notice that the origin folder (**C:\Temp\ForTesting**) is now being backed up to the **D:\Backup** directory, which resides within the external USB drive I've plugged into my computer.

If I now execute the script again, I get the following result. How cool is that?

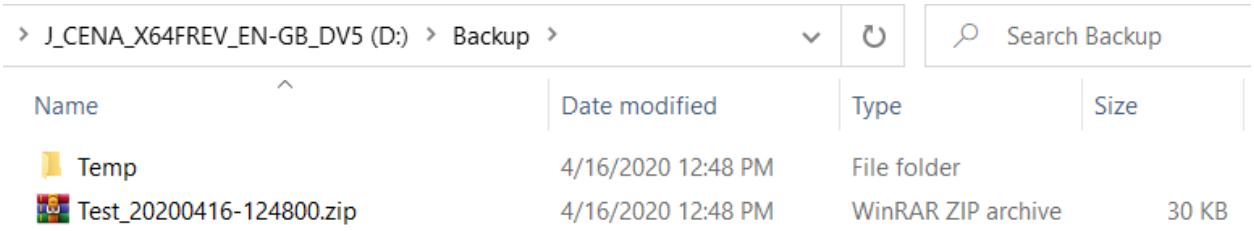


Figure 4-d: Backup Script Execution Updated Results

Notice that the archive has been created within the **D:\Backup** destination folder, and a full copy of the original **C:\Temp\ForTesting** folder has been made, preserving the original folder structure. This is where the usefulness of the **`mergepaths`** function becomes visible.

Summary

Throughout this chapter, we have built a configurable backup script that can execute multiple operations and can be applied to various scenarios. We were able to do this by putting together all the knowledge we gathered in the previous chapters.

We also left open the possibility of adding FTP functionality to the script, which would be a great feature to have. Beyond that, the script could also be further enhanced by adding the file changes tracking features that we previously explored, so that only files that have recently changed can be incrementally backed up into separate zip files.

Please feel free to expand this script and add any extra logic that fits your requirements. If you have the time and enthusiasm to pursue this, go for it. I'd love to see what you come up with.

Chapter 5 Assisted Data Entry Automation

Quick intro

One of the fundamental challenges of business process automation, in general, is how to efficiently extract data from semi-structured or unstructured sources, such as PDF files, and use that data to automatically perform data entry on other systems, with little or no need for human intervention.

This is what we are going to explore in this chapter. We are going to create a Python script that can read PDF form documents, extract specific data from them, and automatically fill in the same data on a web-based system.

To understand this process better, let's have a look at the following diagram.

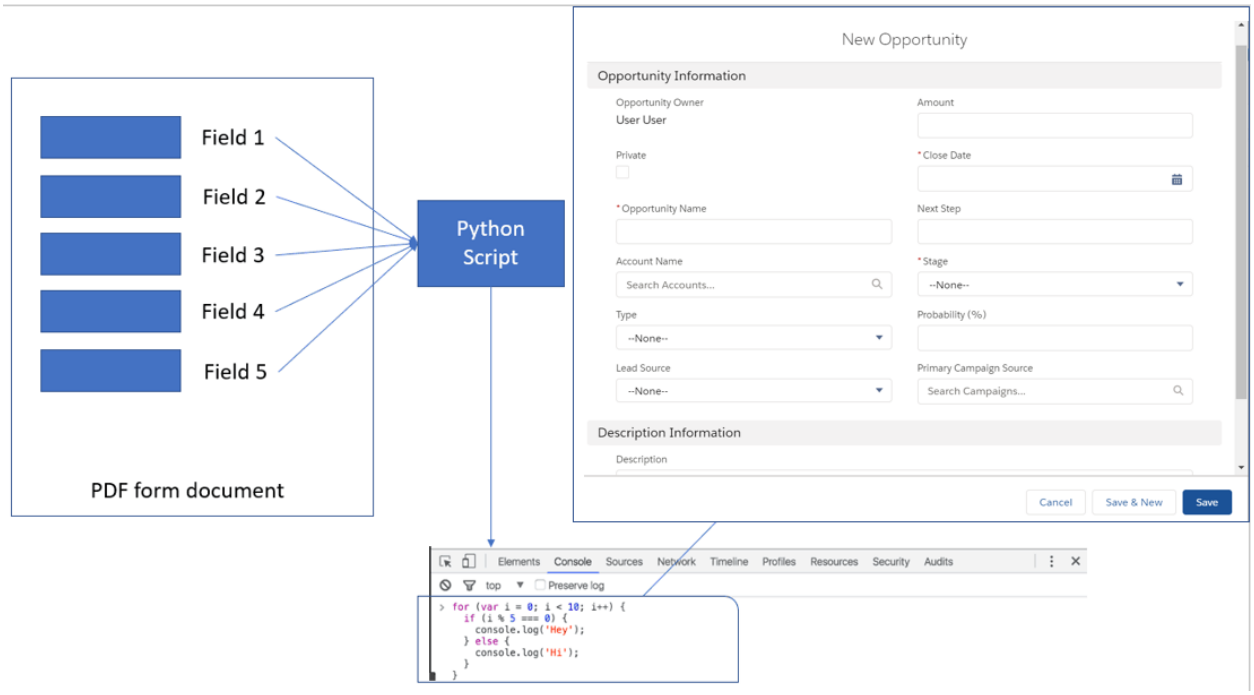


Figure 5-a: PDF Form Data Extraction to Automatic Data Entry Workflow

We can see that our data source is going to be made up of PDF form documents. These are PDF files that have fields that can be filled in by users.

Each PDF file is going to be read by a Python script (which we will create) that is able to extract each field value and generate JavaScript code that can be pasted into the browser's console, which can automatically fill in the values of elements for a specific webpage. In our case, it is going to be a Salesforce Opportunity webpage.

In essence, each PDF form document is going to represent a new Salesforce opportunity that someone would have to manually fill in.

This process would normally be done manually by going through the list of PDFs, opening one PDF at a time, copying each of the field values from the PDF to the webpage, and saving the information to Salesforce.

Carrying out such a manual process is not only prone to errors, but it can be very time-consuming if the number of documents and fields per document to process is high. It can result in a very tedious and repetitive task for any user.

The goal is to facilitate the process for users performing manual data entry by reading all the PDF form documents present on a given folder and, for each document, generating a text file containing JavaScript code that can simply be pasted into the browser's console to fill in the values of the Salesforce Opportunity web form elements.

The Python script will generate the JavaScript code, which then the user will manually copy and paste into the browser's console. This is why the chapter is called Assisted Data Entry Automation—there is still a human element to the process.

We could automate the whole process by creating a browser script with Selenium WebDriver or another browser automation tool, which would be able to open a new Salesforce Opportunity webpage and enter the data for each PDF form document processed. However, we won't take this example that far. Nevertheless, if you want to take the Python script to that level of browser automation, you are encouraged to do so, and I'd love to hear about it.

Reading PDF forms

The PDF file format is a versatile document format. It is one of the world's most-used file formats, along with Microsoft Excel and Word. The world runs on these three essential file formats. This means that more data is contained within these types of documents than probably in most databases.

The PDF file format is versatile because it supports various types of content, such as scanned documents, standard text printed content, and content with fields. In this case, we will focus on how to read PDF forms, which corresponds to content with fields that are part of a PDF file.

To be able to read PDF form documents, we need to install the PyPDF2 library, which can be installed from the command prompt or built-in terminal within VS Code, using the following command.

Code Listing 5-a: Command to Install the PyPDF2 Library

```
pip install PyPDF2
```

With some Python 3.X installations, **`pip3`** can also be used instead of **`pip`**. Both do the same job. Once the library has been installed, we can start writing our script.

The command shown in Code Listing 5-a will install the latest version of PyPDF2 (version 1.26.0), which is very stable and reliable.

Let's start by importing the modules and libraries that our script will require, which we can see in the following listing.

Code Listing 5-b: Importing the Required Modules

```
import os
import sys

import PyPDF2
from collections import OrderedDict
```

We will be using an ordered dictionary (**`OrderedDict`**) to return the list of fields read from a PDF form document. We can read any fields within a PDF form document by using the following function.

Code Listing 5-c: Reading PDF Fields – Part 1

```
def readfields(obj, t=None, res=None, fo=None):
    fa = {'/FT': 'Field Type',
    '/Parent': 'Parent',
    '/T': 'Field Name',
    '/TU': 'Alternate Field Name',
    '/TM': 'Mapping Name',
    '/Ff': 'Field Flags',
    '/V': 'Value',
    '/DV': 'Default Value'}

    if res is None:
        res = OrderedDict()
        items = obj.trailer["/Root"]
        if "/AcroForm" in items:
            t = items["/AcroForm"]
        else:
            return None
    if t is None:
        return res
    obj._checkKids(t, res, fo)
    for attr in fa:
        if attr in t:
            obj._buildField(t, res, fo, fa)
            break
    if "/Fields" in t:
        flds = t["/Fields"]
        for f in flds:
            fld = f.getObject()
```

```
            obj._buildField(fld, res, fo, fa)
    return res
```

Let's explore this function in detail. We start by specifying the AcroForm (Adobe Acrobat form) field attributes that are used within PDF form documents.

```
fa = {'/FT': 'Field Type', '/Parent': 'Parent',
    '/T': 'Field Name', '/TU': 'Alternate Field Name',
    '/TM': 'Mapping Name', '/Ff': 'Field Flags',
    '/V': 'Value', '/DV': 'Default Value'}
```

These attributes are checked whenever the function runs into a potential field and needs to verify the type of field it is and whether it has any value that can be read.

The first step in this process is finding the PDF document root (**/Root**) and checking whether it contains a form (**/AcroForm**).

```
if res is None:
    res = OrderedDict()
    items = obj.trailer["/Root"]
    if "/AcroForm" in items:
        t = items["/AcroForm"]
    else:
        return None
if t is None:
    return res
```

If the PDF document contains a form, then a tree object with the potential fields is returned, which is then inspected. This is done with the following code.

```
obj._checkKids(t, res, fo)
for attr in fa:
    if attr in t:
        obj._buildField(t, res, fo, fa)
        break
if "/Fields" in t:
    flds = t["/Fields"]
    for f in flds:
        fld = f.getObject()
        obj._buildField(fld, res, fo, fa)
```

The fields that are identified as PDF form fields are returned in an ordered dictionary, which contains each field with its respective value as a set of key-value pairs. This is what the function returns.

```
return res
```

By calling the **`readfields`** function, we can read the PDF form fields of any particular PDF that contains a form that follows the AcroForm file specification.

However, to be able to do that, we need to first open the PDF file and read its content, which we can do with the following function.

Code Listing 5-d: Reading PDF Fields – Part 2

```
def getfields(infile):
    infile = PyPDF2.PdfFileReader(open(infile, 'rb'))
    fields = readfields(infile)
    return OrderedDict((k, v.get('/V', '')) for k, v in fields.items())
```

This function essentially opens the PDF file (**`infile`**) in read-only binary mode (**`rb`**) and reads its full content by calling the **`PdfFileReader`** method from the **PyPDF2** library.

Once that full content has been read, it is passed to the **`readfields`** function, which checks if there's a form embedded within the PDF, and if so, extracts each of the fields contained within.

Those fields are then returned in a new ordered dictionary as key-value pairs, such as the following.

```
OrderedDict([('Field1', '0001'), ('Field2', '0002'), ('Field3', '0003')])
```

Generating the browser script

Now that we know how to extract fields from PDF form documents, the most logical and straightforward way to assign field values extracted from those PDF documents to fields on webpages is to use JavaScript—the native language of browsers.

We can do this by using Python to generate for each PDF file a script that can be executed within the browser's console, employing copy and paste to automatically fill in field values on a webpage.

Code Listing 5-e: Generating the Browser Script – Part 1

```
def rselects(fn):
    lst = []
    with open(fn, 'r') as fh:
        for l in fh:
            lst.append(l.rstrip(os.linesep))
    return lst

def sselectitm(l, k, v):
    l.append('function setItem(s, v) {')
    l.append('for (var i = 0; i < s.options.length; i++) {')
    l.append('if (s.options[i].text == v) {')
    l.append('s.options[i].selected = true;')
    l.append('return;')
    l.append('}')
    l.append('}')
    l.append('}')
    l.append('setItem(document.getElementById("' + k + '"), "' + v + '");')

def bscript(si, items, pdff):
    if pdff:
        of = os.path.splitext(pdff)[0] + '.txt'
        lns = []
        for k, v in items.items():
            if (k in si):
                sselectitm(lns, k, v)
                print('Selectable: ' + k + ' -> ' + v)
            else:
                lns.append(
                    "document.getElementById('" + k + "').value = '" + v + "';\n")
                print('Normal: ' + k + ' -> ' + v)
        scriptf = open(of, 'w')
        scriptf.writelines(lns)

    scriptf.close()
```

The first step consists of identifying which of the webpage fields that will be filled in by the generated browser script are text fields, and which are selectable (drop-down) fields.

First, we have the **`rselects`** function. It reads a file called **selects.ini** that will contain the IDs of the fields on the webpage, which are selectable (drop-down) fields.

This function returns a list (**`lst`**) of all the lines read from the **selects.ini** file.

```python
def rselects(fn):
    lst = []
    with open(fn, 'r') as fh:
        for l in fh:
            lst.append(l.rstrip(os.linesep))
    return lst
```

To find the IDs of the fields on the webpage, we'll need to use our browser's dev tools. For a quick overview on how to find field IDs using your browser, please check out this short video.

For more in-depth information on how to use Chrome Developer Tools to inspect web elements, this free crash course is a great resource for diving deeper into this subject.

Imagine that the webpage you want the generated browser script to fill in has a selectable field that looks as follows.

```html
<div class="form-group"><p>Selectable Field<span></span></p>
    <select name="CONCEPT" id="CONCEPT" style="width:250px">
        <option value="This is a Concept">This is a concept</option>
        <option value="This is another concept">This is Another concept</option>
    </select>

</div>
```

We would need our Python script to generate a JavaScript function (**`setItem`**) that would be responsible for selecting the item that corresponds to the value of **`v`** (extracted from the PDF) on the selectable field (where **`s`** is the selectable field with its various options).

```javascript
function setItem (s, v) {

    for (var i = 0; i < s.options.length; i++) {

        if (s.options[i].text == v) {
            s.options[i].selected = true;
            return;
        }
    }
}
```

We would need our generated browser script to invoke the **`setItem`** function as follows.

```javascript
setItem(document.getElementById("CONCEPT"), "This is a Concept");
```

This is exactly what the **`sselectitm`** function does within our Python script.

```python
def sselectitm(l, k, v):
    l.append('function setItem(s, v) {')
    l.append('for (var i = 0; i < s.options.length; i++) {')
```

```
    l.append('if (s.options[i].text == v) {')
    l.append('s.options[i].selected = true;')
    l.append('return;')
    l.append('}')
    l.append('}')
    l.append('}')

    l.append('setItem(document.getElementById("' + k + '"), "' + v + '");')
```

The **sselectitm** function essentially creates an instance of the **setItem** JavaScript function for each selectable field required and assigns the correct value to it. Then, the complete browser script is generated (including regular text fields found on the webpage) by the **bscript** function.

```
def bscript(si, items, pdff):
    if pdff:
        of = os.path.splitext(pdff)[0] + '.txt'
        lns = []
        for k, v in items.items():
            if (k in si):
               sselectitm(lns, k, v)
               print('Selectable: ' + k + ' -> ' + v)
            else:
               lns.append(
               "document.getElementById('" + k + "').value = '" + v + "';\n")
               print('Normal: ' + k + ' -> ' + v)
        scriptf = open(of, 'w')
        scriptf.writelines(lns)

        scriptf.close()
```

The **bscript** function loops through the ordered dictionary of extracted PDF fields (**items**) and checks if any of those PDF field names is a selectable field (**si**). If so, the **sselectitm** function is invoked.

For each of those PDF field names that is not a selectable field, the **bscript** function simply creates a **document.getElementById** instruction with the name of the PDF field (which needs to be the same as the ID of the webpage element to fill in), assigning it the value extracted from the PDF field.

Note: for this to work, all the PDF form fields extracted must have the same names as the IDs of the fields or elements of the webpage to be filled in. These names are case sensitive.

The following figure shows side by side the destination webpage (in my case a Salesforce opportunity), which needs to be filled in by the generated script (seen on the left-hand side), and the PDF form document that contains the information that will be extracted by the Python script (seen on the right-hand side).

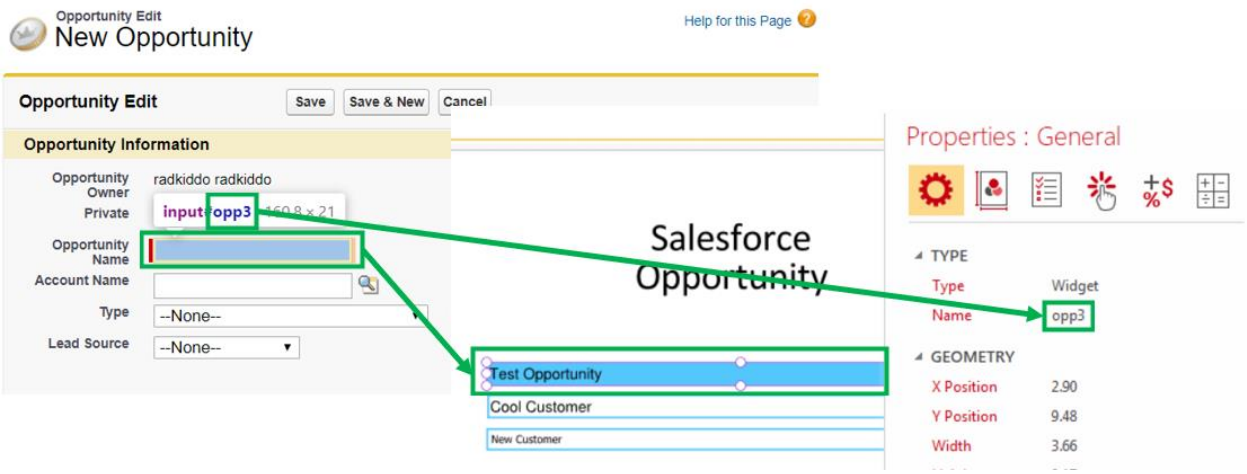


Figure 5-b: Webpage Field Name = PDF Form Field Name

Each of the fields on the webpage that needs to be filled in must be inspected, and their IDs retrieved. Selectable (drop-down) fields must be identified and saved in the **selects.ini** file, one field per line. For this particular example, the following web elements correspond to selectable fields and need to be included in the **selects.ini** file.

```
opp5
opp11
opp6
```

The fields on the PDF form document must be named as the webpage IDs that will be filled in. There must be a one-to-one match between PDF field names and webpage element IDs.

This means that you should design your PDF form documents so that fields have the same names as their corresponding webpage element IDs that you want the script to fill in.

You can design PDF form documents with various desktop applications, such as Adobe Acrobat, PDF Element, PDF Desktop, or others.

To bind this all together, we need to define a function that executes these functions in the right sequence, which is what the **`run`** function does.

Code Listing 5-f: Generating the Browser Script – Part 2

```
def run(args):
    try:
        si = rselects('selects.ini')
        if len(args) == 2:
            pdf_file_name = args[1]
            items = getfields(pdf_file_name)
            bscript(si, items, pdf_file_name)
        else:
            files = [f for f in os.listdir('.')
```

```
            if os.path.isfile(f) and f.endswith('.pdf')]
            for f in files:
                items = getfields(f)
                bscript(si, items, f)
    except BaseException as msg:
        print('An error occured... :( ' + str(msg))
```

This function receives a command line parameter (**`args`**) that can be the name of a specific PDF file to process.

If the name of a PDF file is specified (**`pdf_file_name`**) when executing the Python script—if **`len(args) == 2`**, then (as we will see shortly) the script only executes the fields extraction operation (**`getfields`**) on that specific PDF and generates the corresponding browser script for that PDF (as a .txt file).

If no PDF file is specified when executing the Python script, then for all the PDF files found within the same folder as the Python script, the fields extraction operation on each PDF file will take place, and for each PDF file, a corresponding browser script (with the .txt extension) will be generated.

Code Listing 5-g: Invoking the run Function

```
if __name__ == '__main__':
    run(sys.argv)
```

The AutoForm script

So, let's put all the functions we've explored so far throughout this chapter into a single code block, which we will call **autoform.py**.

Code Listing 5-h: The AutoForm Script

```
import os
import sys

import PyPDF2
from collections import OrderedDict

def readfields(obj, t=None, res=None, fo=None):
    fa = {'/FT': 'Field Type',
    '/Parent': 'Parent',
    '/T': 'Field Name',
    '/TU': 'Alternate Field Name',
    '/TM': 'Mapping Name',
```

```
    '/Ff': 'Field Flags',
    '/V': 'Value',
    '/DV': 'Default Value'}

    if res is None:
        res = OrderedDict()
        items = obj.trailer["/Root"]
        if "/AcroForm" in items:
            t = items["/AcroForm"]
        else:
            return None
    if t is None:
        return res
    obj._checkKids(t, res, fo)
    for attr in fa:
        if attr in t:
            obj._buildField(t, res, fo, fa)
            break
    if "/Fields" in t:
        flds = t["/Fields"]
        for f in flds:
            fld = f.getObject()
            obj._buildField(fld, res, fo, fa)
    return res

def getfields(infile):
    infile = PyPDF2.PdfFileReader(open(infile, 'rb'))
    fields = readfields(infile)
    return OrderedDict((k, v.get('/V', '')) for k, v in fields.items())

def rselects(fn):
    lst = []
    with open(fn, 'r') as fh:
        for l in fh:
            lst.append(l.rstrip(os.linesep))
    return lst

def bscript(si, items, pdff):
    if pdff:
        of = os.path.splitext(pdff)[0] + '.txt'
        lns = []
        for k, v in items.items():
            if (k in si):
                sselectitm(lns, k, v)
```

```
                print('Selectable: ' + k + ' -> ' + v)
            else:
                lns.append(
                    "document.getElementById('" + k + "').value = '" + v + "';\n")
                print('Normal: ' + k + ' -> ' + v)
        scriptf = open(of, 'w')
        scriptf.writelines(lns)
        scriptf.close()

def sselectitm(l, k, v):
    l.append('function setItem(s, v) {')
    l.append('for (var i = 0; i < s.options.length; i++) {')
    l.append('if (s.options[i].text == v) {')
    l.append('s.options[i].selected = true;')
    l.append('return;')
    l.append('}')
    l.append('}')
    l.append('}')
    l.append('setItem(document.getElementById("' + k + '"), "' + v + '");')

def run(args):
    try:
        si = rselects('selects.ini')
        if len(args) == 2:
            pdf_file_name = args[1]
            items = getfields(pdf_file_name)
            bscript(si, items, pdf_file_name)
        else:
            files = [f for f in os.listdir('.')
            if os.path.isfile(f) and f.endswith('.pdf')]
            for f in files:
                items = getfields(f)
                bscript(si, items, f)
    except BaseException as msg:
        print('An error occured... :( ' + str(msg))

if __name__ == '__main__':
    run(sys.argv)
```

Before we run the script and try it with the PDF form document I have designed, I would suggest that you quickly sign up for a free trial Salesforce account, so you can follow along and run the test yourself.

Signing up for a free trial Salesforce account is easy; you will be asked to fill in a small registration form, which takes less than a minute to complete. Once you have submitted your details, you will be immediately redirected to the Salesforce solution, which you can start experimenting with.

When you are in, click on the avatar icon and choose the option **Switch to Salesforce Classic**, as shown in Figure 5-c, which, for this demo, is easier to navigate and use.

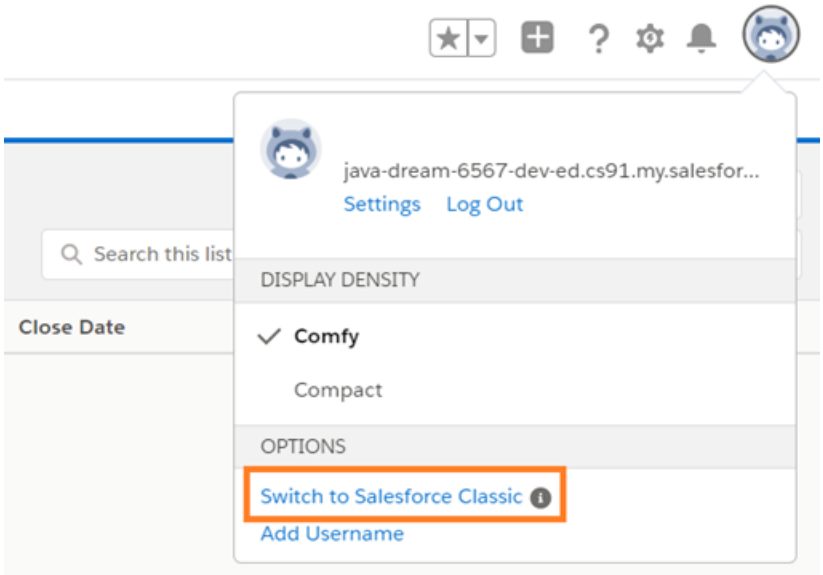


Figure 5-c: Switch to Salesforce Classic Option

When you are in the Salesforce Classic mode, click the **Opportunities** tab to create a new opportunity. You will see the following screen.

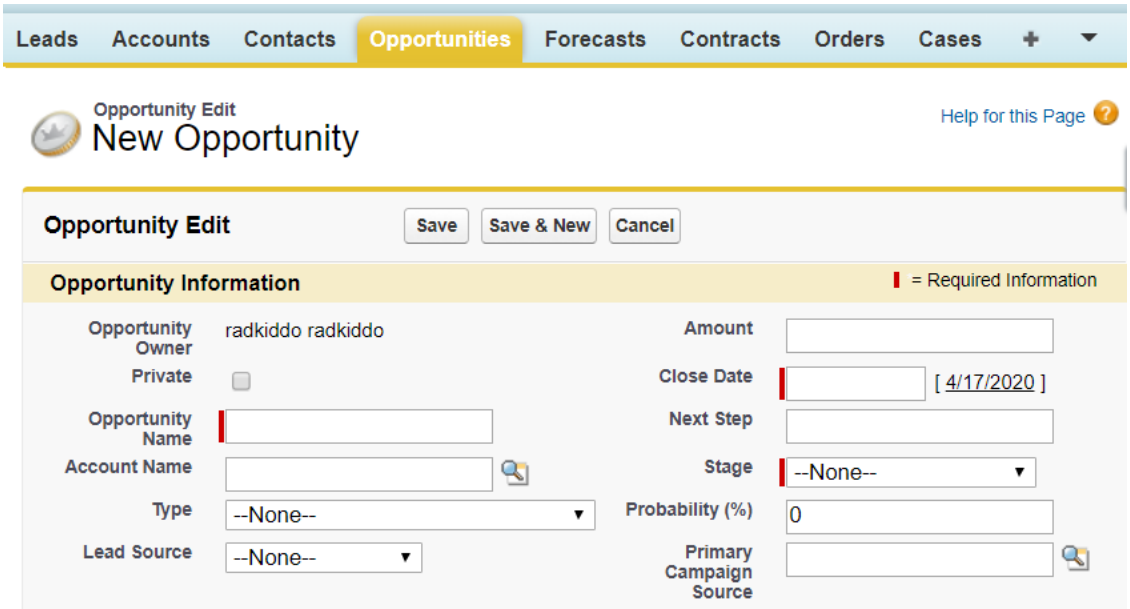


Figure 5-d: New Salesforce Opportunity Webpage (Classic Mode)

Now it's time to run the script. But before we do that, make sure you have downloaded the sample PDF form document for this demo, from this URL.

The sample PDF form document contains the following details seen in Figure 5-3. Make sure you place this PDF file within the same folder as the Python script.

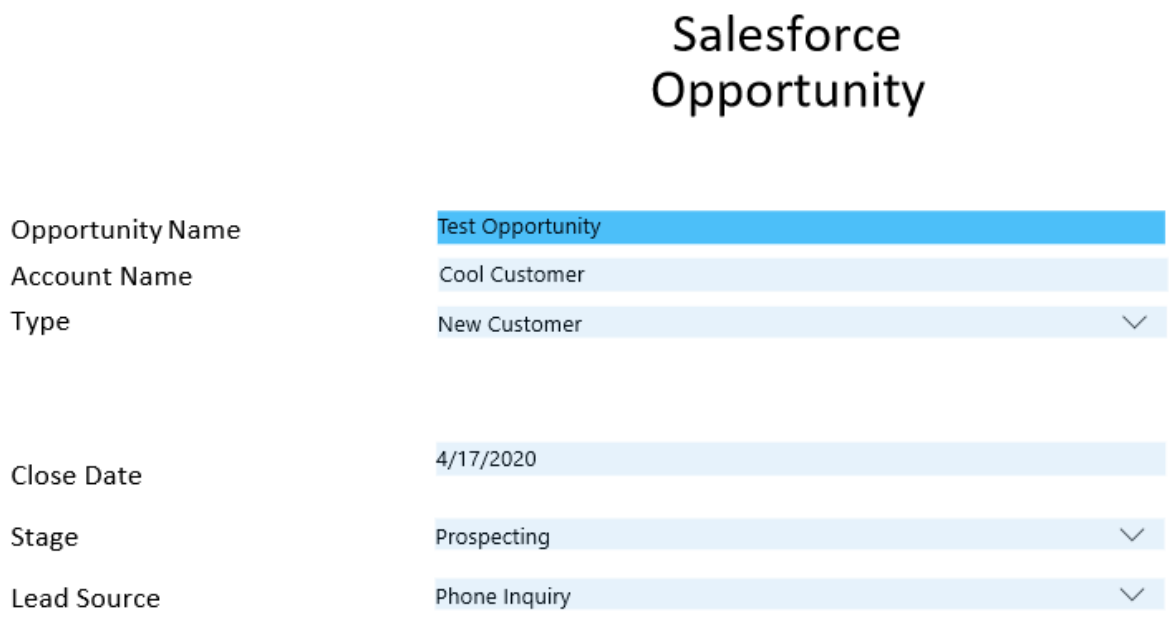


Figure 5-e: Sample PDF Form Document

The Python script can be executed with the following command from the command prompt or within the built-in terminal in VS Code.

Code Listing 5-i: Executing the Script for One PDF File

```
python autoform.py opportunity.pdf
```

Alternatively, you can execute the script as follows. This is how it should be executed when there is more than one PDF form document to process.

Code Listing 5-j: Executing the Script for Multiple PDF Files

```
python autoform.py
```

The execution of the Python script should result in the creation of a .txt file (with the same name as the PDF file) for each PDF file found within the script's folder.

In my case, the execution of the Python script has resulted in the creation of an **opportunity.txt** file in the same folder as the Python script. The console output in VS Code was as follows.

```
PS C:\Projects\Projects\Syncfusion\RPA Succinctly\demo> python3 autoform.py
Selectable: opp5 -> New Customer
Normal: opp4 -> Cool Customer
Normal: opp9 -> 4/17/2020
Normal: opp3 -> Test Opportunity
Selectable: opp11 -> Prospecting
Selectable: opp6 -> Phone Inquiry
```

Figure 5-f: VS Code Script Execution Console Output

Let's now open the **opportunity.txt** file, copy its content, and go to the Salesforce **New Opportunity** webpage. Within Chrome Developer Tools, go to the **Console** section, paste the content copied, and select the **Enter** key.

```
Elements  Console  Sources  Network  Performance  Memory  »
top ▼  Filter  Default levels ▼
> function setItem(s, v) {for (var i = 0; i < s.options.length; i++) {if
  (s.options[i].text == v) {s.options[i].selected =
  true;return;}}}setItem(document.getElementById("opp5"), "New
  Customer");document.getElementById('opp4').value = 'Cool Customer ';
  document.getElementById('opp9').value = '4/17/2020';
  document.getElementById('opp3').value = 'Test Opportunity';
  function setItem(s, v) {for (var i = 0; i < s.options.length; i++) {if
  (s.options[i].text == v) {s.options[i].selected =
  true;return;}}}setItem(document.getElementById("opp11"), "Prospecting");function
  setItem(s, v) {for (var i = 0; i < s.options.length; i++) {if (s.options[i].text ==
   v) {s.options[i].selected =
  true;return;}}}setItem(document.getElementById("opp6"), "Phone Inquiry");
<· undefined
```

Figure 5-g: The Generated Browser (JavaScript) Code

Once the browser has executed the code (which happens immediately after you select Enter), you should see an **undefined** output on the console. This indicates that the script has executed.

The execution of the script by the browser immediately results in the fields on the webpage being automatically filled in, as you can see in the following figure.

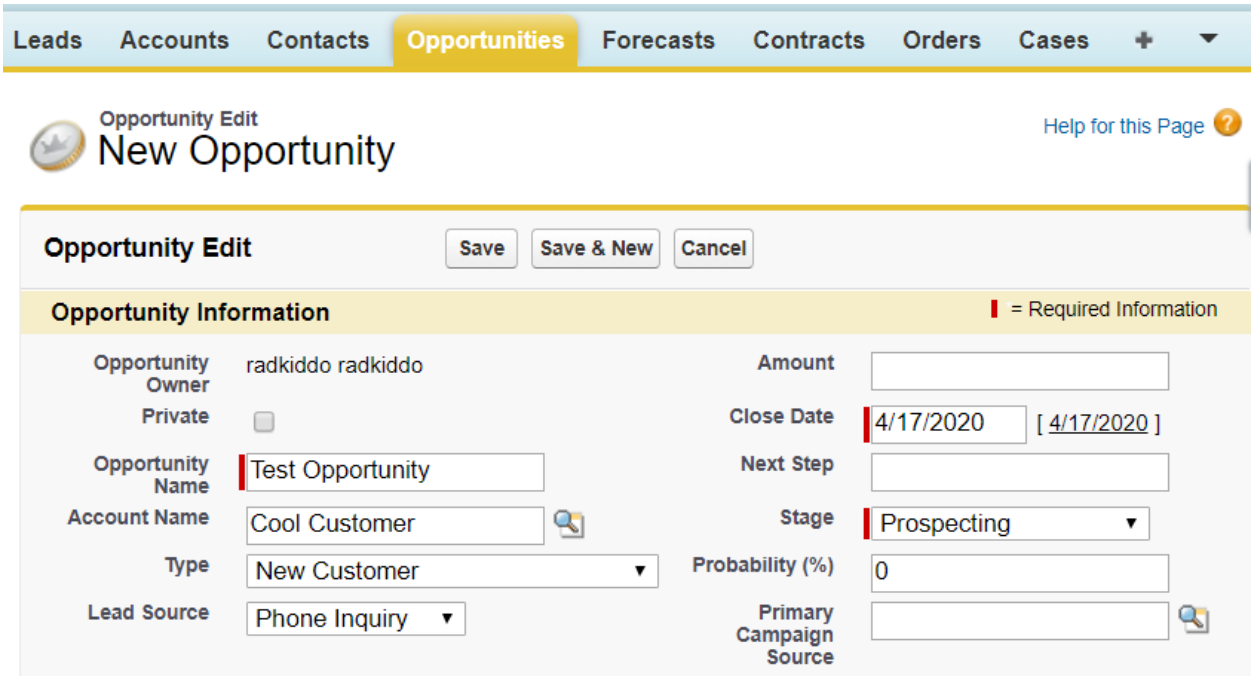


Figure 5-h: Webpage Fields Filled In

There you go—we now have a way to extract data from PDF form documents and generate a script for each PDF file that can semi-automatically fill in elements on a webpage corresponding to fields on PDF forms.

With just a little extra effort, such as programmatically opening the browser console, pasting the generated script, and repeating the process multiple times, we can achieve a complete data entry automation scenario for submitting new Salesforce opportunities.

This was just a demonstration of concept, using one specific web application and one specific PDF form document. Now that you understand the idea behind it, I'm sure you can appreciate it's potential and how it can be applied elsewhere—to other web apps, and using other PDF documents.

Finished code

You can download all the code written throughout this book, as well as supporting material, such as configuration or test files, from the following URL.

Scanned or text PDFs

Throughout this chapter, we explored how to extract data from PDF form documents, but what about extracting data from scanned PDFs, text-based PDFs, or even images?

Extracting data from scanned-based PDFs and images requires optical character recognition (OCR) technology. There are open-source libraries that can be used with Python to assist with the job, such as OpenCV.

However, the challenge of extracting data from sources that are unstructured, such as scanned PDFs, images, and text-based PDFs, is that even after OCR has been performed, what you have as a result are simply lines of text, words, and characters.

In other words, there are no relationships between those lines of text, and no definition of fields and values. This means you still have to write a layer of logic, to interpret the text extracted by using business rules or even multiple templates.

Say you would be processing invoices to automate an accounts payable system. It wouldn't be enough to perform OCR and extract the text from the scanned or text-based PDF invoices. You would need to have some sort of logic capable of finding supplier information, invoice line items, purchase order numbers, bank account details, and other values to be able to do meaningful invoice processing.

That is a tough task. There are many companies with proprietary solutions that solely focus on automating invoice processing and accounts payable systems.

So, if you want to extract meaningful data from scanned PDFs, images, or text-based PDFs using Python, without having to write your layer of logic, there are two excellent options, both cloud services that can return key-value pairs (field information) and table details from unstructured documents using a machine-learning approach.

These services are Amazon Textract and Azure Forms Recognizer—both support Python and provide amazing results. In my opinion, they are two of the best and most cost-effective methods to extract meaningful data from unstructured documents.

Delving into either one would require a book of its own, and this is well beyond the scope of this particular book, but both offer great possibilities for working with unstructured documents with Python.

Imagine having to extract correctly all the values for the following receipt (which is available on the Forms Recognizer website as an example).

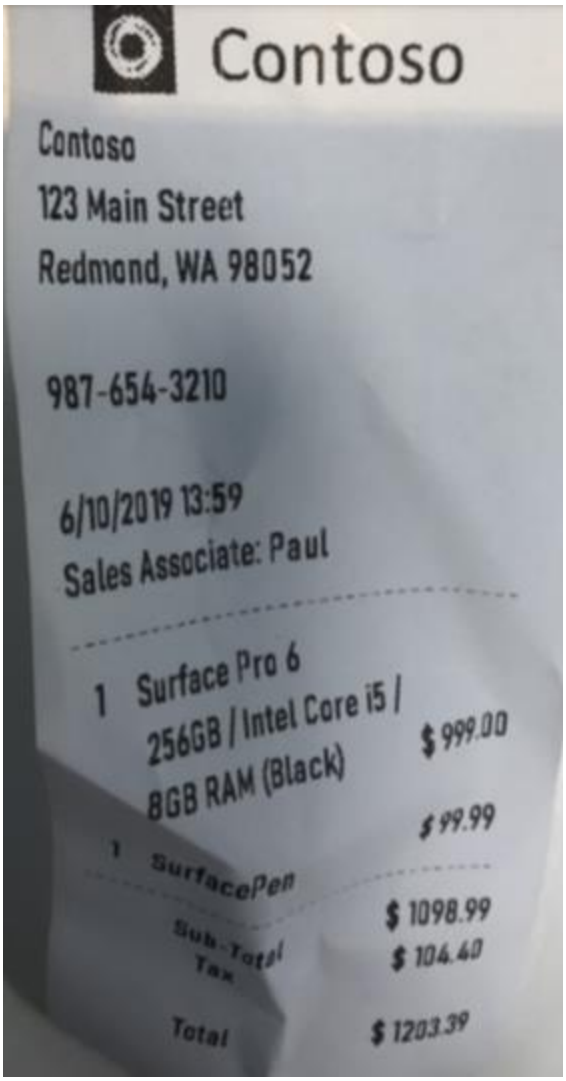


Figure 5-i: Forms Recognizer Sample Image (Courtesy of Microsoft)

I recently (briefly) experimented with Forms Recognizer, and to give you a sense of how a Python script that can extract key-value pairs from a reciept like this would look, I've placed the complete script in the following code listing.

Code Listing 5-k: Forms Recognizer Demo Script

```
########### Python Form Recognizer Async Layout #############

import json
import time
from requests import get, post

# Endpoint URL
endpoint = r"https://formsrecogtest.cognitiveservices.azure.com/"
```

```
apim_key = "<< Your Forms Recognizer key goes here >>"
post_url = endpoint + "/formrecognizer/v2.0-
preview/prebuilt/receipt/analyze"
source = r"contoso.jpg"

headers = {
    # Request headers
    'Content-Type': 'image/jpeg',
    'Ocp-Apim-Subscription-Key': apim_key,
}

params = {
    "includeTextDetails": True
}

with open(source, "rb") as f:
    data_bytes = f.read()

try:
    resp = post(url = post_url,
    data = data_bytes,
    headers = headers,
    params = params)

    if resp.status_code != 202:
        print("POST analyze failed:\n%s" % resp.text)
        quit()
    print("POST analyze succeeded:\n%s" % resp.headers)
    get_url = resp.headers["operation-location"]
except Exception as e:
    print("POST analyze failed:\n%s" % str(e))
    quit()

n_tries = 10
n_try = 0
wait_sec = 6
while n_try < n_tries:
    try:
        resp = get(url = get_url,
        headers = {"Ocp-Apim-Subscription-Key": apim_key})
        resp_json = json.loads(resp.text)
        if resp.status_code != 200:
            print("GET Layout results failed:\n%s" % resp_json)
            quit()
```

```
        status = resp_json["status"]
        if status == "succeeded":
            print("Layout Analysis succeeded:\n%s" % resp_json)
            quit()
        if status == "failed":
            print("Analysis failed:\n%s" % resp_json)
            quit()
        # Analysis still running. Wait and retry.
        time.sleep(wait_sec)
        n_try += 1
    except Exception as e:
        msg = "GET analyze results failed:\n%s" % str(e)
        print(msg)
        quit()
```

Executing this Python script produces the results shown in Code Listing 5-l. Please keep in mind that to achieve this, you will need to sign up for Forms Recognizer and set up an instance of the service within the Azure Portal.

The results are way beyond what I expected. Not only was Forms Recognizer able to detect fields (key-value pairs) from unstructured text, but it was able to do so by analyzing an image that is not even straightened—you can see from the sample image that the original paper receipt that was scanned is skewed. Now to me, that is impressive.

Code Listing 5-l: Forms Recognizer Script Results

```
Merchant: Contoso
Address: 123 Main Street Redmond, WA 98052
Phone number: +9876543210
Date: 2019-06-10
Time: 13:59:00
Subtotal: 1098.99
Tax: 104.4
Total: 1203.39
```

Final thoughts

We’ve reached the end of this chapter and book. It’s given you a good taste of what robotics process automation (RPA) can achieve by mostly using out-of-the-box Python modules and libraries. However, we barely scratched the surface of what can be done with RPA using Python.

RPA is not only about automation, but also about imagination. It's about how far you can let your imagination run wild and see which day-to-day tasks you can automate. It's about testing, experimenting, iterating, seeing what works and what doesn't, and ending up with code that can solve a particular problem—while saving time.

I will leave you with a teaser and a challenge. Go to GitHub and search for **RPA**, and you'll be surprised how many projects exist that are dedicated to this fascinating subject.

One of my favorites is Automagica, which is a Python library completely dedicated to RPA, leveraging multiple applications, such as Excel, Word, browsers, and many others.

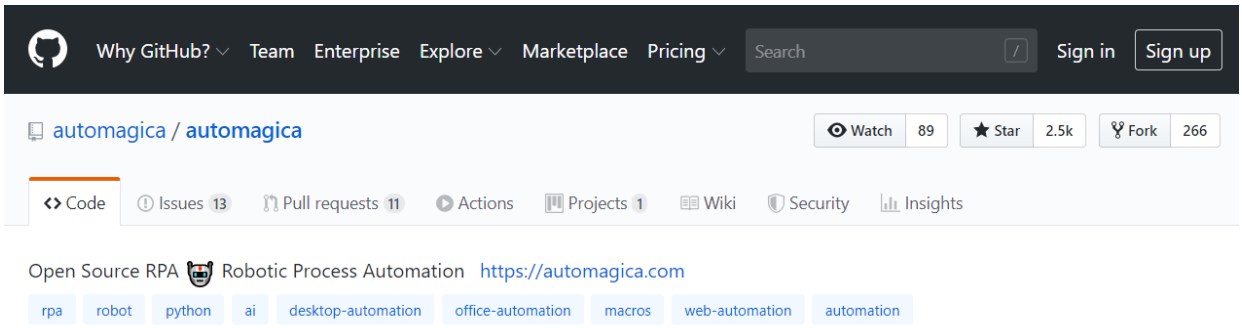


Figure 5-i: GitHub Automagica Repo

Exploring Automagica, or any other specialized RPA framework, would require a book all on its own. The beauty of RPA is not only the prospects it gives us by saving valuable time, but also that, with some out-of-the-box thinking; an easy, yet powerful programming language; and some imagination, time-saving and interesting utilities can be developed, making us more productive and better at what we do.

I sincerely hope this book has inspired you and encouraged you to dig deeper into this promising technology and shown you that robots are not as scary as Hollywood movies portray them, after all.

There's a final teaser before wrapping up the book, which is included as an appendix. It's part of a multi-file Python 2.7 backup script I wrote that uses classes. It's my backup script (it includes FTP functionality), which keeps my data safe. I use it on all my computers. With one click, it backs up all my data to different locations.

Until next time, thank you for reading and following along. Stay well, and be awesome!

Appendix – My Python 2.7 Backup Script

Background

Some time ago, I wanted to automate my backups. I looked around on the internet for a solution that would fit my needs, and I really couldn't find any that would suit me 100 percent. Don't get me wrong—there are great tools out there, but I needed something that I could adapt and customize whenever I needed.

So, one day I told my wife that I needed to spend the following two or three weekends solving a major problem. She seemed worried at the beginning, but once I explained that I had to write a backup script from scratch, her face changed—as you might have guessed, that wasn't particularly exciting to her.

Anyway, I explained the "urgency" of the situation and that years of work could be at risk if I didn't have a way to properly back up and manage my data. She eventually got it, and told me that once I had it ready, she wanted a copy running on her machine, too! I spent roughly two full weekends writing, testing, and optimizing my backup script (called **Copyzip**), which to this day keeps my data safe.

At the time, I was still working with Python 2.7. It works so well that I haven't even bothered upgrading it to Python 3.6 or later. It just works. Besides, I've set it up to have a pinned taskbar icon for it on all my Windows 10 machines, so I simply click the icon and it runs my whole backup operation (for all my work folders) without requiring me to do anything else.

Figure Appendix-a: My Backup Pinned Taskbar Icon – Copyzip

Here is the command that executes the script.

Code Listing Appendix-a: Copyzip Execute Command

```
C:\Python27\python.exe C:\Python27\copyzip\copyzip.py
```

The following figure shows the properties of the pinned taskbar icon shortcut.

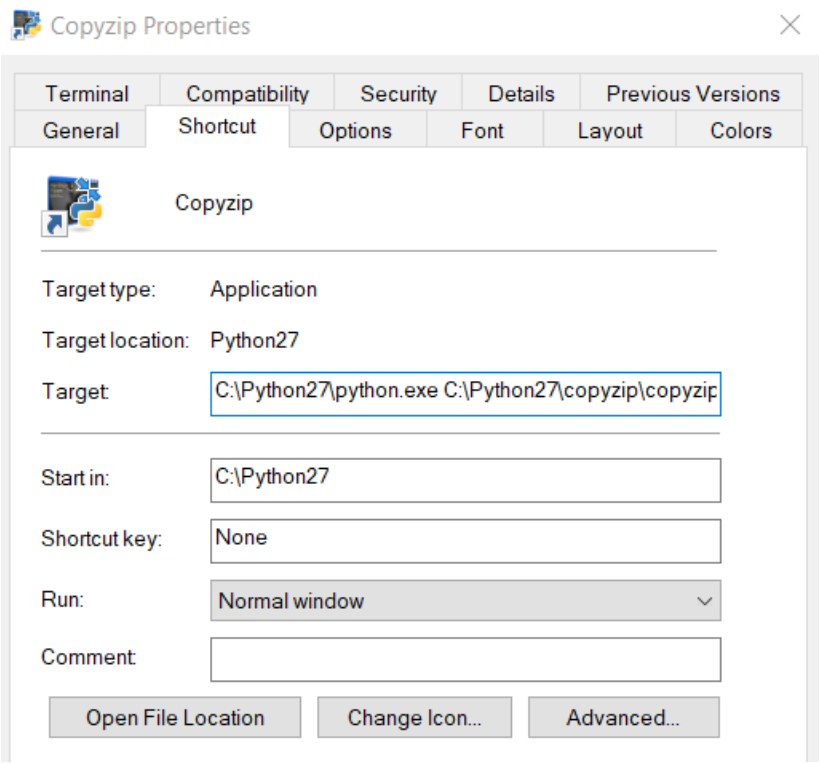


Figure Appendix-b: My Backup Pinned Taskbar Icon Properties – Copyzip

I have put all the backup script files directly into the **Python27** folder on all my machines, under the **copyzip** folder.

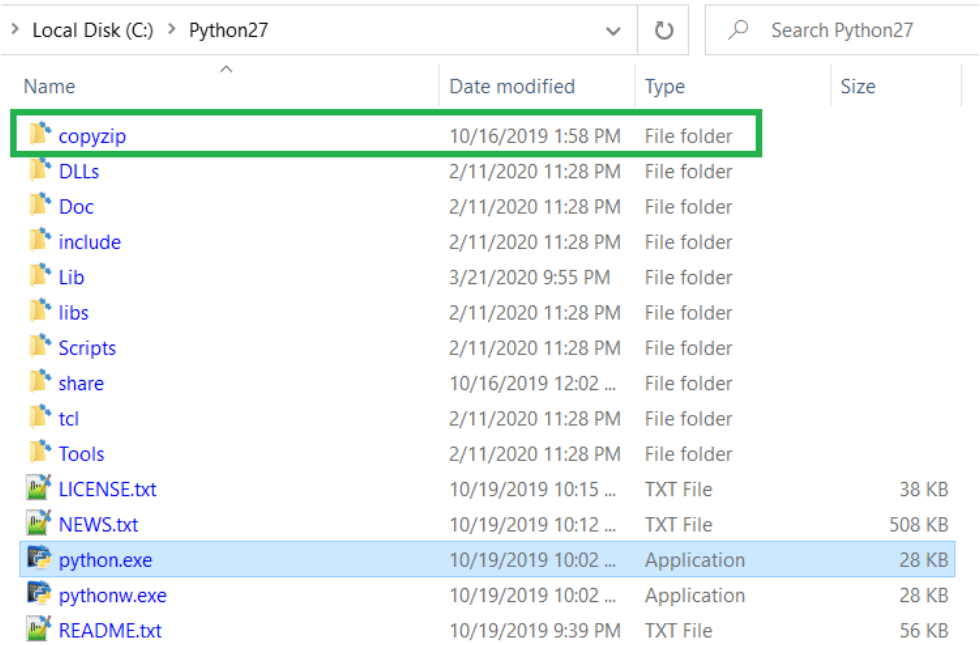
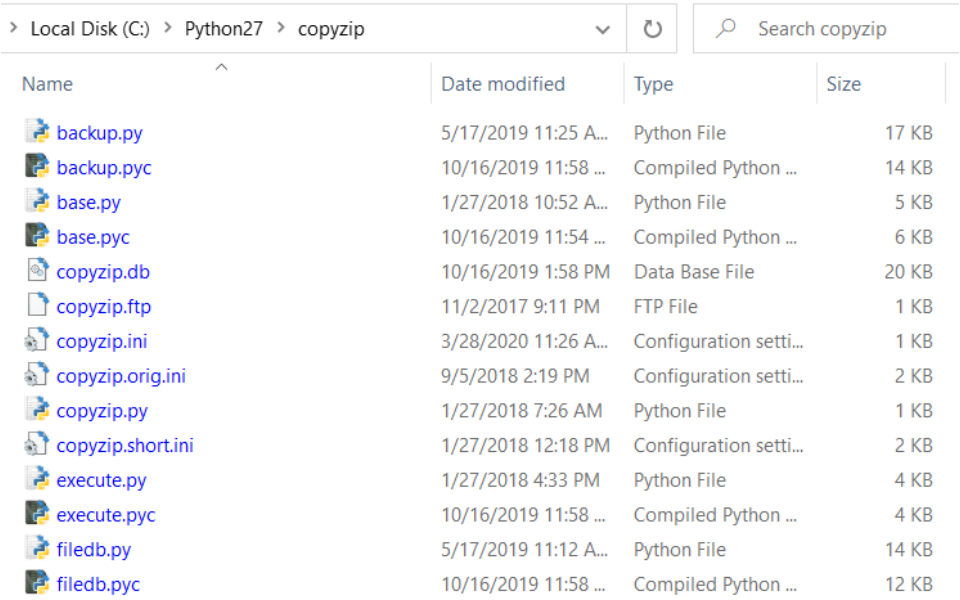


Figure Appendix-c: The Backup Script within the Python27 Folder

Let's now explore the contents of the **copyzip** folder. You can see that the script is made up of several files.



Local Disk (C:) > Python27 > copyzip

Search copyzip

Name	Date modified	Type	Size
backup.py	5/17/2019 11:25 A...	Python File	17 KB
backup.pyc	10/16/2019 11:58 ...	Compiled Python ...	14 KB
base.py	1/27/2018 10:52 A...	Python File	5 KB
base.pyc	10/16/2019 11:54 ...	Compiled Python ...	6 KB
copyzip.db	10/16/2019 1:58 PM	Data Base File	20 KB
copyzip.ftp	11/2/2017 9:11 PM	FTP File	1 KB
copyzip.ini	3/28/2020 11:26 A...	Configuration setti...	1 KB
copyzip.orig.ini	9/5/2018 2:19 PM	Configuration setti...	2 KB
copyzip.py	1/27/2018 7:26 AM	Python File	1 KB
copyzip.short.ini	1/27/2018 12:18 PM	Configuration setti...	2 KB
execute.py	1/27/2018 4:33 PM	Python File	4 KB
execute.pyc	10/16/2019 11:58 ...	Compiled Python ...	4 KB
filedb.py	5/17/2019 11:12 A...	Python File	14 KB
filedb.pyc	10/16/2019 11:58 ...	Compiled Python ...	12 KB

Figure Appendix-d: The Backup Script File Contents

The script works almost in the same way as the one we developed previously, though it's a bit fancier. It contains progress bars and colors that can be configured independently for each backup operation.

The script can also track file changes on specific folders (by appending **db@** before an operation), has FTP capabilities, and has the ability to loop continuously and silently back up changes as they take place.

The following listing shows the actual backup configuration file from my main machine.

Code Listing Appendix-b: Copyzip Execute Command

```
C:\Python27\copyzip,.pyc||E:\lenovo\zipped\cdrive_python27_copyzip.zip||yellow|z
C:\Personal||E:\lenovo\zipped\cdrive_Personal.zip||cyan|z
C:\Projects||E:\lenovo\zipped\cdrive_Projects.zip||purple|z
D:\Projects||E:\lenovo\zipped\ddrive_Projects.zip||cyan|z
C:\Users\fasta\Downloads||E:\lenovo\zipped\cdrive_Downloads.zip||yellow|z
db@C:\Pictures\Saved||E:\lenovo\zipped\cdrive_Saved_Pictures.zip||cyan|z
C:\Temp||E:\lenovo\zipped\cdrive_Temp.zip||yellow|z
```

This is how the execution of the script looks on one of my machines.

```
Copyzip
INFO:keyring.backend:Loading KWallet
INFO:keyring.backend:Loading SecretService
INFO:keyring.backend:Loading Windows
INFO:keyring.backend:Loading chainer
INFO:keyring.backend:Loading macOS
Starting...
INFO:root:2020-04-18 13:16:10: Starting...
>> Zipping files from: C:\Python27\copyzip
INFO:root:2020-04-18 13:16:10: >> Zipping files from: C:\Python27\copyzip
INFO:root:2020-04-18 13:16:10: backup.pyc -> skipped (omitted)
INFO:root:2020-04-18 13:16:10: base.pyc -> skipped (omitted)
INFO:root:2020-04-18 13:16:10: execute.pyc -> skipped (omitted)
>> [===============] 100.0% -> filedb.pyc -> skipped (omitted)
INFO:root:2020-04-18 13:16:10: filedb.pyc -> skipped (omitted)
Writing... E:\lenovo\zipped\cdrive_python27_copyzip.zip
>> 10 files zipped successfully to: E:\lenovo\zipped\cdrive_python27_copyzip.zip
INFO:root:2020-04-18 13:16:10: >> 10 files zipped successfully to: E:\lenovo\zipped\cdrive_python27_copyzip.zip
>> Zipping files from: C:\Personal
INFO:root:2020-04-18 13:16:10: >> Zipping files from: C:\Personal
>> [===============] 100.0% -> cdrive_Personal.zip
Writing... E:\lenovo\zipped\cdrive_Personal.zip
>> 1 files zipped successfully to: E:\lenovo\zipped\cdrive_Personal.zip
INFO:root:2020-04-18 13:16:12: >> 1 files zipped successfully to: E:\lenovo\zipped\cdrive_Personal.zip
>> Zipping files from: C:\Projects
INFO:root:2020-04-18 13:16:20: >> Zipping files from: C:\Projects
>> [---------------] 0.1% -> Bing Image Search API.rar
```

Figure Appendix-e: The Backup Script Running

I could have packaged it into an executable, installer, or a single Python package, so that all its external Python package dependencies would be automatically installed, but to be honest, I haven't had the time to do that yet. So if you want to use it or test it out, you will have to manually install some Python 2.7 packages that the script depends on.

There are two reasons why I wanted to share this with you. The first is that I wanted to show you how you can add FTP file synchronization capabilities to your script. The second is that I wanted to show you how you can expand and structure your script to use Python classes, split it into multiple files, and have more capabilities overall.

I'll leave you with the script's main FTP method—it's a method (and not a function) because the code is structured into Python classes.

Main FTP method

Here is code for the script's main FTP functionality.

Code Listing Appendix-c: Main FTP Functionality

```
@classmethod
def __doftp(cls, localfolder, remotefolder, server, color):
    stats = {}
    count = 0
    if not Base.hasended() and localfolder != '' and remotefolder != '':
        if cls.__numfiles(localfolder) > 0:
            msg = '>> FTPing files from: ' + localfolder
```

```
            print colored(msg, color)
            Base.log(msg, False)
            try:
                svr = server['ftp']
                usr = server['user']
                pwd = server['pwd']
                local = FsTarget(localfolder)
                cls.__chkftpfld(remotefolder, svr, usr, pwd)
                remote = FtpTarget(remotefolder, svr,
                username=usr, password=pwd, tls=False)
                opts = {'force': True, 'delete_unmatched': True, 'verbose': 3}
                sync = UploadSynchronizer(local, remote, opts)
                sync.run()
                stats = sync.get_stats()
                remote.close()
            except BaseException, err:
                Base.log(str(err), True)
                msg = 'Failed FTPing to: ' + remotefolder
                print colored(msg, 'red')
                Base.log(msg + ' -> ' + str(err), True)
            finally:
                if bool(stats):
                    count = stats['upload_files_written']
                output = '>> ' + str(count) + ' files FTPed to: ' + remotefolder
                print colored(output, 'green')
                Base.log(output, False)
    return count
```

Full source code

Here you can find the full source code of my personal backup script. If you can upgrade it to Python 3.6.X or later, please let me know and send me a copy. Please note that I won't provide any email support for this script.

- backup.py
- base.py
- copyzip.py
- execute.py
- filedb.py