Enterprise Automation with Python

Automate Excel, Web, Documents, Emails, and Various Workloads with Easy-to-code Python Scripts

AMBUJ AGRAWAL

bpb

Enterprise Automation with Python

Automate Excel, Web, Documents, Emails, and Various Workloads with Easy-to-code Python Scripts

AMBUJ AGRAWAL

bpb

Enterprise Automation with Python

Automate Excel, Web, Documents, Emails, and Various Workloads with Easy-to-code Python Scripts

Ambuj Agrawal

www.bpbonline.com

Group Product Manager: Marianne Conor

Publishing Product Manager: Eva Brawn

Senior Editor: Connell

Content Development Editor: Melissa Monroe

Technical Editor: Anne Stokes

Copy Editor: Joe Austin

Language Support Editor: Justin Baldwin

Project Coordinator: Tyler Horan

Proofreader: Khloe Styles

Indexer: V. Krishnamurthy

Production Designer: Malcolm D'Souza

Marketing Coordinator: Kristen Kramer

First published: August 2022

Published by BPB Online

WeWork, 119 Marylebone Road

London NW1 5PU

UK | UAE | INDIA | SINGAPORE

ISBN 978-93-55511-379

www.bpbonline.com

Dedicated to

My beloved Parents:

Anil Agrawal & Saroj Agrawal

About the Author

Ambuj Agrawal is an industry expert in Artificial Intelligence and Enterprise Automation. He has received numerous innovation awards from Citibank, Imperial College London, Ministry of Justice UK, Bristol University, among others. He is also one of the youngest recipients of the "Exceptional Talent Visa in Digital Technology" by the UK Government for expertise in Compiler Design and Machine Learning.

He has been one of the youngest Speakers at the Money2020 Europe, Fin.Techsummit Europe, Future of Work Summit London and Automation Summit Paris on the topic “Automation and Future of Work”.

About the Reviewer

Akanksha Sinha is an Architect in Intelligent Process Automation team with 8 years of Automation experience out of her 15 years of journey in various progressive roles of Software Development in Cognizant. She is extensively involved in digital transformation of client's landscapes through automations in domains of Digital Marketing and Technology. Her area of work revolves in exploring automation potential, identifying automation use cases, solutioning and developing a suite of bots along with her team in close collaboration of multiple client stakeholders and eventually providing digital experience to her clients through a suite of automation solutions.

She has worked with clients like Google, Twitter and Hartford Life. Her skills are in Javascript, Google Apps Script, Google Cloud, web development, Unix, NLP, Python, Chatbots, Open Source RPA. Open Source RPAs and NLP are her areas of interest where she has worked on several prototypes.

She was one of the founding members of the Tools & Automation team in Cognizant for Operation Teams working for Tech giants.

She is Google Cloud certified associate and Kore.AI certified Virtual Assistant developer.

She has a B.E in E&E from RTM Nagpur University, and is working with Cognizant.

Acknowledgement

First and foremost, I would like to thank my parents who continuously encouraged me to write this book — I could have never completed this book without their support.

I would also like to thank my family and friends who provided me with the continued support during my writing of this book.

I am also grateful to the team at BPB Publications, who provided me the opportunity to publish this book and for providing valuable feedback throughout the process of writing this book.

Preface

This book takes the reader through different examples and code samples to automate repetitive work tasks. This book also gives solutions to common automation requirements and repetitive tasks faced during the day to day work environment. After reading this book you will be able to create automations for business processes using Python. You will also be able to identify the most common business process for automation.

This book will equip you with the knowledge of creating, reading, modifying and extract data from Excel documents using Python programs. You will also be able to extract data from websites, PDF documents and send and read messages using Gmail, Outlook and WhatsApp. This book will help readers to create automations to automate their boring work and increase the efficiency of their organizations by 500%.

This book is divided into 11 chapters. The details are listed below.

In **Chapter 1**, you will be introduced to the installation steps and setting up the development environment for Python. We will also cover the installation of Python packages and libraries required for building automations.

In **Chapter 2**, you will be introduced to the installation steps and setting up the development environment for Python. We will also cover the installation of Python packages and libraries required for building automations.

In **Chapter 3**, we will discuss the mindset needed to be successful in implementation of automations within your organizations. We will go through the process of identifying and prioritizing automation opportunities. We will also discuss the ways to share the developed automations with the wider organization once they are created.

In **Chapter 4**, we will discuss ways to automate Excel workflows including creating, writing, and updating the Excel documents. We will also discuss the data manipulation techniques with Excel and CSV documents.

In **Chapter 5**, we will go through automation for websites and web-based tasks. We will look at how to download data from websites and automate

data extraction from websites by parsing HTML documents. We will also look at the Selenium framework to automate web actions such as mouse click and keyboard actions on different websites.

In **Chapter 6**, we will look at various file-based automations for different file types in Python. We will discuss some of the Python libraries that are used to automate different file types. We will also look at ways to extract data from PDF documents and Word documents type file structure.

In **Chapter 7**, we would learn to automate email-based tasks using Gmail, Outlook and other SMTP clients. We will also look at Text message and WhatsApp automation using the Twilio API.

In **Chapter 8**, we would learn to automate Graphical User Interface (GUI) by controlling the Keyboard and Mouse Actions. We will be using the Python library PyAutoGUI which works with Windows, Mac and Linux and provides automations for GUI elements within the application.

In **Chapter 9**, we will look at computer Image fundamentals and the Pillow Python library for manipulating images. We would also look at the Tesseract library which can be used to extract text within images and scanned documents.

In **Chapter 10**, we will look at scheduling automations using dates and timers. We would also look at external applications that can allow us to run automations based on certain events such as receiving a new email or during the start of an application.

In **Chapter 11**, we will look at methods to extend your Python scripting knowledge and develop complex end to end process automations based on your requirements. We will learn how to work with external libraries and use external code to build these automations. We would also look at creating Python web services and using Machine Learning for automation.

Code Bundle and Coloured Images

Please follow the link to download the
Code Bundle and the ***Coloured Images*** of the book:

https://rebrand.ly/de9f96

The code bundle for the book is also hosted on GitHub at **https://github.com/bpbpublications/Enterprise-Automation-with-Python**. In case there's an update to the code, it will be updated on the existing GitHub repository.

We have code bundles from our rich catalogue of books and videos available at **https://github.com/bpbpublications**. Check them out!

Errata

We take immense pride in our work at BPB Publications and follow best practices to ensure the accuracy of our content to provide with an indulging reading experience to our subscribers. Our readers are our mirrors, and we use their inputs to reflect and improve upon human errors, if any, that may have occurred during the publishing processes involved. To let us maintain the quality and help us reach out to any readers who might be having difficulties due to any unforeseen errors, please write to us at :

errata@bpbonline.com

Your support, suggestions and feedbacks are highly appreciated by the BPB Publications' Family.

customer, you are entitled to a discount on the eBook copy. Get in touch with us at: **business@bpbonline.com** for more details.

At **www.bpbonline.com**, you can also read a collection of free technical articles, sign up for a range of free newsletters, and receive exclusive discounts and offers on BPB books and eBooks.

Piracy

If you come across any illegal copies of our works in any form on the internet, we would be grateful if you would provide us with the location address or website name. Please contact us at **business@bpbonline.com** with a link to the material.

If you are interested in becoming an author

If there is a topic that you have expertise in, and you are interested in either writing or contributing to a book, please visit **www.bpbonline.com.** We have worked with thousands of developers and tech professionals, just like you, to help them share their insights with the global tech community. You can make a general application, apply for a specific hot topic that we are recruiting an author for, or submit your own idea.

Reviews

Please leave a review. Once you have read and used this book, why not leave a review on the site that you purchased it from? Potential readers can then see and use your unbiased opinion to make purchase decisions. We at BPB can understand what you think about our products, and our authors can see your feedback on their book. Thank you!

For more information about BPB, please visit **www.bpbonline.com.**

Table of Contents

CHAPTER 1

Setting Up the Automation Environment

Introduction

In this chapter, you will be introduced to the installation steps and setting up the development environment for Python. We will also cover the installation of Python packages and libraries required for building automations.

Structure

In this chapter, we will cover the following topics:

- Installing and getting started with Mu for Python 3
- Installing third party packages with Mu

Objectives

After studying this chapter, you will be able to set the automation environment on your machine. You will also get an understanding of Python development environments and be able to run Python on your machine.

Installing and getting started with Mu for Python 3

The code with **Mu** is a simple Python editor for beginner programmers. Download **Mu installer** from **https://codewith.mu/en/download**. Find the installer you just downloaded (it's probably in your `Downloads` folder). Double click on the installer to run it. If you get any warning while installing, accept those warnings and run the installer. Once the installation has completed successfully, click on `Finish` to close the installer.

Start Mu

You can start Mu by clicking on the icon in the **`Start`** menu or by typing **`Mu`** in the *Search* box. The first run will take a bit of time, and it will install and load all the required modules. Once you have started Mu, the code editor will look as shown in the following figure:

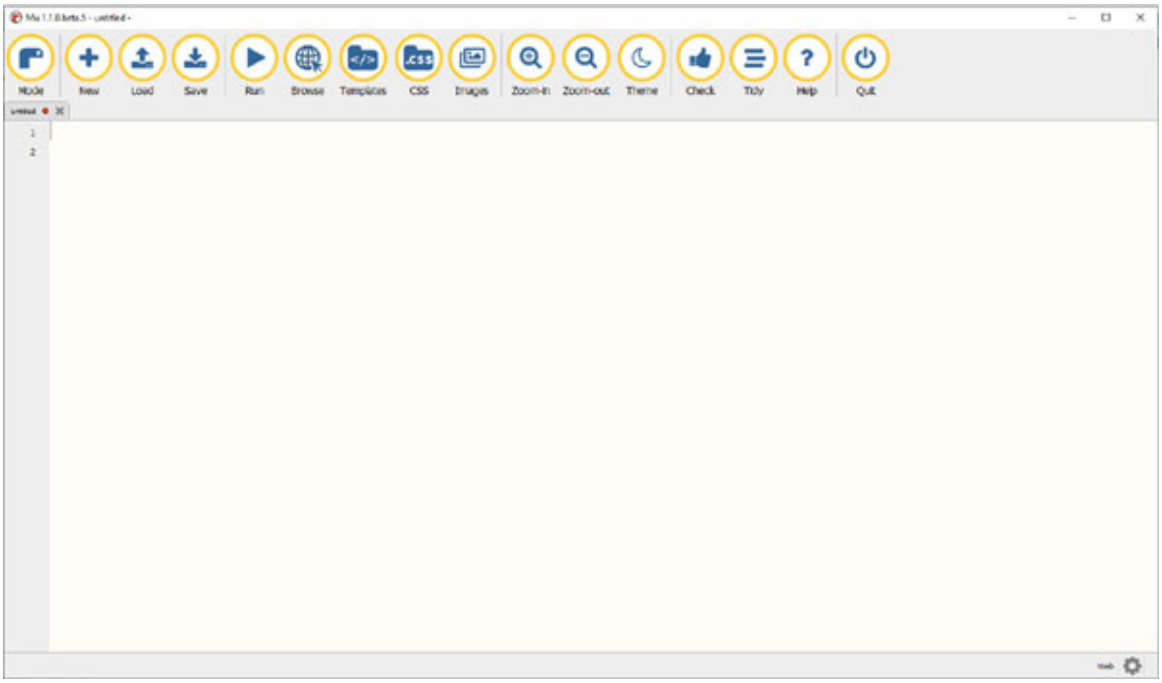


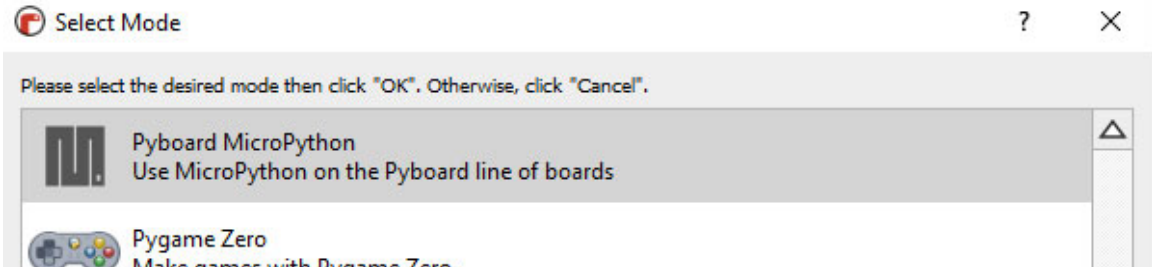
Figure 1.1: *Mu code editor*

The button bar in **Mu** contains buttons for creating and running the Python code along with the help instructions:

Figure 1.2: *Mu code editor toolbar*

The following are the button descriptions to help you get started with Mu:

- The **`Mode`** button is used for changing Mu modes. We will use the **`Python 3 Mode`** in this book:



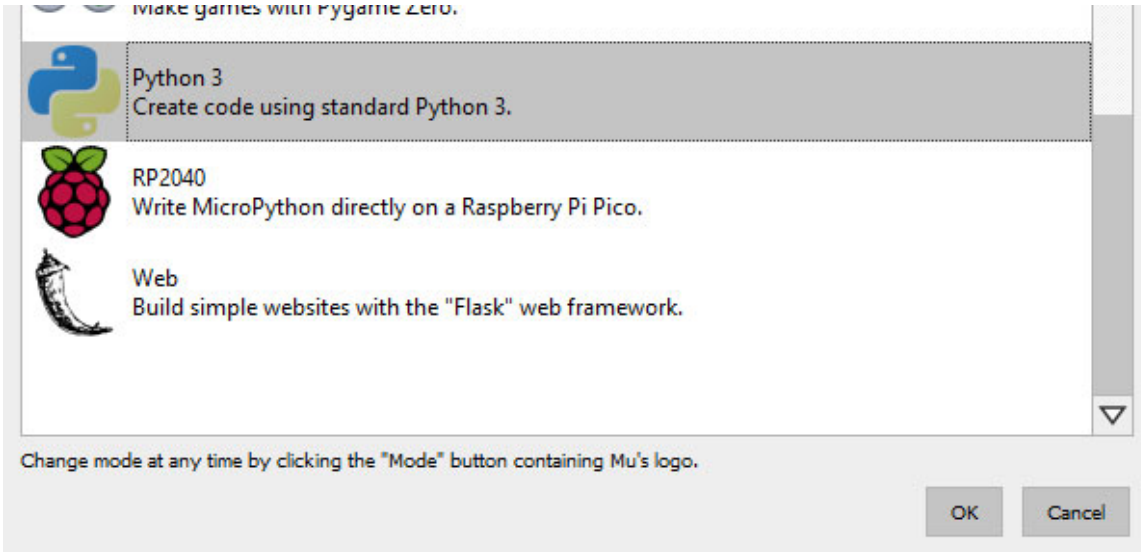


Figure 1.3: *Mu change mode view*

- The **`New`**, **`Load`**, and **`Save`** allow you to interact with files on your computer's hard drive:
  - **`New`**: This creates a new blank file.
  - **`Load`**: This opens a file selector to choose a file to load into Mu.
  - **`Save`**: This saves the file to your computer's hard drive. If the file has no name, you'll be asked to give one.
- The **`Run`** button runs the current script. When the code is running, the **`Run`** button turns into a **`Stop`** button. Click on **`Stop`** to force your code to exit in a clean way.
- The **`Debug`** button will start Mu's visual debugger allowing you to debug Python programs.
- The **`REPL`** button creates a new panel and the code you type here is evaluated line by line by Python.

You can learn more about the Mu editor from the Mu tutorial page - **https://codewith.mu/en/tutorials/1.1/**.

If you are an experienced programmer, then you can also use other Python code editing tools such as **PyCharm**, **VS Code**, **Jupyter** notebook, or any other code editor tool that works for you.

Installing third party packages with Mu

In this book, we will use a lot of third party packages to complete our automation scripts. Packages (sometimes called **libraries** or **modules**) are re-usable code that you can download, install, and use in your programs. They reduce the development time exponentially as you don't have to rewrite the code to achieve the same functionality in your project.

One of the main advantages of Python is that they have a huge collection of packages that allow you to achieve the desired functionality in your programs.

Mu comes with its own package installer which will download the code from the **Python Package Index pypi.org** and install it so that you can use it in your Mu projects.

To install a package Mu, click on the `Mu Administration` cog in the bottom right of the page. It is the *Setting* shaped button which is used to install Python packages and change settings for the code editor:

Figure 1.4: *Mu code editors settings button*

Select the `Third Party Packages` tab as shown in the following screenshot:

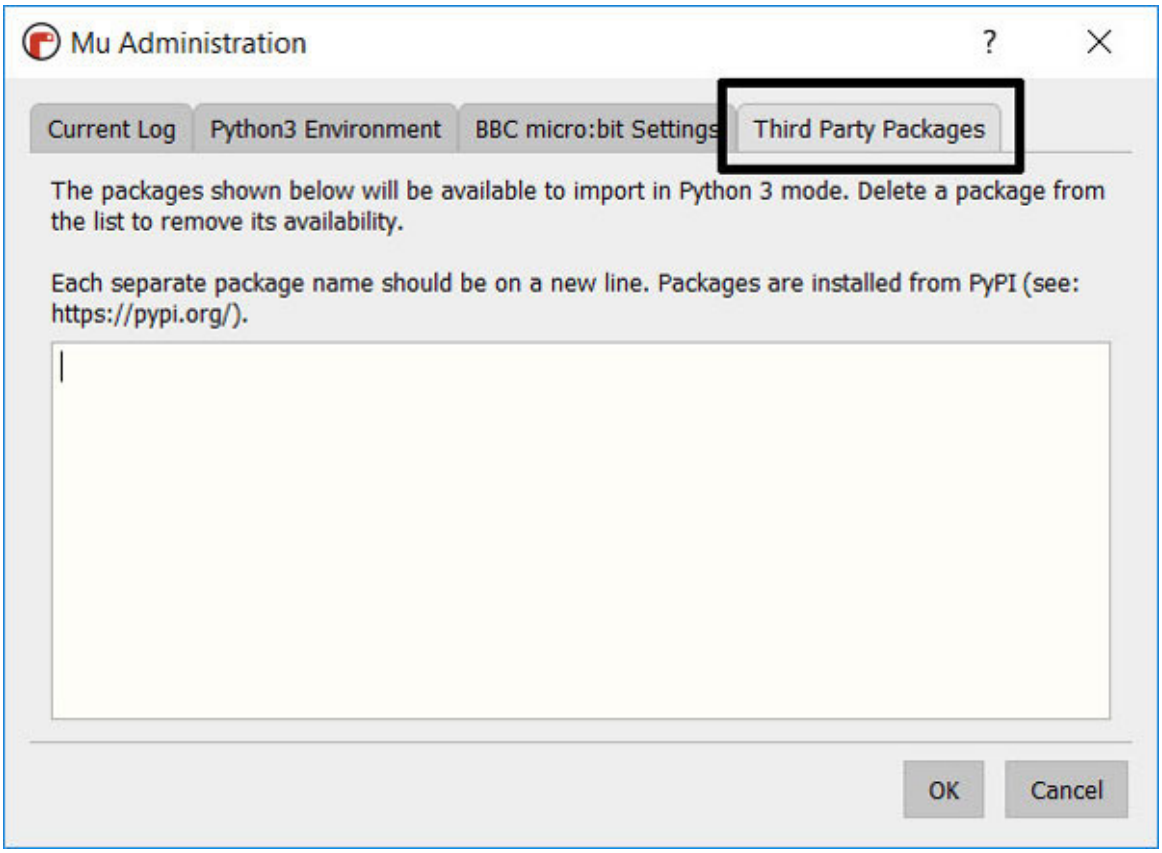


Figure 1.5: *Mu package installer page*

Enter the name of the package you wish to install and click on `OK`. The package will be downloaded and installed.

Advanced users can also install third party packages using `pip` - the package installer for Python.

Conclusion

In this chapter, we discussed about the steps to set up the Python development environment. In the next chapter, we will go through the fundamentals of Python to get you up and running with automating your day-to-day enterprise tasks.

Further reading

There are a lot of code editing tools and resources available on the Internet to get started with Python development. Some popular ones and their tutorials are given in the table as follows:

Resource name	Link
Code with Mu	https://codewith.mu/en/
Anaconda for Python	https://www.anaconda.com/products/individual
Jupyter notebooks for Python	https://jupyter.org/
VS Code for Python	https://code.visualstudio.com/docs/languages/python
PyCharm Python IDE	https://www.jetbrains.com/pycharm/
Code with Mu tutorials	https://codewith.mu/en/tutorials/
Python code editors guide	https://realpython.com/python-ides-code-editors-guide/
Top Python development editors	https://www.simplilearn.com/tutorials/python-tutorial/python-ide

Table 1.1: *Python code editing tools for developing with Python*

Questions

1. What are the different Python development editors available?
2. What are the advantages for using Mu for Python?
3. How can you install additional libraries using Mu?

CHAPTER 2

Fundamentals of Python

Introduction

In this chapter, we will introduce you to the Python programming language. We will cover fundamentals of Python, including decision statements, functions, and data structures. We will also look at how to import and use external libraries to achieve the desired goals.

Structure

In this chapter, we will cover the following topics:

- Introduction to Python
- Decision statements
- Data structures
- Loops/repetition
- Functions
- Libraries, modules, or packages

Objectives

After studying this chapter, you will be able to code basic programs in the Python programming language. You will gain knowledge of programming to get up and running with building Python programs. You will also have an understanding of the Python scripting language, syntax and data structures.

Introduction to Python

Python is a general-purpose programming language which has been built on top of C programming language. Python is also an interpreted language and can be used interactively (similar to using it as an *advanced calculator*

executing one command at a time). The **`Scripting`** mode in Python allows you to execute a series of commands in a saved text file, usually with a **`.py`** extension after the name of your file.

You can do just about anything with Python and it is one of the easiest languages to learn for beginners. Python is widely used all over the world to build automations, machine learning models, data analysis, and web development. It can help you build automations for day-to-day work tasks, create web applications, perform data analysis, and build machine learning models.

We are using *Python version 3.8.5* in this book and the code should work for minor Python versions updates in the future. To start with a simple program in Python, open the Mu editor, type **`print('Hello World')`**, save the file, and click on **`Run`**. You will see the **`Hello World`** printed in the console window as shown in the following figure:

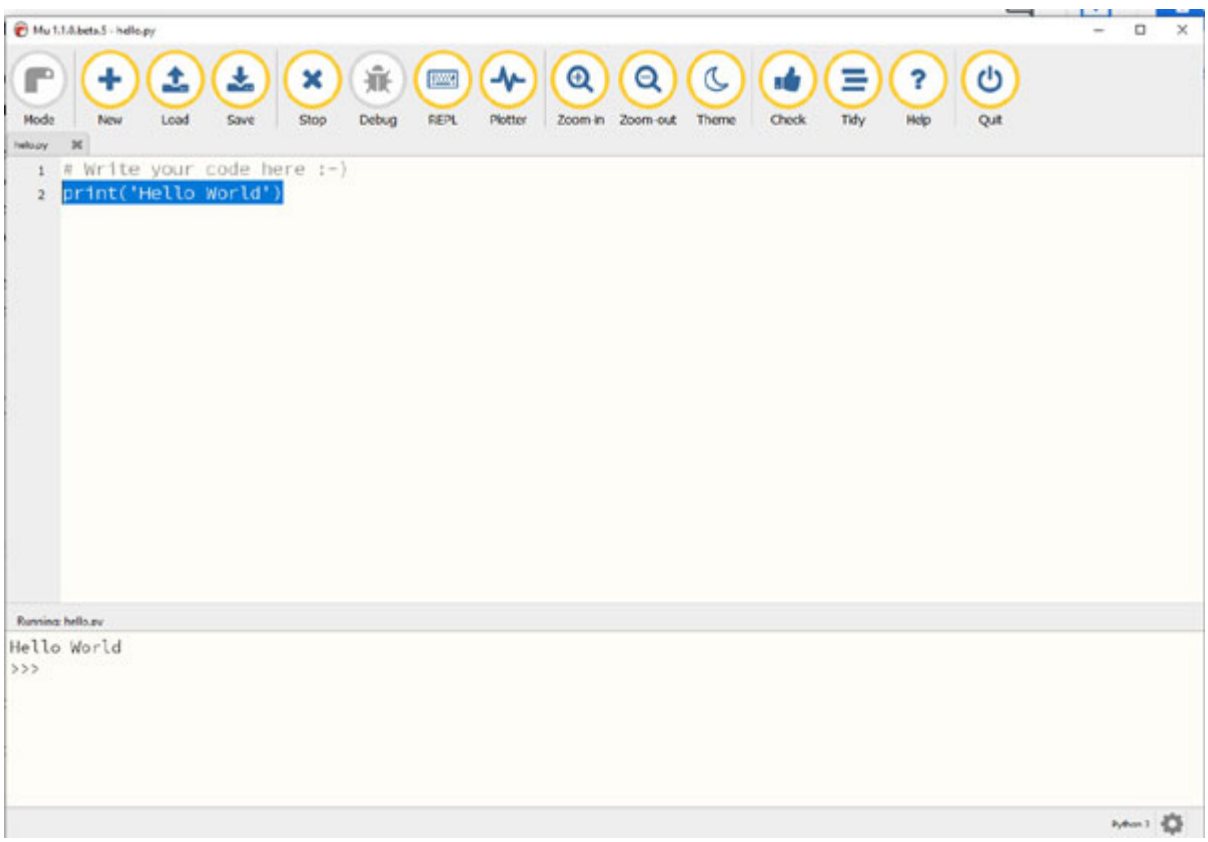


Figure 2.1: *Hello World program*

With Python, you can easily assign values to variables. Following are some examples of values of different data types assigned to variables. In Python,

the naming convention for the variables, functions, classes, and code structure is according to the *PEP 8 style guide*. Variables are *snake cased* and case sensitive as per the following examples:

- **`my_string = "Hello World"`** # *An example of a string*
- **`my_number = 12321312`** # *An example of an integer*
- **`my_float = 3.1415`** # *An example of a float*

Here, we've assigned data to the **`my_string`**, **`my_number`** and **`my_float`**, using the assignment operator `=`. We can use these assigned values by typing them in the Python interpreter.

Python also supports arithmetic operators to perform mathematical operations, such as `+`, `-`, `/`, `*`, `%`. In the following figure, we see some of the examples of the mathematical operations performed in Python:

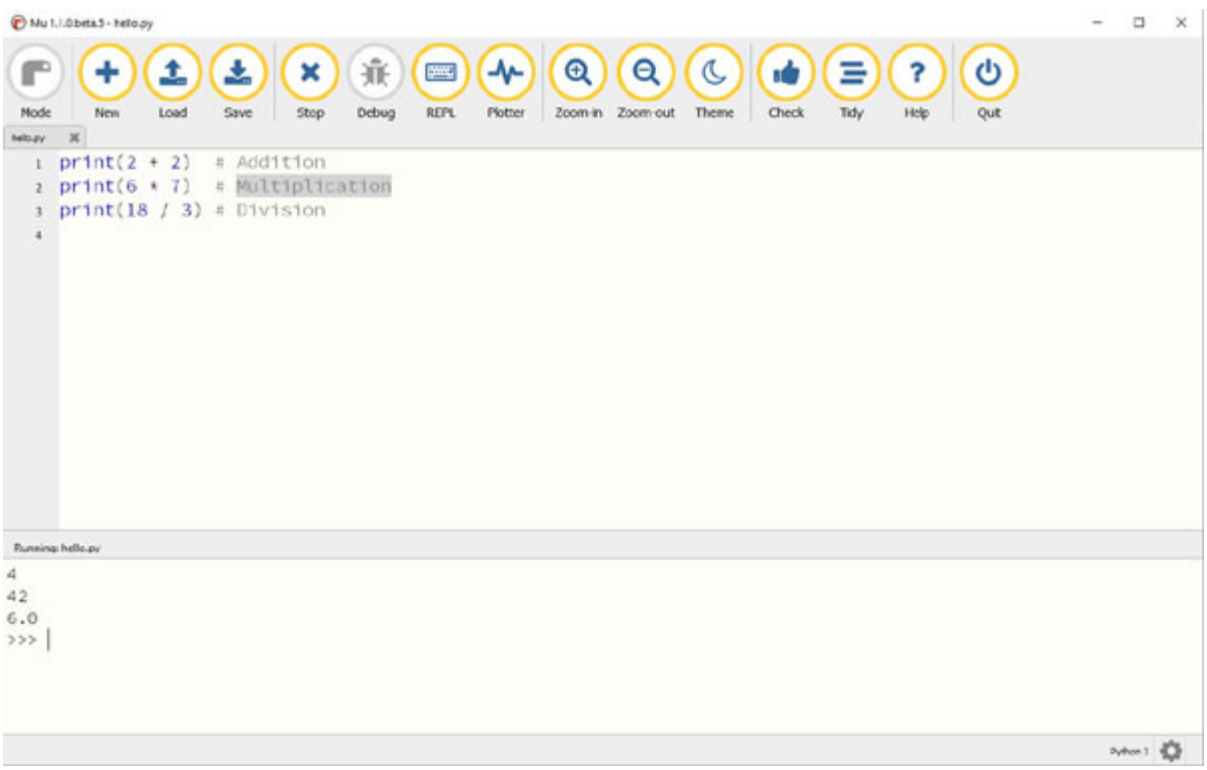


Figure 2.2: *Mathematical operations in Python*

We can also use comparison and logic operators: `<`, `>`, `==`, `!=`, `<=`, `>=`, and statements of identity, such as **AND**, **OR**, **NOT**. The data type returned by this is called a **Boolean** (refer to *Figure 2.3*):

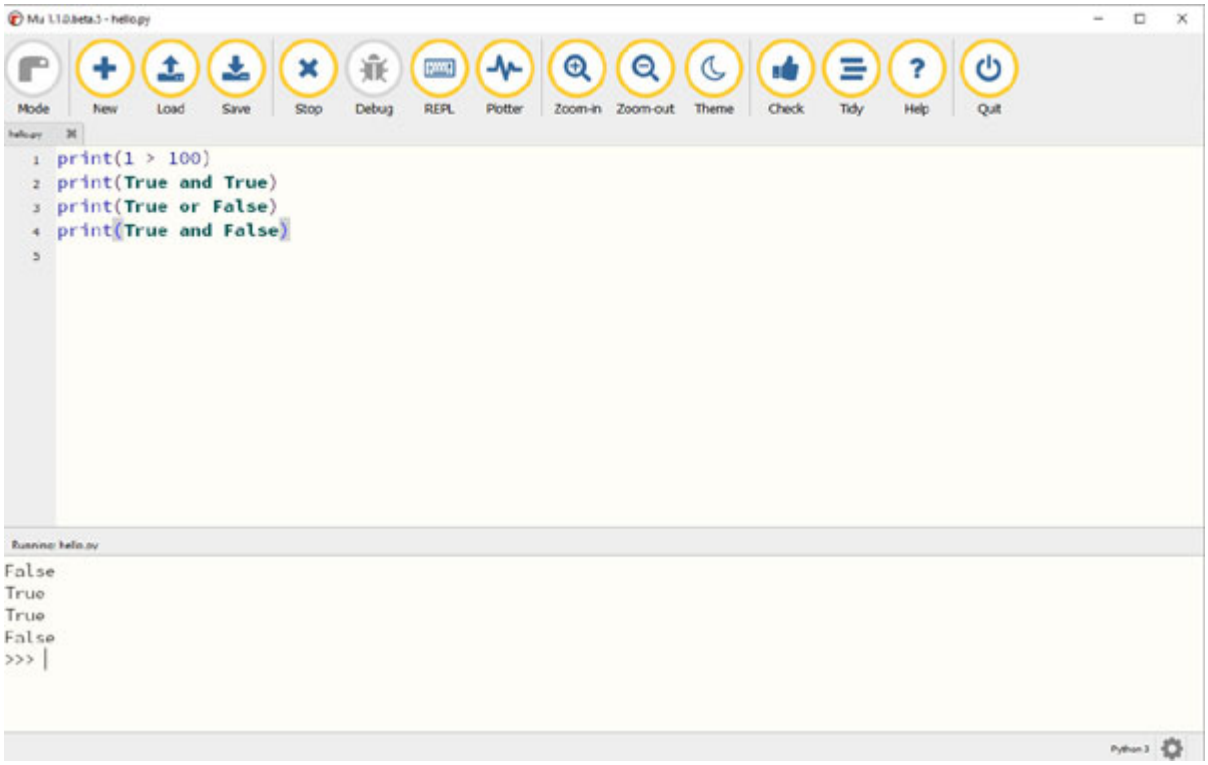


Figure 2.3: *Boolean statement in Python*

Decision statements

Decision-making statements decide the direction of the flow of program execution. In Python, **if**, **else**, and **elif** statements are used for decision making. In Python, **indentation** is used to indicate a block of code instead of brackets and it is very important to use consistent indentation in the Python code. We generally use four spaces per indentation level in Python as per the *PEP 8 style guide*.

if statement

The `if` statement is used to decide whether to execute a certain block of code or not.

Syntax:

```
if (condition):
  # Statements to execute if true
```

In *Figure 2.4*, we see the use of the if statement to check whether a variable is greater than `100` or not. In the case, as the variable value is `1000`, the

`print` statement is executed printing `1000 is greater than 100`:

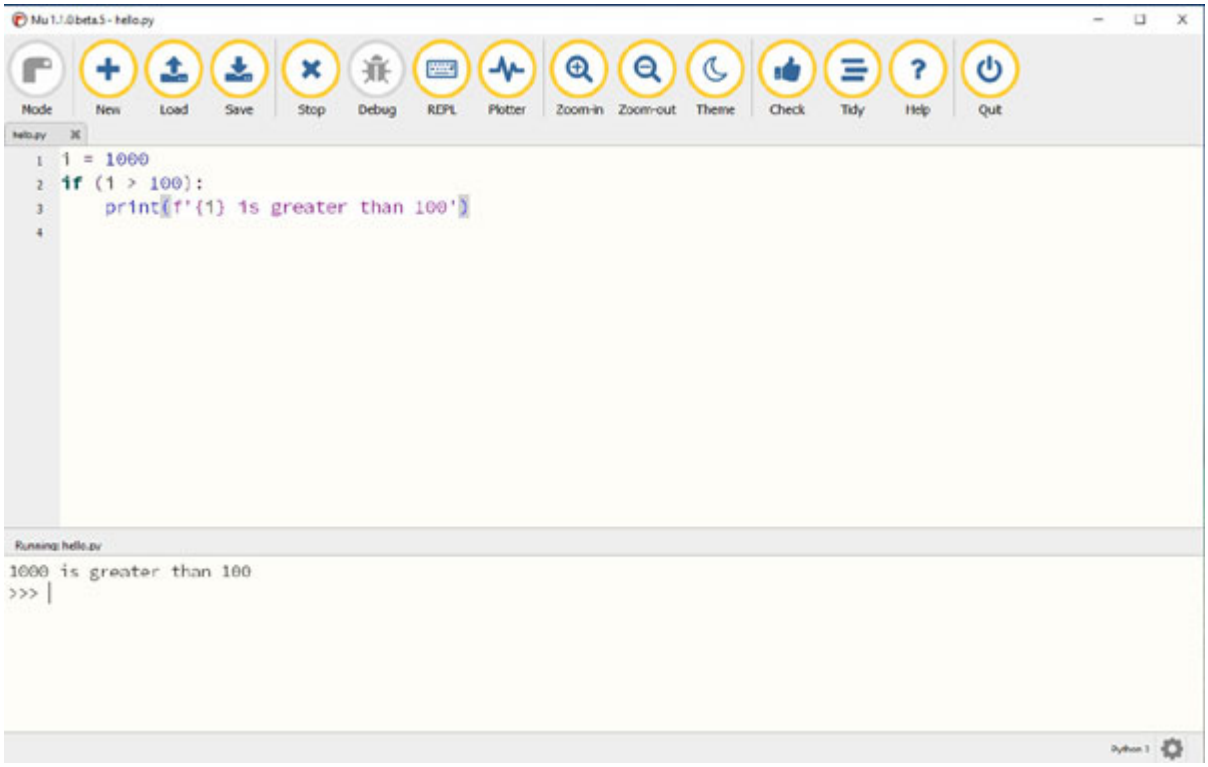


Figure 2.4: *If statement*

if-else

The `else` statement allows you to execute the code when the `if` statement condition is *false*.

Syntax:

```
if (condition):
  # Executes this block if condition is true
else:
  # Executes this block if condition is false
```

In *Figure 2.5*, the variable value is `10`, so the `print` statement `else` statement is executed printing `10 is less than 100`:

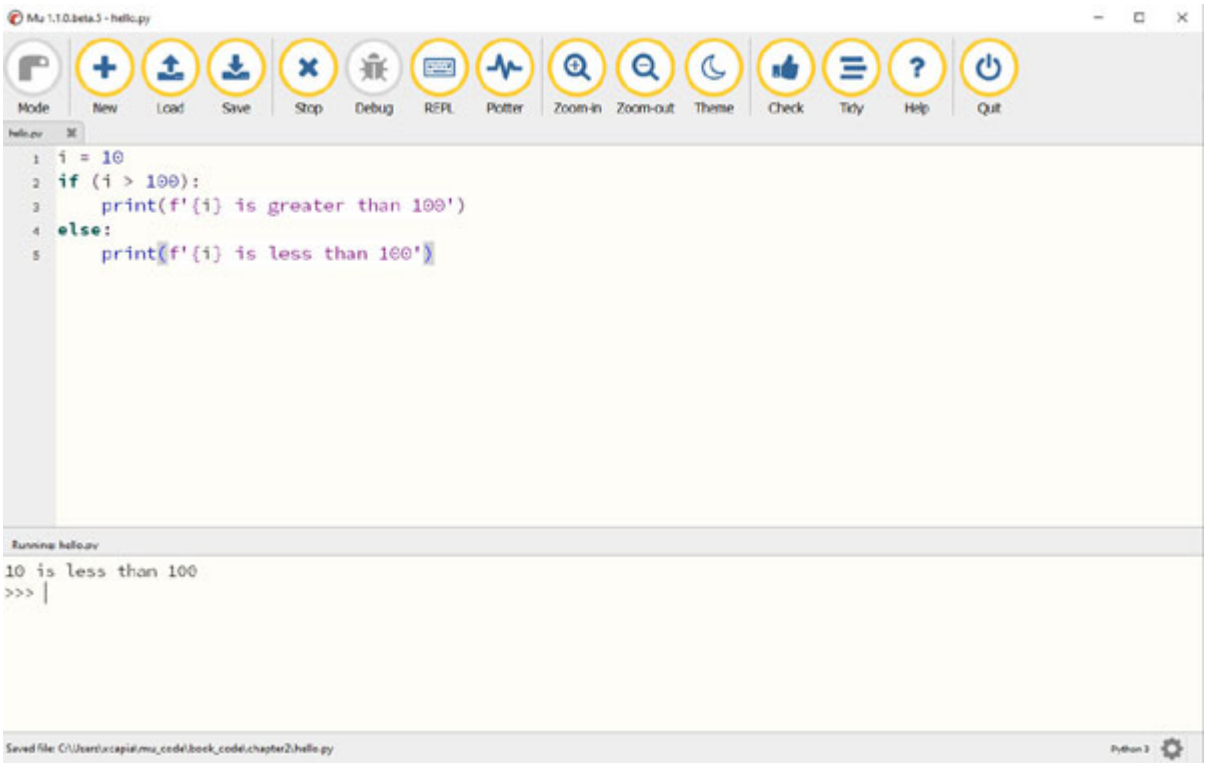


Figure 2.5: *If - else program*

[if-elif-else]

Here, the programmer can decide among multiple options. The **`if`** statements are executed from the top down. As soon as one of the conditions controlling the **`if`** is *true*, the statement associated with that **`if`** is executed, and the rest of the ladder is bypassed. If none of the conditions is *true*, then the final **`else`** statement will be executed.

Syntax:

```
if (condition):
  statement
elif (condition):
  statement
.
.
else:
  statement
```

In *Figure 2.6*, the variable value is `10`, so the `print` statement `elif` statement is executed printing `10 is greater than 1`:

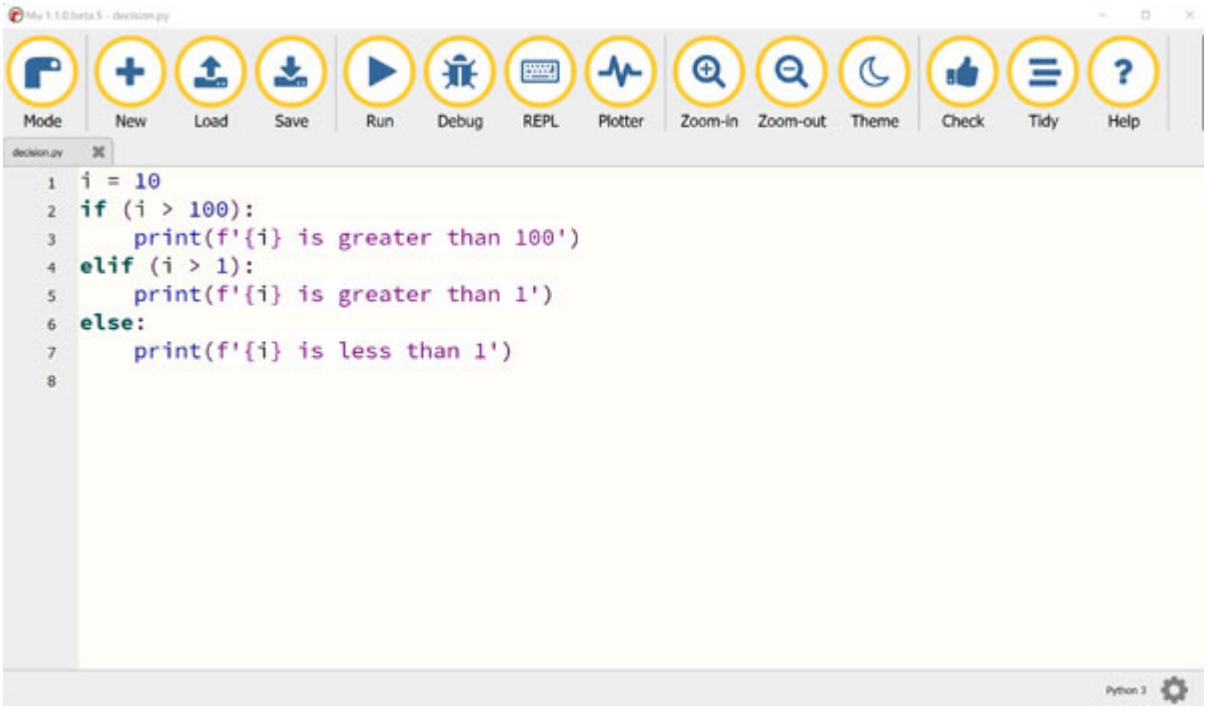


Figure 2.6: *If - elif - else program*

Loops/repetition

There are two types of loops in Python, **for** and **while**.

The for loop

The `for` loops iterate over a given sequence. We can use the `range()` function in Python to loop through a set of code a specified number of times. The `range()` function returns a sequence of numbers, starting from `0` by default, and increments by `1` (by default), and ends at a specified number.

In *Figure 2.7*, we see a simple `for` loop being executed printing numbers from range `0` to `3` using the `range` function:

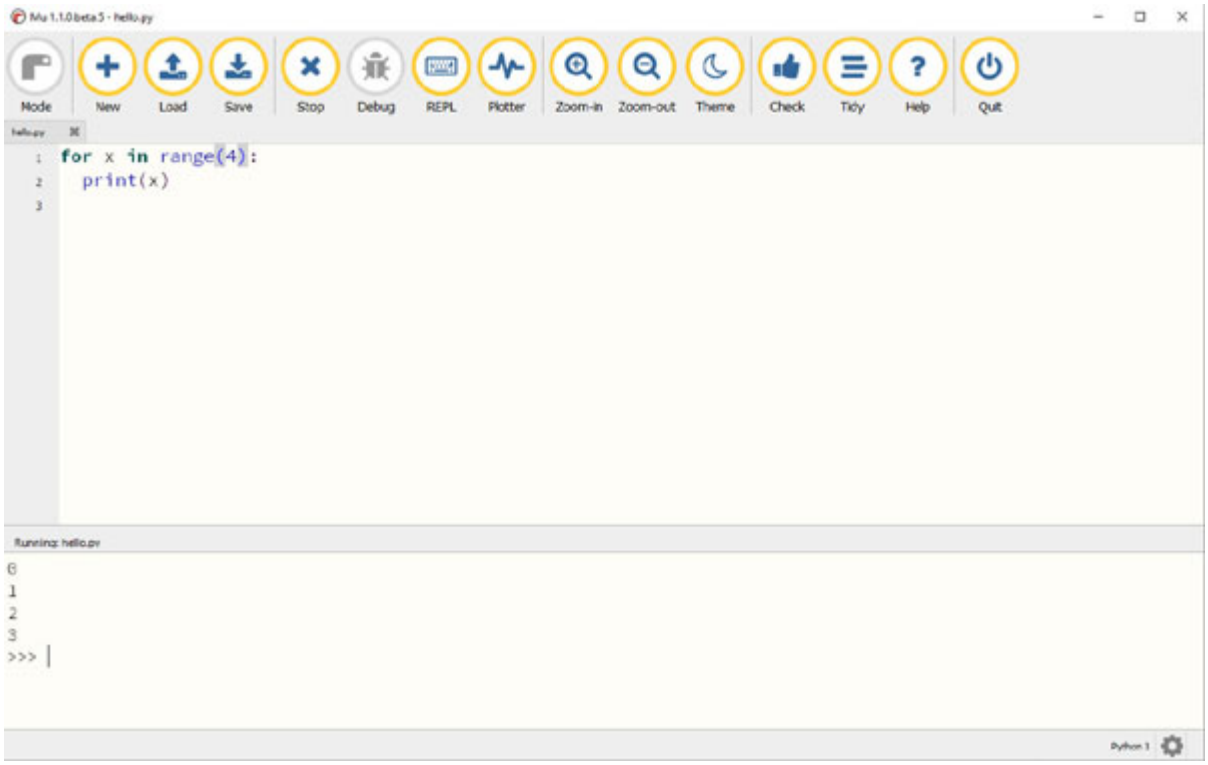


Figure 2.7: *Simple for loop in Python*

while loops

The `while` loops are similar to `for` loops and they repeat as long as a certain *Boolean* condition is met.

In *Figure 2.8*, we see a `while` loop being executed with the initial value of variable `i` as `1` and the *end* condition stating that the loop should run till the value is less than `4`. Inside the loop, we increment the variable by `1` on each iteration. This loop terminates as soon as the variable value reaches `4`:

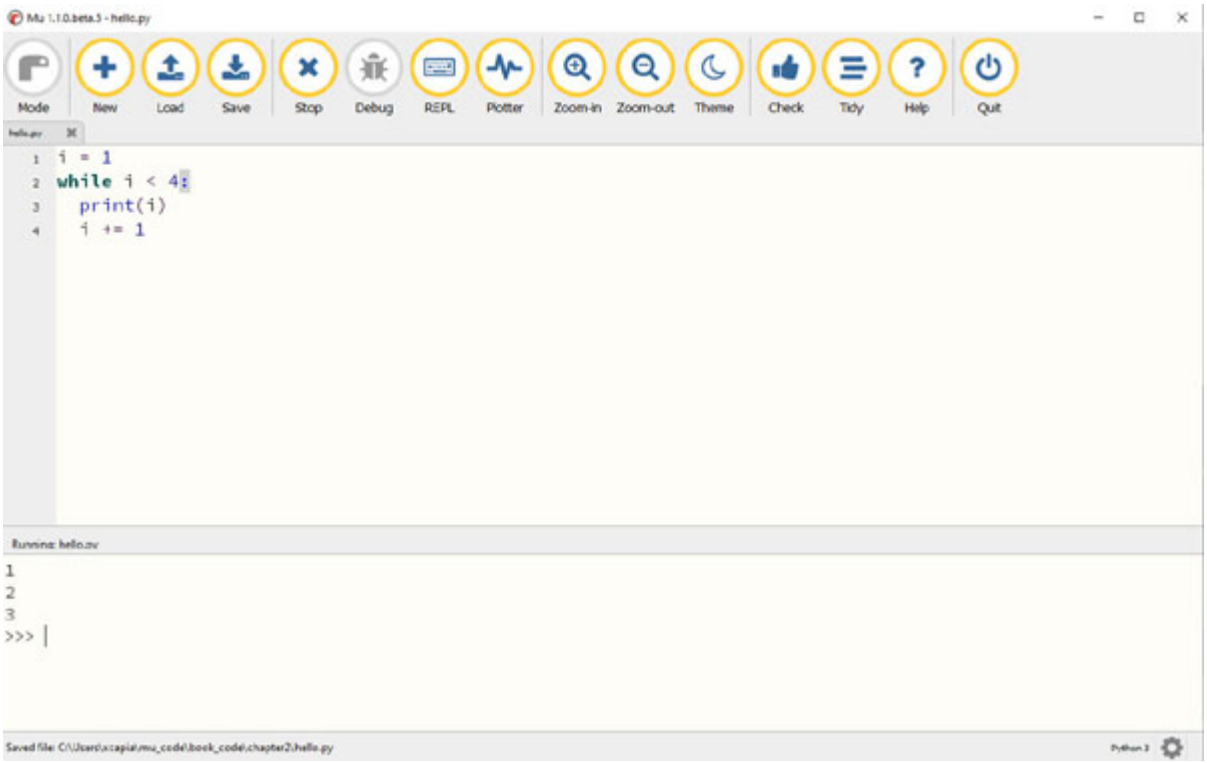


Figure 2.8: *While loop in Python*

The break statement

The **`break`** statement is used to exit from the **`for`** loop or a **`while`** loop. With the **`break`** statement, we can stop the loop before it has looped through all the items.

In *Figure 2.9*, a **`break`** statement is used to exit the loop when the value of the variable is **`2`**. This loop terminates after printing values **`0`** and **`1`**, and it exits the loop as soon as the value reaches **`2`**:

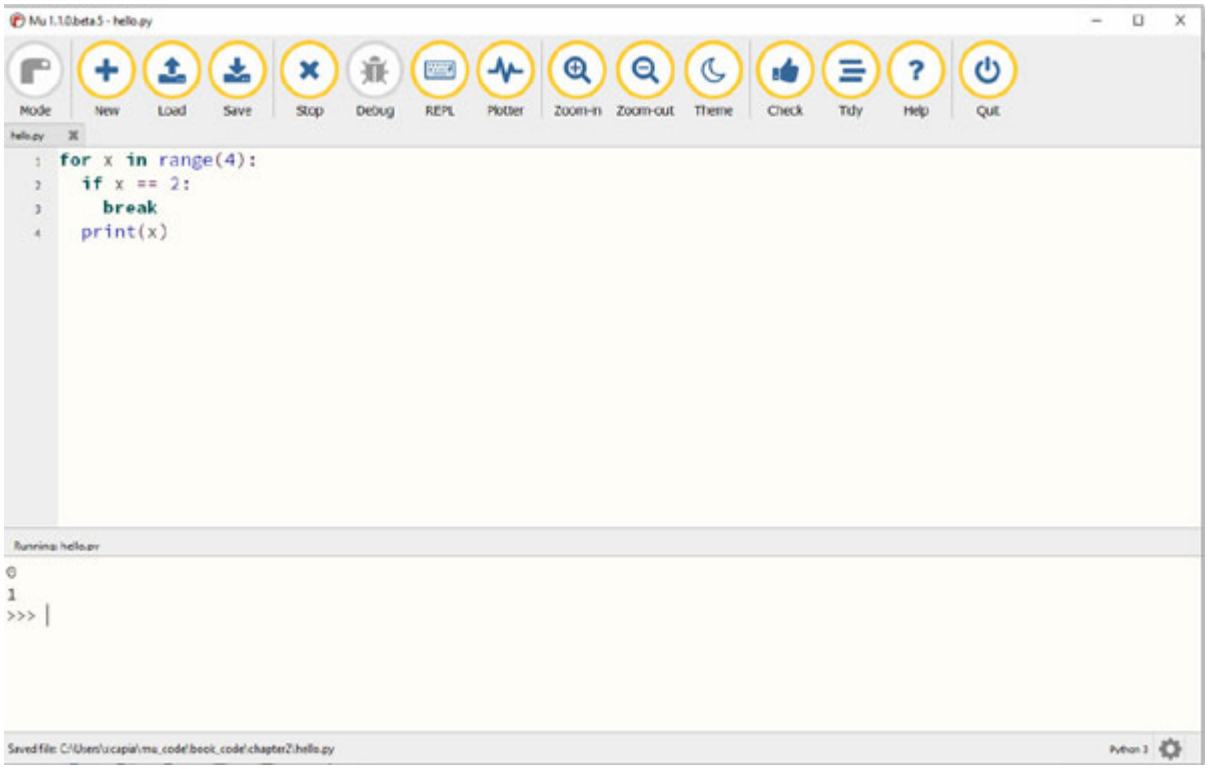


Figure 2.9: *Break statement*

The continue statement

With the **`continue`** statement, we skip the current iteration of the loop and continue with the next iteration of loop.

In *Figure 2.10*, the **`continue`** statement is used to skip printing of the variable when the variable value is `2`. So, the values `0`, `1`, and `3` are printed, and the **`print`** statement is skipped using the **`continue`** statement when the variable value is `2`:

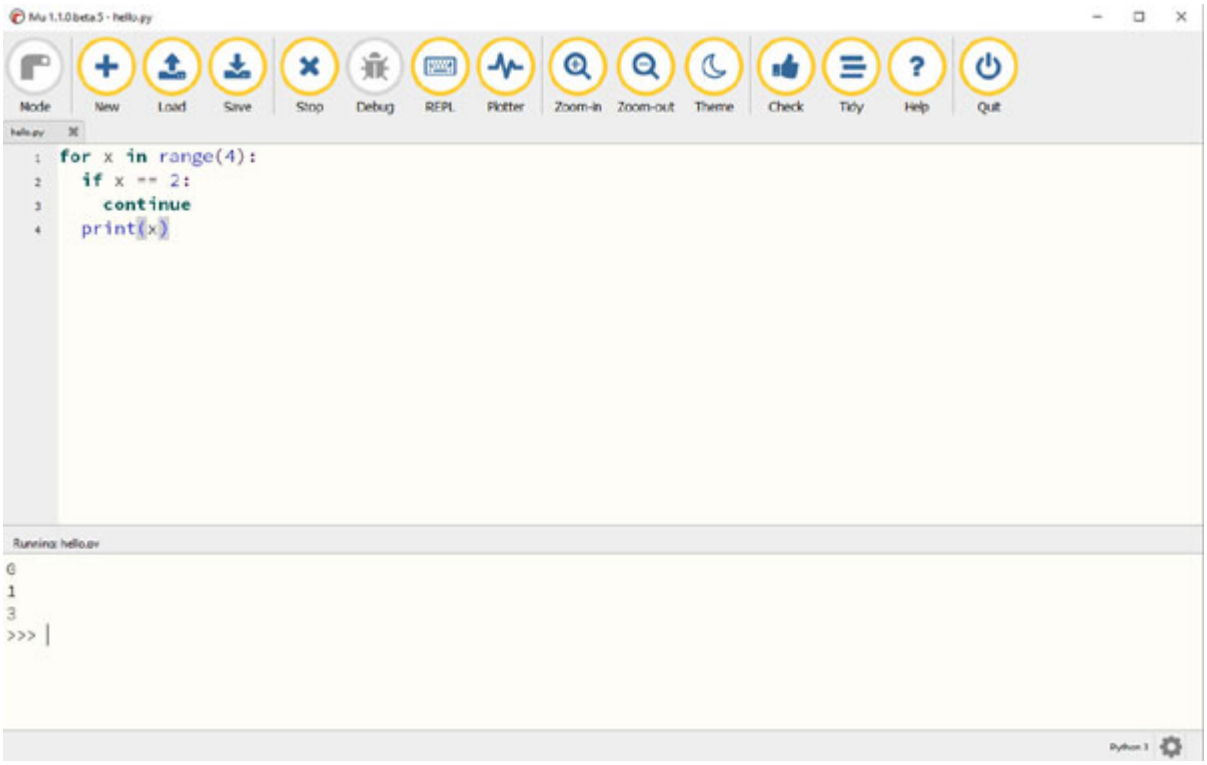


Figure 2.10: *Continue statement*

Data structures

Data plays a very important role in the current work environment. Data structures in Python enable you to store data, retrieve them, and perform operations on them easily. **Lists**, **tuples**, **dictionaries**, and **sets** are four basic types of data structures in Python.

Lists

Lists hold an ordered sequence of elements in Python. Each element can be accessed by an *index*. In Python, indexes start with **0** instead of **1**, so the first element of a list is numbered at **0,** and the last element for a list with n elements in numbered *n - 1*. There is also negative indexing which starts from *-1* enabling you to access elements from the last to first.

Lists are created by placing *comma-separated values* inside parentheses **[]**.

In the following figure, we see several examples of creating and modifying a list in Python:

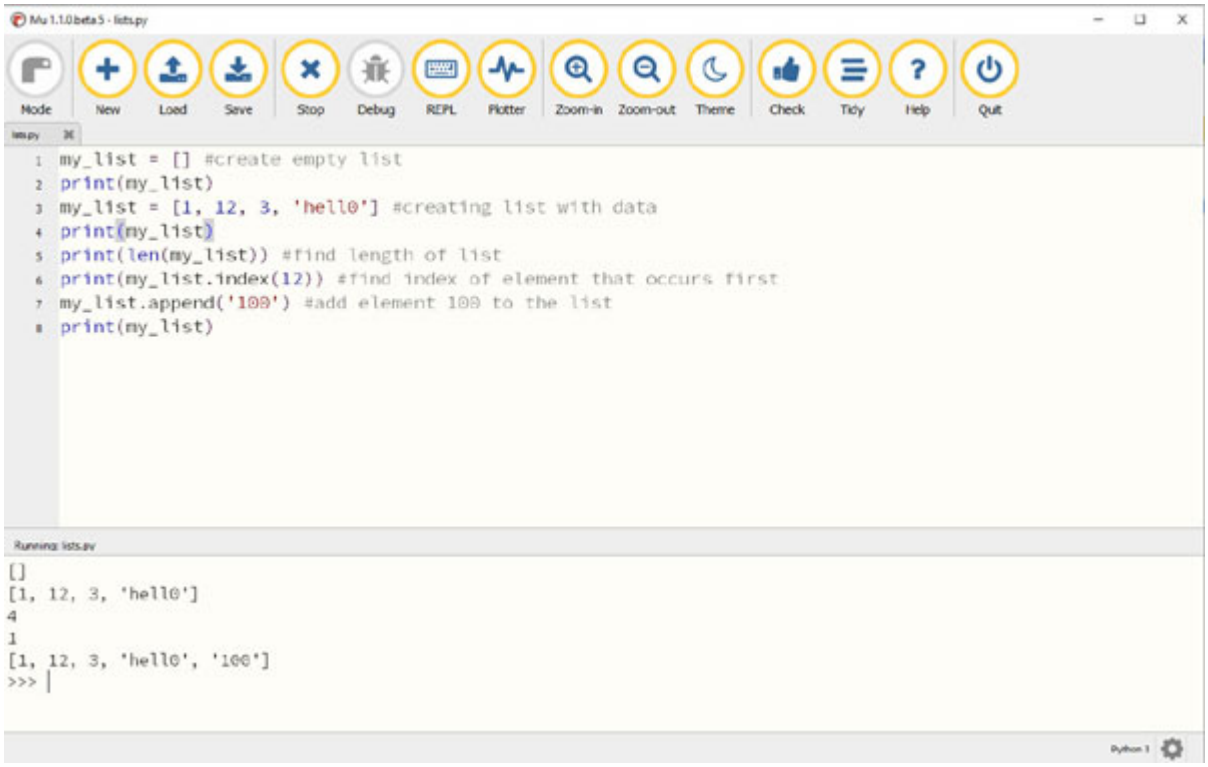


Figure 2.11: *Lists in Python*

A **`for`** loop can be used to access the elements in a list one at a time.

In *Figure 2.12*, the **`for`** loop is used to iterate through each item of the list and then print the items of this list using a **`print`** statement:

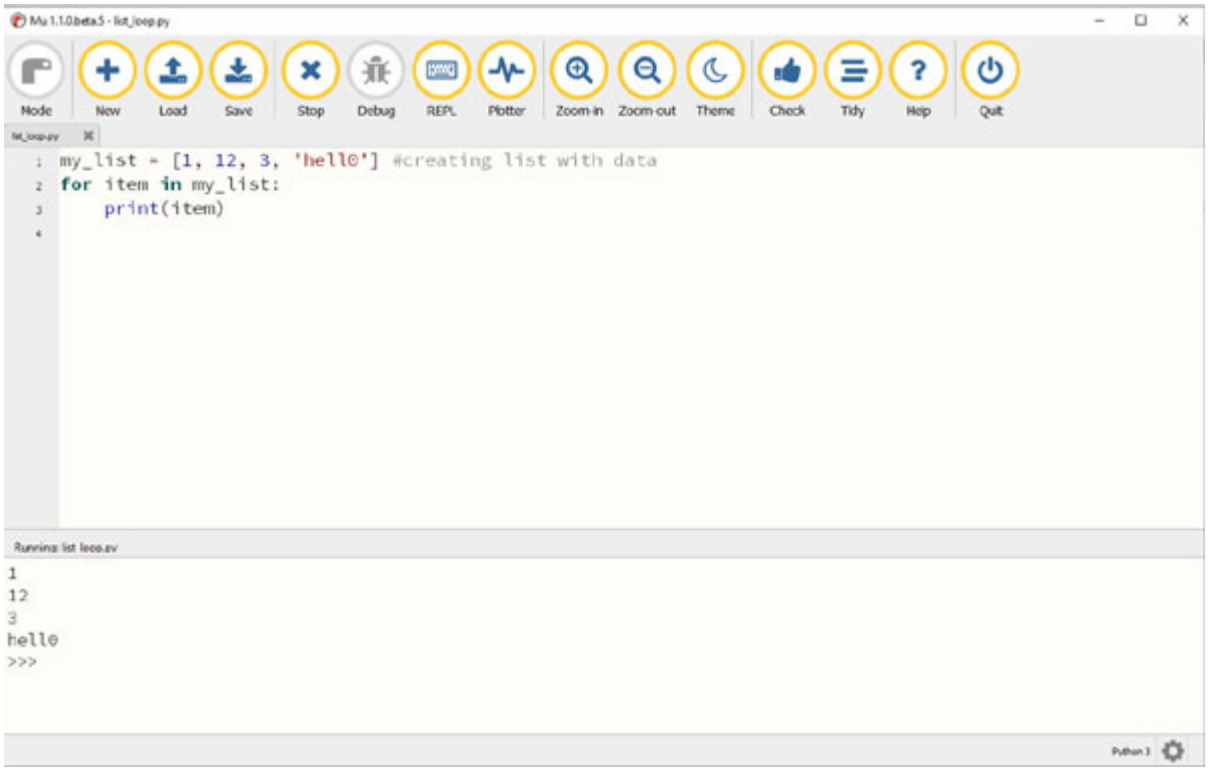


Figure 2.12: *For loop with lists*

Tuples

A **tuple** is similar to a list and is an ordered sequence of elements. However, tuples are *immutable* (they cannot be changed once they are created).

Tuples are created by placing *comma-separated values* inside parentheses **()**. There is no **`append`** method in tuple as it cannot be changed once created.

In the following figure, a tuple is created, and the **`for`** loop is used to iterate through each item of this tuple, and then print them:

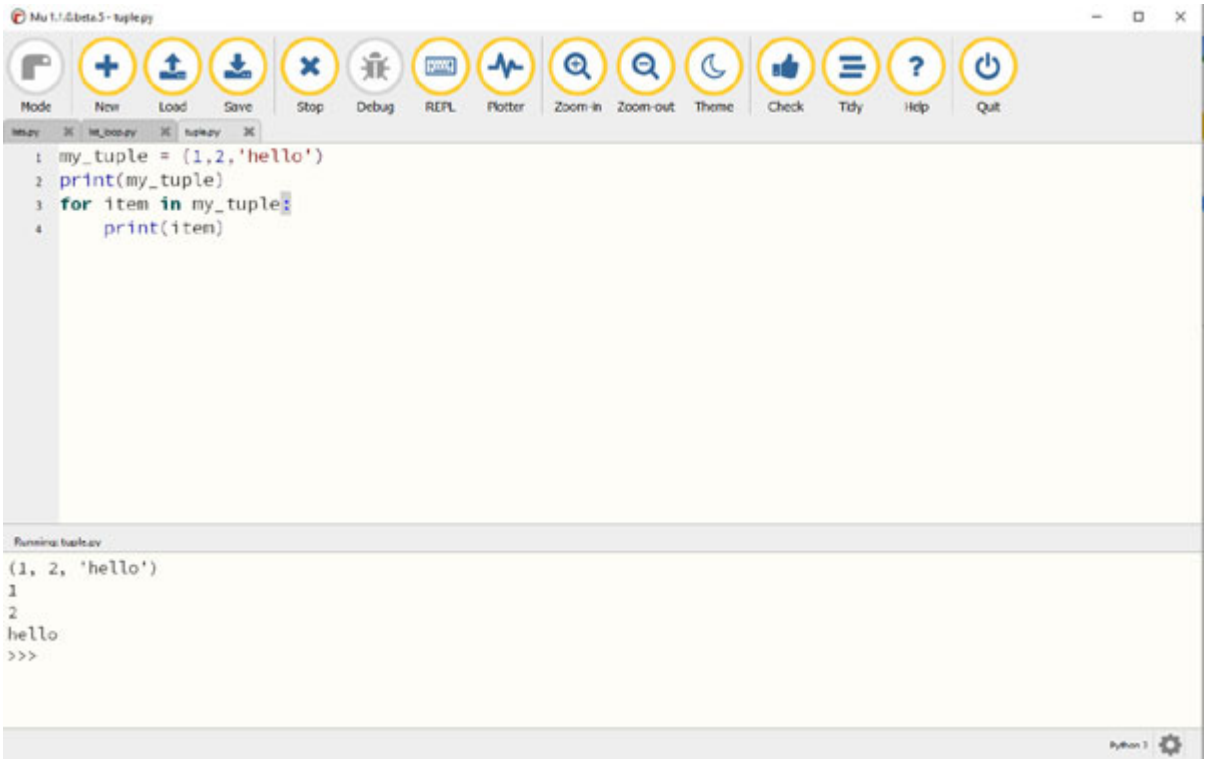


Figure 2.13: *Tuple in Python*

Dictionaries

A **dictionary** is a data structure that stores *key-value pairs*. A simple analogy of a dictionary would be a phone directory where phone numbers are keys and the names would be the values. You can access the name by the dictionary's phone number.

Dictionaries are created by placing comma-separated **`key: value`** pairs inside parentheses `{}`. Dictionaries work similar to lists (but you index them with keys).

In the following figure, we see several examples of creating, accessing, and modifying a dictionary in Python:

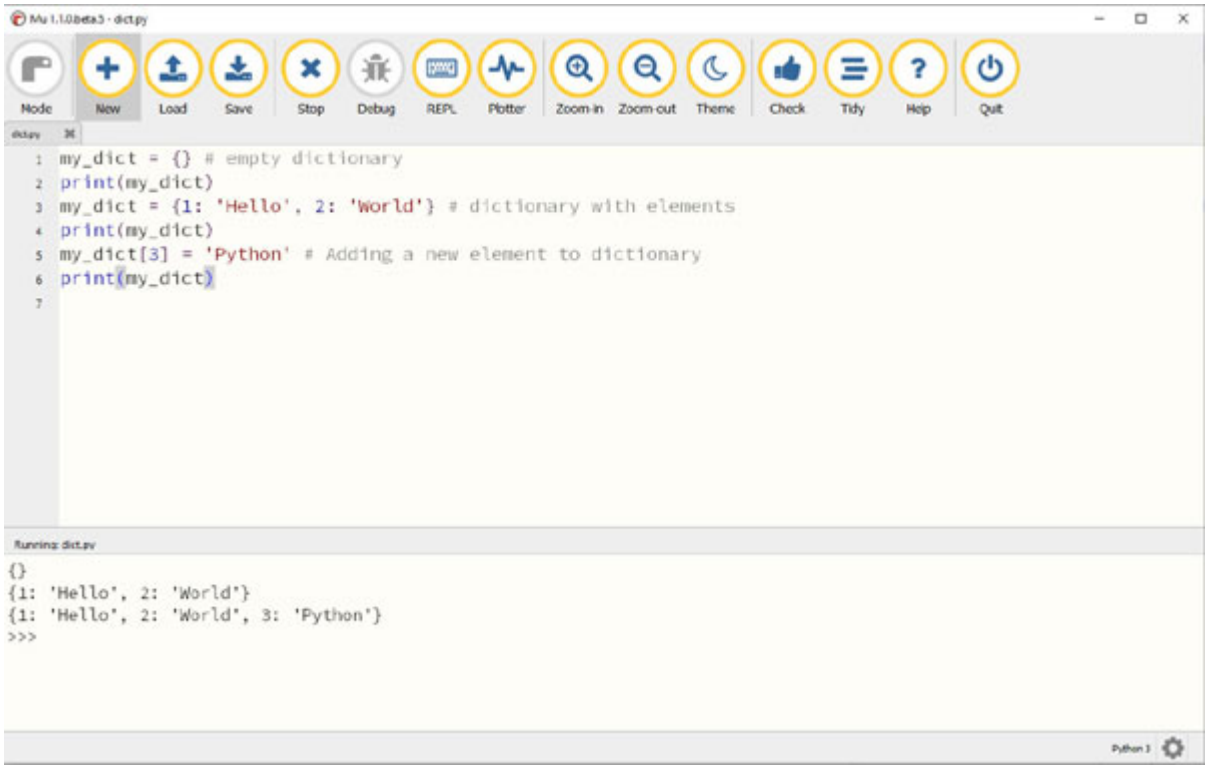


Figure 2.14: *Dictionaries in Python*

A **`for`** loop can be used to access the elements in a dictionary using the following methods:

- **`items()`**: Loop through the **`key: value`** pairs in the dictionary.
- **`values()`**: Loop through the values in the dictionary.
- **`keys()`**: Loop through the keys in the dictionary.

In *Figure 2.15*, we see an example to iterate through a dictionary using dictionary keys, values, or both:

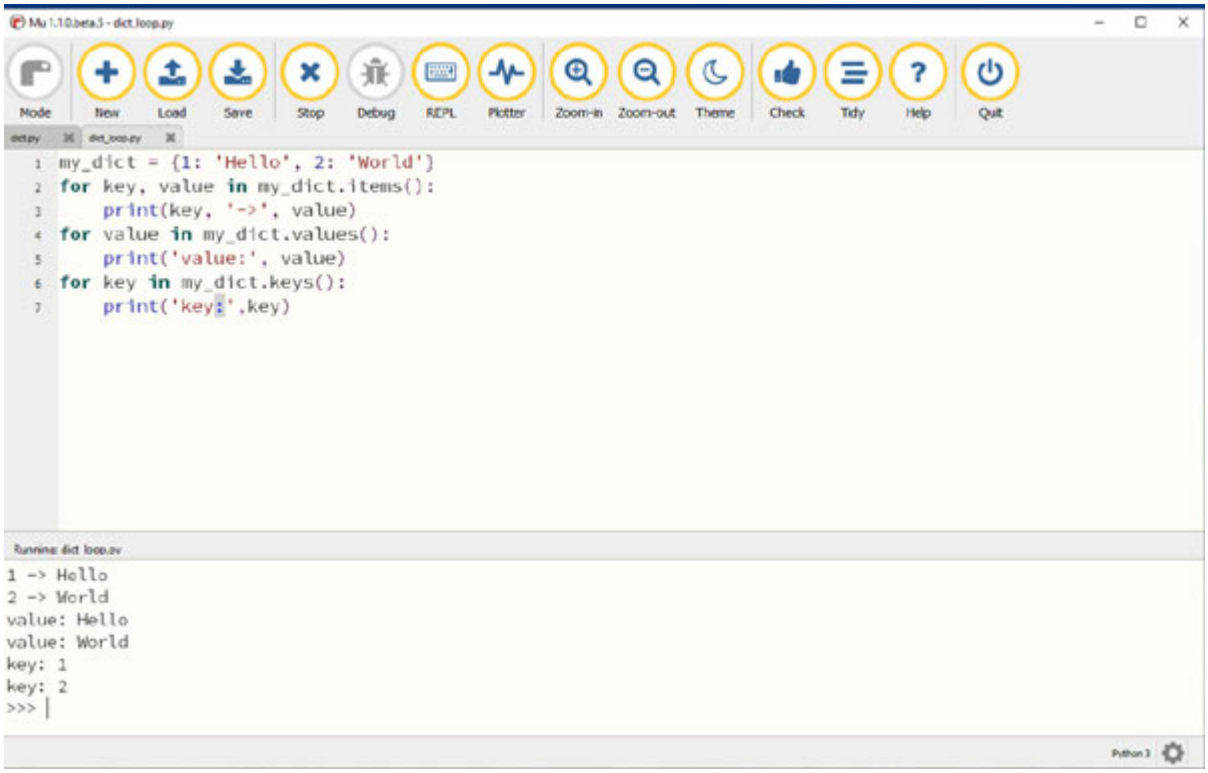


Figure 2.15: *For loops with dictionary*

Sets

Sets are a collection of unordered elements that are unique. They only hold unique values and duplicate values are automatically deleted in the set.

Sets are created by placing comma-separated values inside parentheses **{}**.

In the following figure, we see an example of creating and looping through a set in Python. Notice that the duplicate elements are omitted in the set and a unique set of element list is printed when we loop through this set:

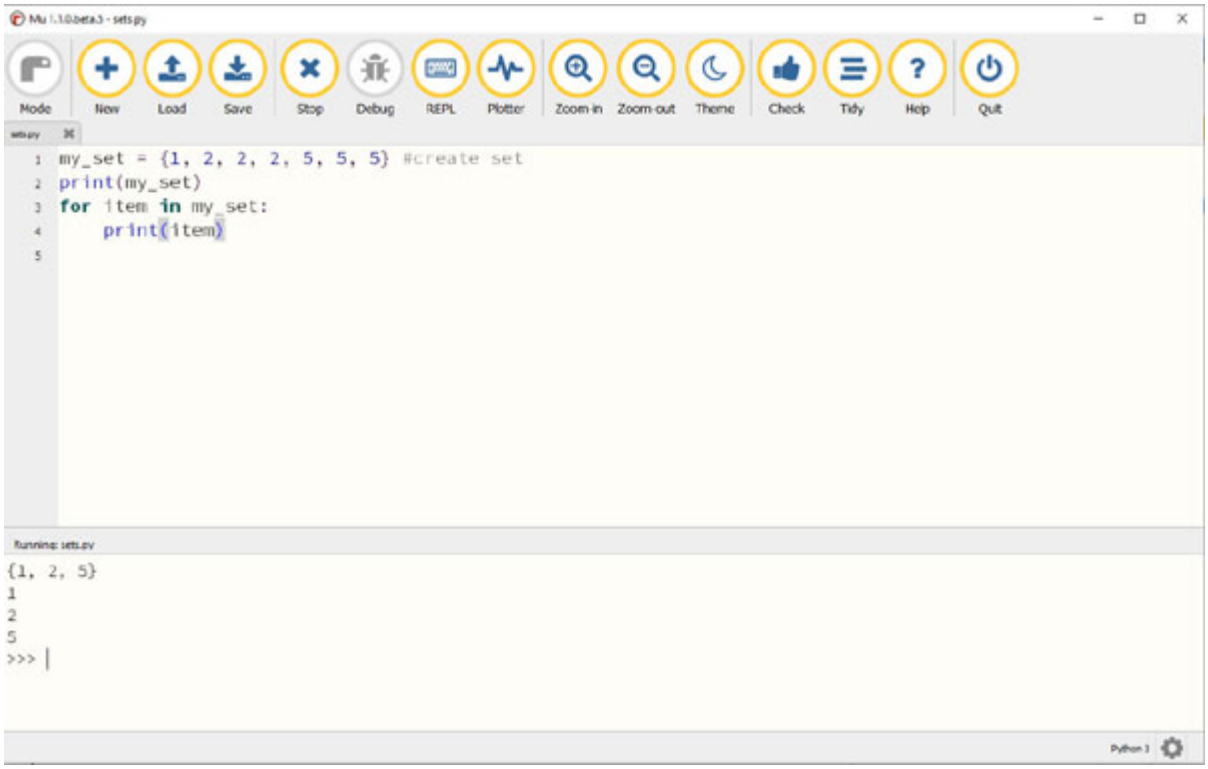


Figure 2.16: *Sets in Python*

Functions

Functions are used to divide the code into blocks, thus allowing you to reuse the code over time. It makes the program easier to understand and allows you to share the code across programs.

Functions in Python are defined using the **`def keyword`**, followed by the function's name. Functions are called by their name and passing appropriate arguments in the function definition.

Example syntax is as follows:

```
def func_name(arguments):
  func_operation
```

In the following figure, we see an example of creating a simple function to add two numbers in Python. The function is called inside the **`print`** statement with the two numbers we want to add as an argument inside the function:

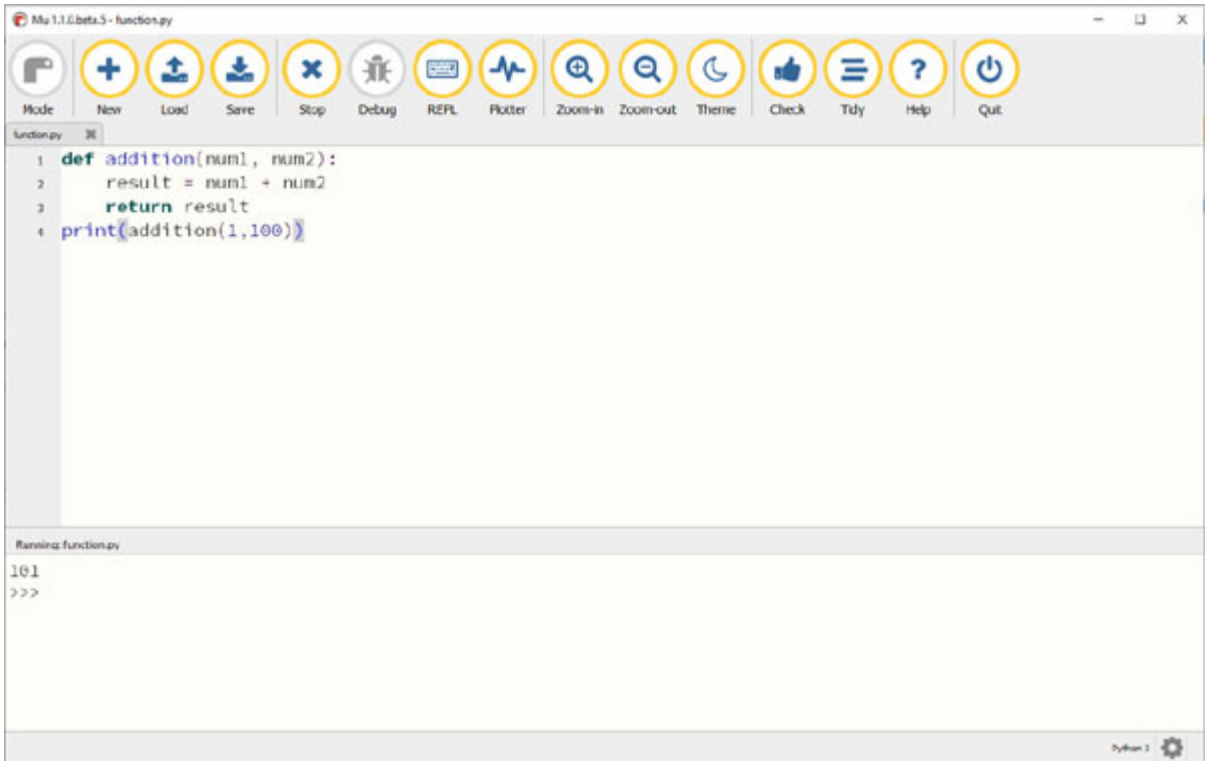


Figure 2.17: *Simple addition function in Python*

Libraries, modules, or packages

A Python library or module is a file containing reusable definitions and statements. Python libraries are the best way to share the code among applications. There are thousands of Python libraries created and maintained by different communities, and companies. The **Python Package Index** (https://pypi.org/) provides an extensive collection of reusable repositories of software for the Python programming language. We will be frequently using Python libraries to help us build the required work automation programs throughout this book.

Modules can be added using **Mu** (installing third-party packages with Mu) or `pip` installer. Modules are imported using the `import` keyword followed by the module name.

In the following figure, we import a Python library called `math`. After importing this library, we can use the functions available within the library. The function definitions and example of using the function for a library are found in the library documentation which in this case is the Python `math` library documentation (**https://docs.python.org/3/library/math.html**):

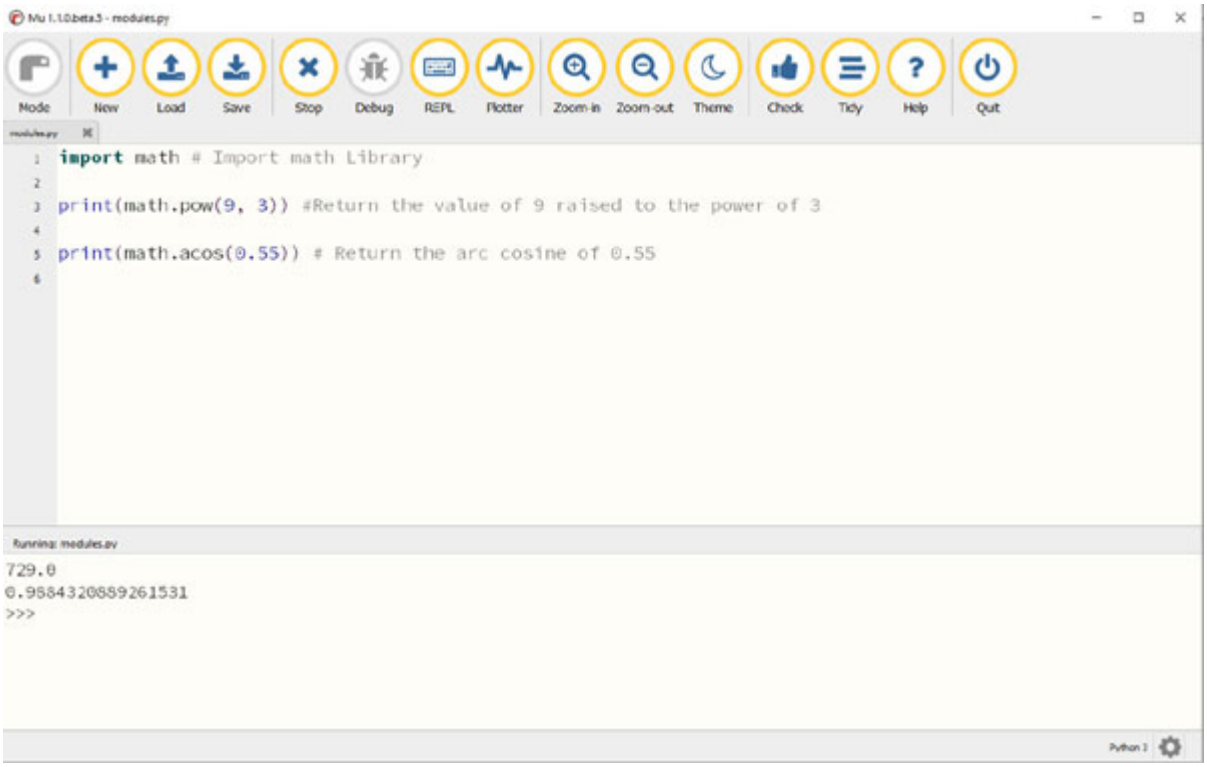


Figure 2.18: *Importing and using Python math library*

Conclusion

In this chapter, we discussed the basics of the Python programming language with examples of decision statements, data structures, loops, functions, and Python Libraries. We also went through the syntax of Python to help you with the basic programming knowledge required for building and editing work automations.

In the next chapter, we will discuss the automation mindset essential to identify and automate your daily tasks. We will also discuss how Python can be used as a tool for building automation and discuss some real-world scenarios where Python is used for work automation.

Further reading

There are a lot of online Python tutorials and resources available on the Internet to get started with learning the Python programming language. Some popular tutorials are given in the following table:

Resource name	Link

The official Python tutorial	**https://docs.python.org/3/tutorial/index.html**
Data structures you need to learn in Python	**https://www.edureka.co/blog/data-structures-in-python/**
Real Python tutorials	**https://realpython.com/**
w3schools Python tutorial	**https://www.w3schools.com/python/default.asp**
Tutorials point Python tutorial	**https://www.tutorialspoint.com/python/index.htm**
Short introduction to programming in Python	**https://datacarpentry.org/python-ecology-lesson/01-short-introduction-to-Python/**

Table 2.1: *Tutorials for learning Python*

Questions

1. How does a **while** loop work in Python?
2. What are different data structures in Python?
3. How to you stop a **For** loop in Python?
4. What is a package in Python?

CHAPTER 3

Automation Mindset – Python as a Tool for Automation

Introduction

In this chapter, we will discuss the mindset needed to be successful in implementation of automation within your organizations. We will go through the process of identifying and prioritizing automation opportunities. We will also discuss the ways to share the developed automations with the wider organization once they are created.

Structure

In this chapter, we will cover the following topics:

- Mindset for automation
- Common processes for automation
- Identifying business processes

Objectives

After studying this chapter, you will be able to identify the automation opportunities in your organization. You will also have the right mindset to decide when to implement the automation and when to look for other solutions for optimizing your workflow.

Mindset for automation

Automation mindset involves a way of working where we look for continuous improvement of existing processes and finding opportunities for automation. It is a way of reimagining the whole process of doing a task, or an entire workflow, and finding opportunities to make it more efficient. You

need to be comfortable for change and look for tools and solutions to help the process run more efficiently.

In the next section, we will discuss some of the common processes and tasks that can be easily automated with Python.

Common processes for automation

The best starting process for automation are the ones which are highly repetitive in nature and take a substantial amount of time of your overall workload.

The most common automations opportunities come from the following three subsets of categories:

1. **Data entry**: The data entry process involves tasks which require you to enter data from one application to another. These tasks are highly manual in nature and can be easily automated with Python. The main candidates for data entry automations include:
    a. **Filling up forms**: Any task that requires repetitive filling up of forms for single or multiple data sources.
    b. **Sending similar emails**: Tasks where you have to send bulk emails or similar emails to a lot of people.
    c. **Copying data between two systems**: Any task requiring duplication of data between multiple systems.
    d. **Maintaining the ERP and CRM systems**: Tasks involving data entry to ERP and CRM systems.
    e. **Updating legacy systems**: Tasks involving working with legacy systems and updating data into these systems.
    f. **Entering data to in-house systems**: Any task where you have to work and maintain the in-house proprietary systems.
2. **Data extraction**: Data extraction processes involve work where you have to extract data from different file formats to be consumed by other teams or applications. These tasks can be easily automated saving a lot of time with day-to-day work tasks. Almost every job involves some tasks to extract data from different files. The main candidates for data extraction automations include:

a. **Extracting customer details**: Tasks involving extracting customer details from emails, documents, and other systems.
b. **Converting PDF data to Excel sheet**: Tasks involving extracting data from PDF documents by converting it to the Excel format.
c. **Extracting data from reports**: Tasks involving extracting data from external and internal reports such as financial reports, press releases, legal reports, and corporate reports.
d. **Extracting data from images**: Tasks involving data extraction from scanned or online images.

3. **Data gathering**: Data gathering processes involve work where you have to gather data from multiple sources such as websites, files, and applications. These tasks generally involve collecting, cleaning, and collating data from multiple sources, and performing some analysis on them. The main candidates for data gathering automations include:

    a. **Gathering stock prices**: Tasks involving collecting stock prices data from stock exchange websites and other market data systems.
    b. **Performing market research**: Tasks involving collecting particular pieces of information from social media sites, competitor websites, or media documents.
    c. **Gathering website data**: Any task involving collecting data for any website on the Internet.
    d. **Online reports extraction**: Tasks involving data extraction from online HTML-based reports.

There are also process discovery and process mining tools that can help you with discovering the processes that should be prioritized and automated. We will discuss some of these tools in the next section.

Identifying business processes

Business process discovery is a common way to identify processes. Business process discovery involves techniques and ways to manually or automatically construct the organization's business processes and their variations.

The processes can also be identified using process documentation or time-based analysis of activities performed in the organization. Once the processes are identified and the steps involved in the process are listed, use the following checklist as a guide to select best candidates for automation:

- Requires more than a couple of hours of manual time to complete.
- Processes which have defined steps/rules.
- Processes which are repetitive.
- Processes running frequently, at least once a month.
- Have multiple steps involved in completing the process.
- Work involves multiple data files such as Excel, text, PDF files.
- Work involves dealing with legacy systems.
- High accuracy required on the task.
- Process requires high level of documentation.
- Process has high risk of human error due to complexity or number of steps involved.
- Process that is expensive in terms of time, resources, and other intangible assets.
- Process can be easily automated.

The following figure shows time-based analysis for work performed to identify most suitable applications for automation. Further analysis can be performed based on user interviews to identify the final process for automation.

Figure 3.1 displays the time-based analysis across different applications which help with the identification of the most time-consuming tasks performed by the user:

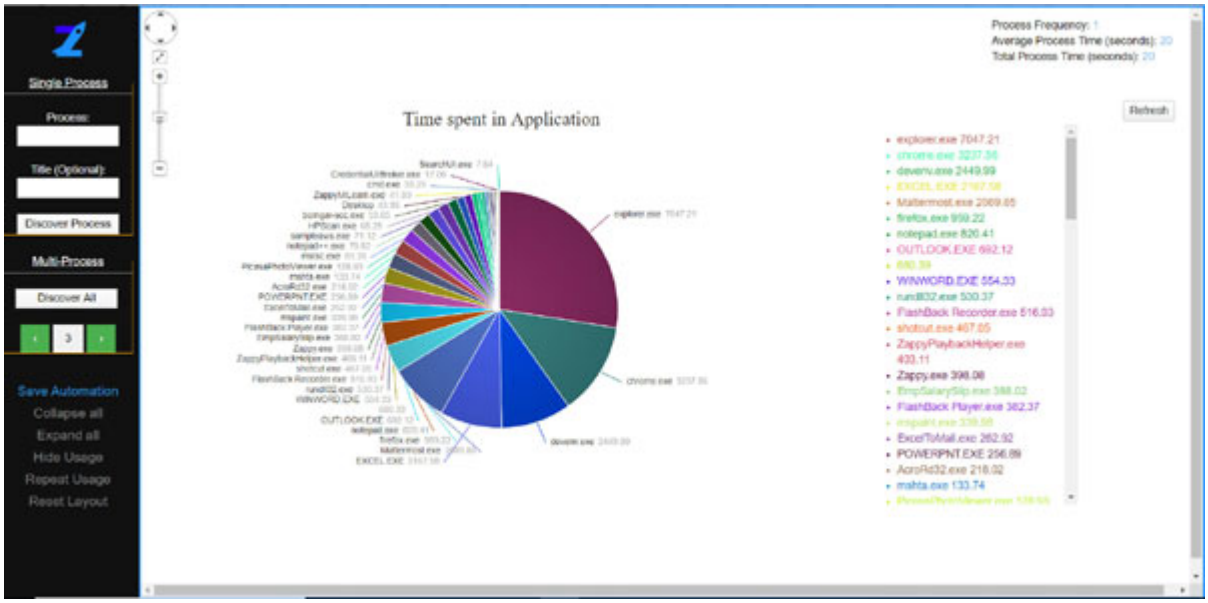


Figure 3.1: *Time spent across applications*

Figure 3.2 displays the process map for various processes performed by the user, the frequency of these processes, and the time taken to perform these processes:

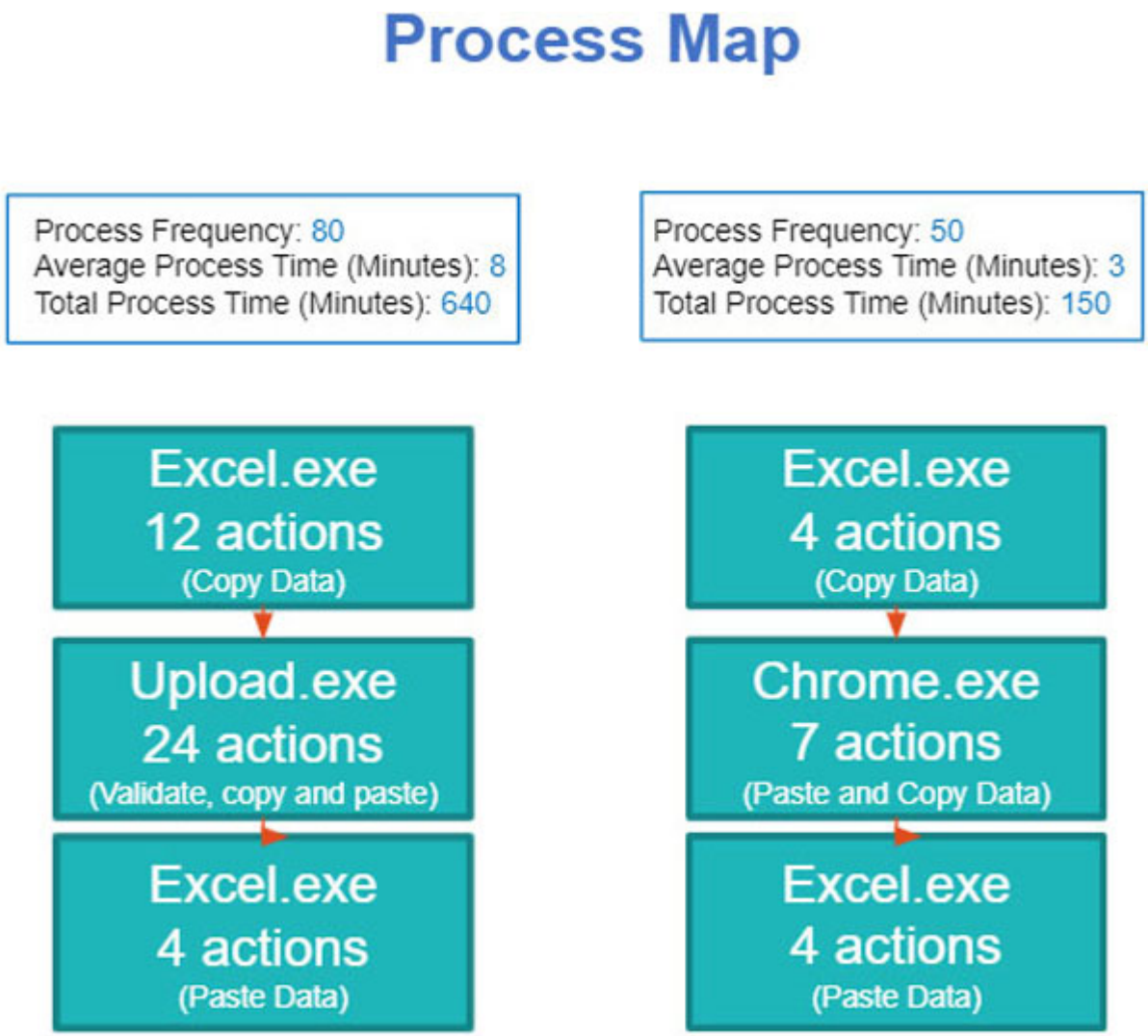


Figure 3.2: *Process map of application*

Figure 3.3 displays the time-based analysis across the **`explorer.exe`** application and shows the place where the user spends the most amount of time in this application. We can do this time based analysis across multiple teams and the whole organization:

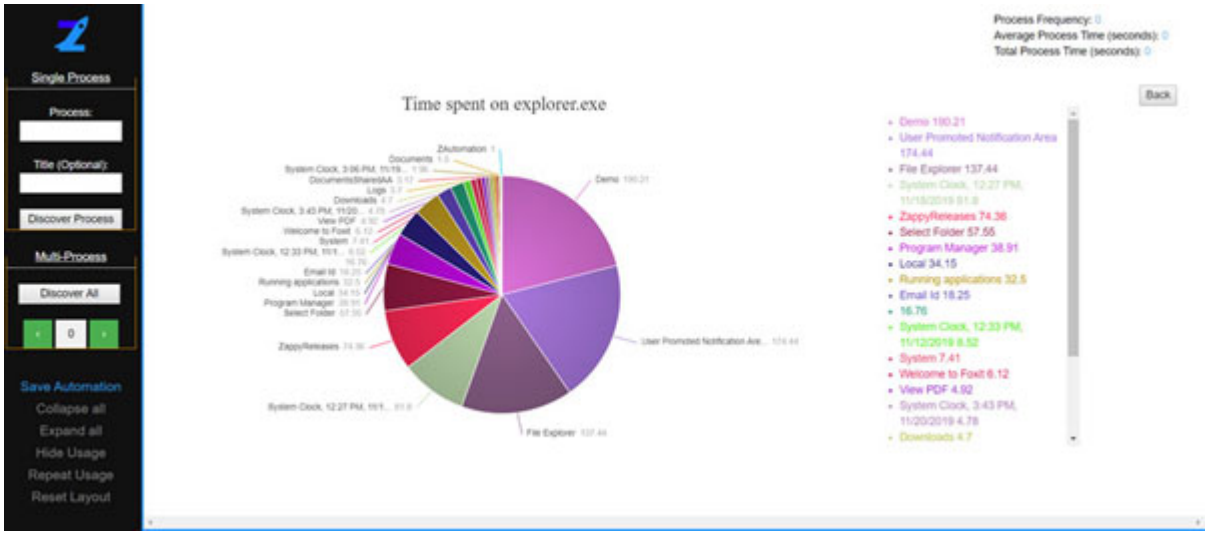


Figure 3.3: *Time spent on explorer*

There are also process mining and process discovery software that generates process maps based on recorded steps, documentation, and existing organizational methods of work. Any data consisting of a unique Id (helpful in grouping tasks belonging to the same task), name of activity (description of the tasks taking place), and timestamp (the time the task took place) are called **event logs**. The event logs are used to discover the underlying process model. **PM4Py** (**https://github.com/pm4py/pm4py-core**) is a process mining package for Python and is used extensively to discover business processes.

Conclusion

In this chapter, we discussed the importance of an automation mindset to be successful in improving the quality of work. We have gone through the list of most common processes that can be automated with Python. We also looked at various tools and techniques available to help us identify the most likely candidates for automation.

In the next chapter, we will take a look at the various techniques to implement automation with Excel-based data files and spreadsheets. We will discuss the Python modules to help with Excel-based dataset automation and various examples of automations that can be performed for these types of files.

Further reading

There are few good tools available which can be used to perform process discovery and process mining to find opportunities for improving the processes within the organization.

Resource name	Link
Process discovery in 2021: what it is and how it works	**https://research.aimultiple.com/process-discovery/**
PM4Py - Process mining for Python	**https://pm4py.fit.fraunhofer.de/**
Introduction to process mining	**https://towardsdatascience.com/introduction-to-process-mining-5f4ce985b7e5**
Celonis process mining	**https://www.celonis.com/**

Table 3.1: *Resources on process discovery and process mining*

Questions

1. What is a process discovery tool?
2. What are the processes that you should not automate?
3. How do you find automation opportunities within an organization?
4. What is the mindset required to build automations?

CHAPTER 4

Automating Excel-Based Tasks

Introduction

In this chapter, we will discuss ways to automate Excel workflows, including creating, writing, and updating the Excel documents. We will also discuss the data manipulation techniques with Excel and CSV documents.

Structure

In this chapter, we will cover the following topics:

- Installing the library to read/write Excel
- Creating Excel documents
- Reading Excel documents
- Updating a workbook
- Sample Excel-based automation
- CSV file-based automation

Objectives

After studying this chapter, you will gain the knowledge and understanding of the Python library for manipulating Excel files. You will also be familiar with code snippets of how to automate Excel-based tasks such as reading, writing, and updating a workbook. You will see a few common examples of Excel-based tasks that can be automated using Python.

Installing the library to read/write Excel

We will use **`openpyxl`** (the most popular Python library to read/write Excel files) to automate Excel-based tasks. It allows you to read, write, and update a workbook in a very simple way:

1. To install `openpyxl`, use the `mu` package manager. Type `openpyxl` and click on `OK`, as shown in the following figure. We use `3.0.9` in this book, so to import the same version, type `openpyxl==3.0.9` in the package manager. Later versions should work as well with the examples in this chapter:

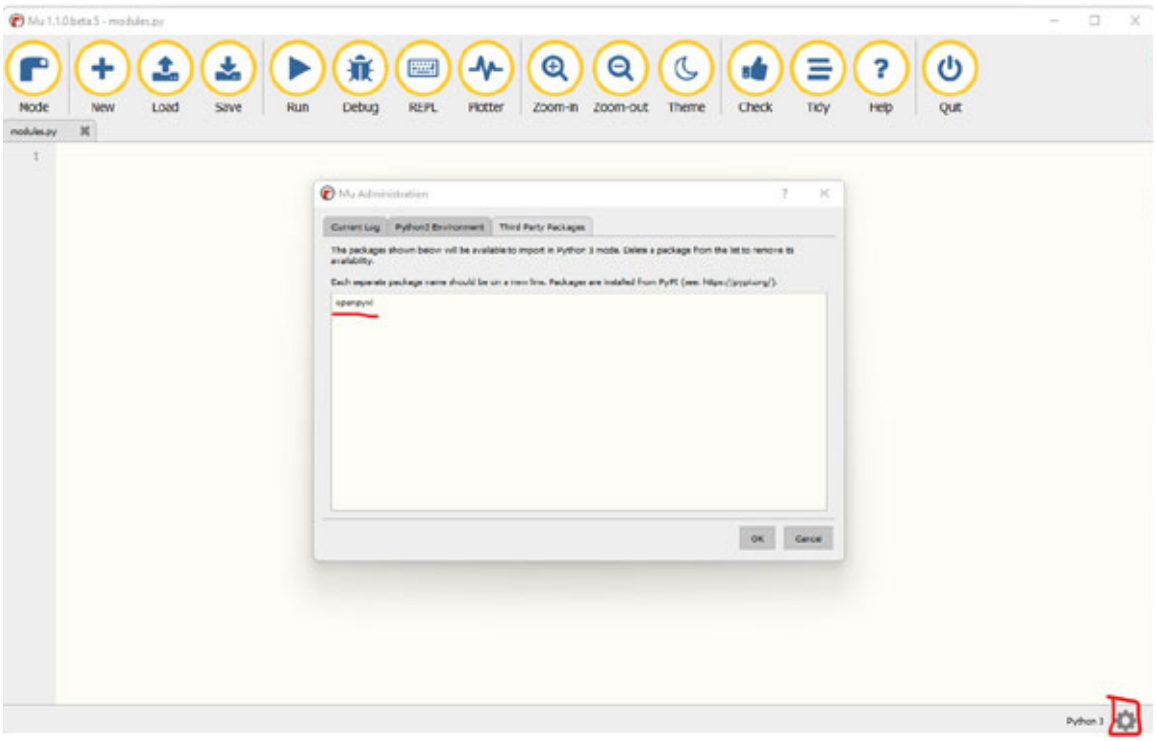


Figure 4.1: *Mu settings option*

2. After clicking on the `OK` button, you will see the message that the `openpyxl` package is being installed on the computer as shown in *Figure 4.2*:

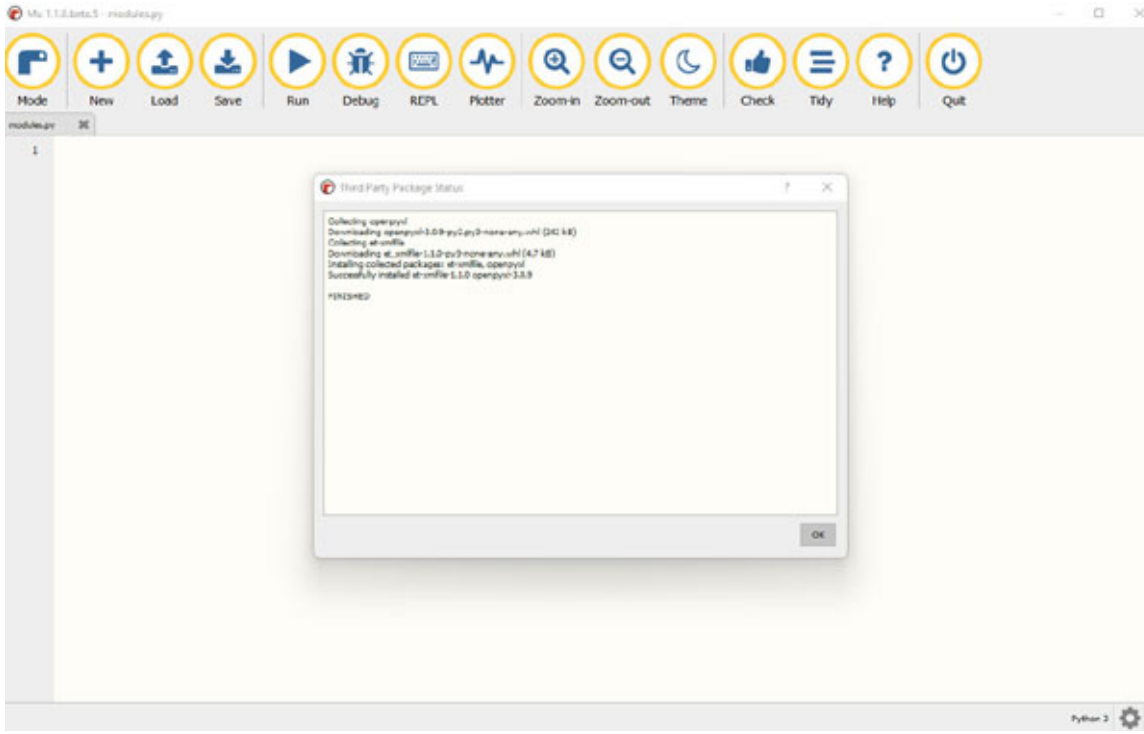


Figure 4.2: *Openpyxl being installed in Mu*

3. You can verify if the **`openpyxl`** is installed properly by importing the library and printing the library version by using the **`__version__`** property as shown in *Figure 4.3*:

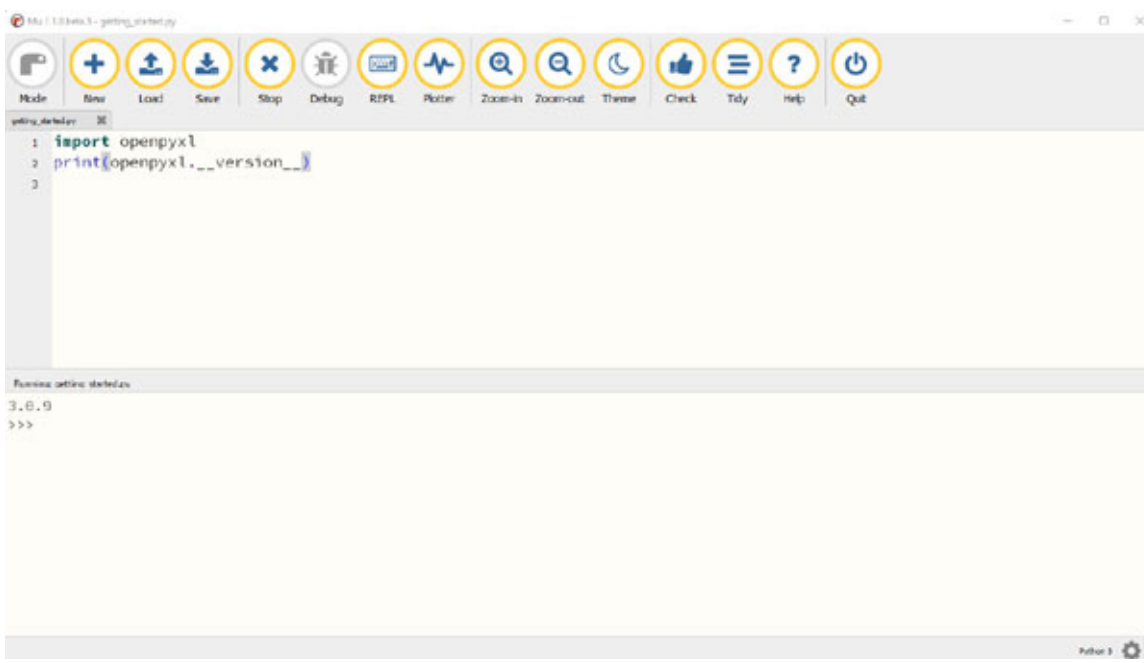


Figure 4.3: *Importing Openpyxl module*

Creating Excel documents

In this section, we will look at an example to create a new Excel workbook and write some data into the workbook.

We can create a new workbook using the **`Workbook()`** function in the **`openpyxl`** library. We can add some data to the working book by accessing the active worksheet, and selecting the row and column by their names. To access *row 1* and *column 1,* use **`A1`** as an index, where **`A`** represents *column 1* and **`1`** represents *row 1,* as shown in the following figure. The location where the file can be saved for the workbook is in the same folder where the script is residing by default:

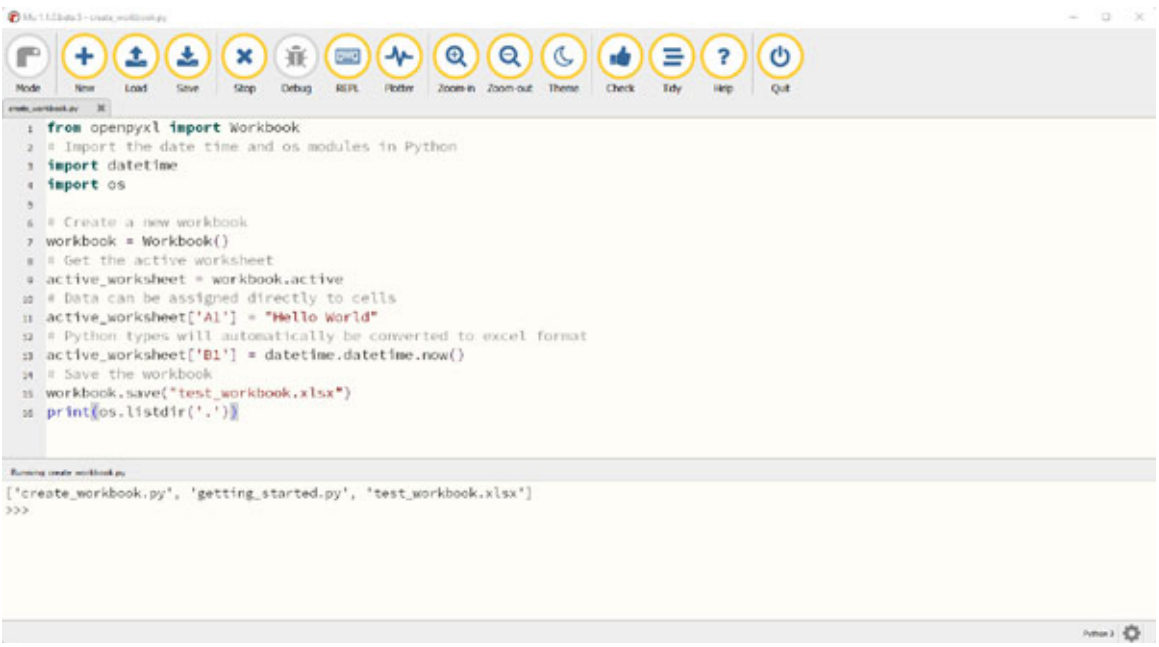


Figure 4.4: *Creating new workbook*

As soon as you save the new workbook, you can access it from the file explorer, and open it from the Excel application:

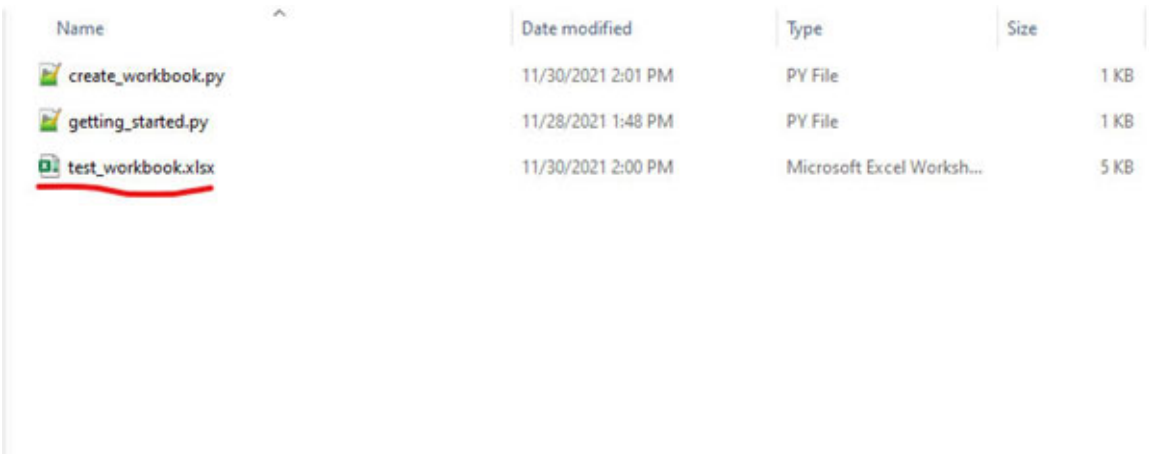


Figure 4.5: *New workbook being saved in the code directory*

After opening the file in the Excel application, you can see the data **`Hello World`** and the date when the program is executed is added in the worksheet, as shown in the following figure:

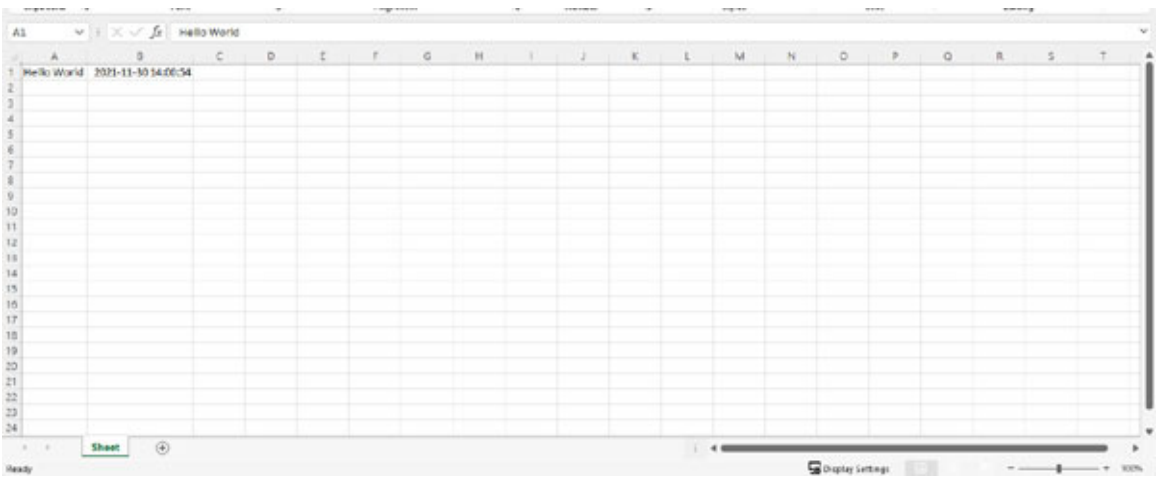


Figure 4.6: *Data being added to the new workbook*

Reading Excel documents

You can read and loop through the data using the Python **`openpyxl`** library. There are a few different ways to iterate through the data depending on your needs.

You can slice the data with a combination of columns and rows. To access a value of a cell, use the **`.value`** method. For example, you can access data in *row 1* and *column 1* by using the assessor **`A1`**, or by using the **`.cell`** function with row and column number as an argument as shown in the following figure:

```
from openpyxl import load_workbook

# load the test workbook
workbook = load_workbook(filename="test_workbook.xlsx")

# Note the str() function is used to convert the return type to string
print("Sheetnames are: " + str(workbook.sheetnames))
sheet = workbook.active

# Access sheet element value by id
print("Value of cell A1: " + sheet["A1"].value)
# Access sheet element value by row column id
print("Value of cell with row as 1 and column as 2 is: " + str(sheet.cell(row=1, column=2).value))
```

```
Sheetnames are: ['Sheet']
Value of cell A1: Hello World
Value of cell with row as 1 and column as 2 is: 2021-11-30 14:00:53.856000
>>>
```

Figure 4.7: *Access cell values from Excel workbook*

You can also loop through the whole row or columns by using the ***range*** function in the format of *Row or Column 1:Row or Column 2* to get the cells between columns and rows as shown in the following figure:

```
from openpyxl import load_workbook

# load the test workbook
workbook = load_workbook(filename="test_workbook.xlsx")
sheet = workbook.active

# Access columns A and B - gives cell tuple as output
print(sheet["A:B"])

# Access row 1
print(sheet[1])
```

```
((<Cell 'Sheet'.A1>,), (<Cell 'Sheet'.B1>,))
(<Cell 'Sheet'.A1>, <Cell 'Sheet'.B1>)
>>>
```

Figure 4.8: *Printing cell tuples*

There are also multiple ways of using normal Python generators to go through the data. The main methods you can use to achieve this are the ***.iter_rows()*** and ***.iter_cols()*** functions. These functions take the arguments ***min_row*** for the starting row number, ***max_row*** for the ending row number, ***min_col*** for the starting column number, and ***max_col*** for the ending column number. In *Figure 4.9*, you can see an example where we are looping through rows and columns of ***test_workbook*** using the ***.iter_rows()*** and ***.iter_cols()*** functions:

```
from openpyxl import load_workbook

workbook = load_workbook(filename="test_workbook.xlsx")
sheet = workbook.active

# Using iterator to get rows values
for value in sheet.iter_rows(
    min_row=1, max_row=2, min_col=1, max_col=3, values_only=True
):
    print("Rows Values:" + str(value))

for value in sheet.iter_cols(
    min_row=1, max_row=2, min_col=1, max_col=3, values_only=True
):
    print("Column Values:" + str(value))
```

```
Rows Values:('Hello World', datetime.datetime(2021, 11, 30, 14, 0, 53, 856000), None)
Rows Values:(None, None, None)
Column Values:('Hello World', None)
Column Values:(datetime.datetime(2021, 11, 30, 14, 0, 53, 856000), None)
Column Values:(None, None)
>>>
```

Figure 4.9: *Iterating through row and columns on the Excel sheet*

In *Figure 4.10*, you can see an example where we are looping through rows of ***test_workbook*** using the ***.iter_rows()*** function and accessing the value with another for loop and the ***.value*** function. This is helpful if you need to manipulate or read specific rows in the workbook:

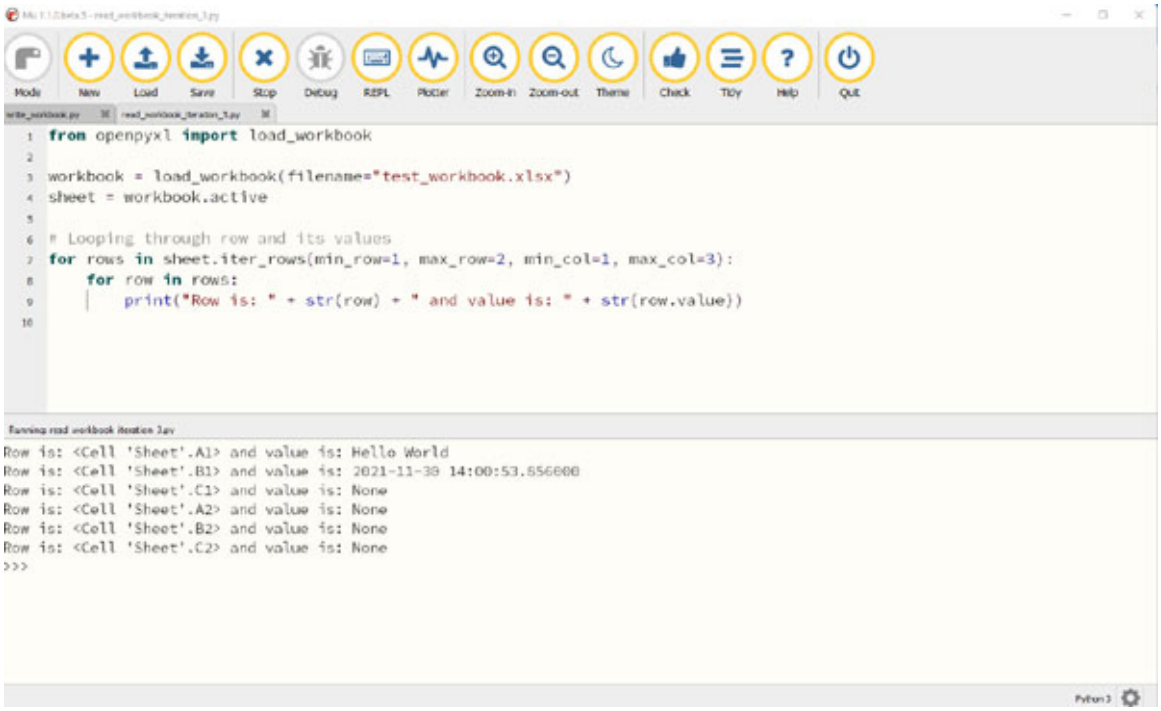


Figure 4.10: *Iterating through individual row and its value*

Updating a workbook

In this section, we will look at ways to update an existing workbook, and add or remove new data to the workbook.

To update, add, or remove data to an existing cell, use the cell assessor (like ***A1***) and set its value to the new value as required. In *Figure 4.11*, we see an example where we add new data to ***test_workbook*** and print the data using the ***iter_rows()*** function:

```python
from openpyxl import load_workbook

workbook = load_workbook(filename="test_workbook.xlsx")
sheet = workbook.active

# functions to print data from the active sheet
def print_data():
    for row in sheet.iter_rows(values_only=True):
        print(row)

sheet["A2"] = "Adding more data"
sheet["A3"] = "Adding more data"
sheet["B3"] = "Adding more data"

print_data()

# Save the workbook
workbook.save(filename="test_workbook.xlsx")
```

```
('Hello World', datetime.datetime(2021, 11, 30, 14, 0, 53, 856000))
('Adding more data', None)
('Adding more data', 'Adding more data')
>>>
```

Figure 4.11: *Updating the workbook with new data*

You can also add or remove worksheets by using the ***openpyxl*** library. To add a new worksheet, use the ***create_sheet(sheet_name)*** function, to copy the worksheet, use the ***copy_worksheet(sheet_name)*** function, and to remove the worksheet, use the ***remove(sheet_name)*** function as shown in the following figure:

```python
from openpyxl import load_workbook

workbook = load_workbook(filename="test_workbook.xlsx")

# create_sheet() helps with sheet creation
workbook.create_sheet("TestSheet2")
print(workbook.sheetnames)
# you can also add a new sheet at a perticular postion
workbook.create_sheet("TestSheet3", 0)
# To access the sheet use the sheetname
test_sheet_3 = workbook["TestSheet3"]
# To copy the sheet use copy_worksheet function
workbook.copy_worksheet(test_sheet_3)
print(workbook.sheetnames)
# to remove the sheet pass the sheet as an argument
workbook.remove(test_sheet_3)
print(workbook.sheetnames)

# Save the workbook
workbook.save(filename="test_workbook.xlsx")
```

```
['Sheet', 'TestSheet2']
['TestSheet3', 'Sheet', 'TestSheet2', 'TestSheet3 Copy']
['Sheet', 'TestSheet2', 'TestSheet3 Copy']
>>>
```

Figure 4.12: *Adding new workbooks to the Excel*

A sample of Excel-based automation

In this section, we will look at a simple example of Excel automation where you need to move some data from one Excel file to another Excel file.

In *Figure 4.13*, we can take a look at an example where we want to copy *columns 1* and *2* from the source workbook to the destination workbook. To achieve this, we can add the number of rows in the source workbook using the ***max_row*** method (number of columns in a workbook can be found similarly using the ***max_column*** method). Then, we loop through all the rows and required columns using the ***for*** loop function, getting the value from the source workbook, storing it in the variable, and then storing it in the destination workbook for the same row and column number. We then save the destination file using the ***.save()*** function:

```
from openpyxl import load_workbook

# opening the source excel file
source_file ="source_workbook.xlsx"
source_workbook = load_workbook(source_file)
source_worksheet = source_workbook.worksheets[0]
# opening the destination excel file
destination_file ="destination_workbook.xlsx"
destination_workbook = load_workbook(destination_file)
destination_worksheet = destination_workbook.active
# calculate total number of rows in source excel file
max_row = source_worksheet.max_row
# copying the column 1 and 2
for row in range (1, max_row + 1):
    for column in range (1, 3):
        # reading cell value from source excel file
        c = source_worksheet.cell(row = row, column = column)
        # writing the read value to destination excel file
        destination_worksheet.cell(row = row, column = column).value = c.value
# saving the destination excel file
destination_workbook.save(destination_file)
```

Figure 4.13: *Copying columns 1 and 2 to another workbook*

The preceding automation would work when the source and destination workbook exist and *Figure 4.14* shows the data of the source workbook:

E9

	A	B	C	D	E	F	G
1	NAME	VALUE	NET CHAN	% CHANGI	1 MONTH	1 YEAR	TIME (EST)
2	INDU:IND	35,738.71	511.68	1.45%	-1.62%	18.85%	3:19 PM
3	DOW JONES INDUS. AVG						
4	SPX:IND	4,688.62	96.95	2.11%	-0.19%	27.00%	3:04 PM
5	S&P 500 INDEX						
6	CCMP:IND	15,668.30	443.15	2.91%	-1.90%	25.15%	3:19 PM
7	NASDAQ COMPOSITE						
8	NYA:IND	16,886.86	294.89	1.78%	-2.06%	17.60%	3:04 PM
9	NYSE COMPOSITE INDEX						
10	SPTSX:IND	21,194.56	333.46	1.60%	-1.22%	20.60%	2:59 PM
11	S&P/TSX COMPOSITE INDEX						
12							
13							
14							
15							
16							

Figure 4.14: *Source workbook with sample data*

Figure 4.15 shows the data of the destination workbook with *column 1* and *2* copied from the source workbook when we run the copying data automation:

E12

	A	B	C	D	E	F
1	NAME	VALUE				
2	INDU:IND	35738.71				
3	DOW JONES INDUS. AVG					
4	SPX:IND	4688.62				
5	S&P 500 INDEX					
6	CCMP:IND	15668.3				
7	NASDAQ COMPOSITE					
8	NYA:IND	16886.86				
9	NYSE COMPOSITE INDEX					
10	SPTSX:IND	21194.56				
11	S&P/TSX COMPOSITE INDEX					
12						
13						

Figure 4.15: *Destination workbook with column 1 and 2 copied from source workbook*

CSV file automations

Python also has libraries and functions to manipulate CSV files. A ***CSV*** file is a ***Comma Separated Values*** file that contains a list of data where different elements are separated by a comma. These files are often used for exchanging data between different applications.

Python has an inbuilt CSV module which can be imported using the ***import csv*** command. This module provides functions to read, write, and update CSV files. As shown in *Figure 4.16*, we can use the CSV module to read the ***test_file*** CSV file using the ***reader()*** function, and update the file with the new data using the ***writer()*** function. Note, Python also has file I/O functions and we can open the file using the ***open()*** function as shown in the following figure. The argument ***a+*** in the ***open()*** function used in *line 10* is used to denote that we want to open the file to append the new data in it:

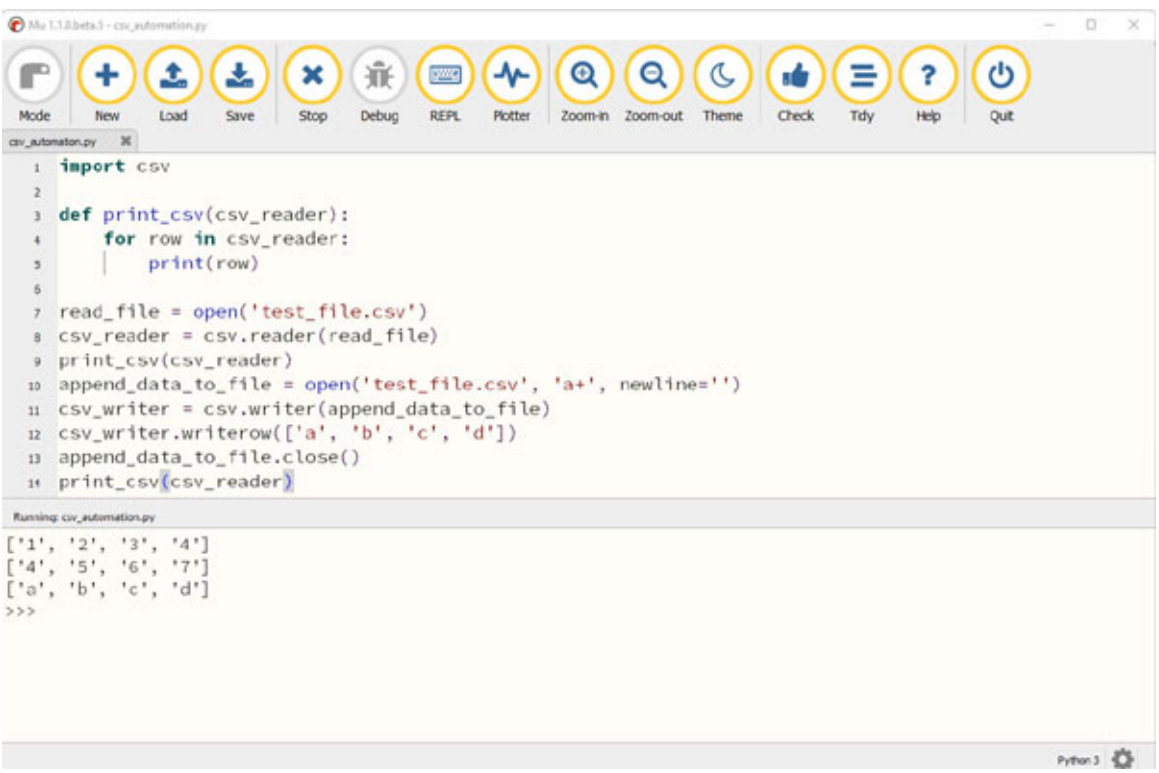


Figure 4.16: *Using CSV reader and writer to manipulate CSV files*

Figure 4.17 shows the CSV file before the ***csv_automation*** program was executed:

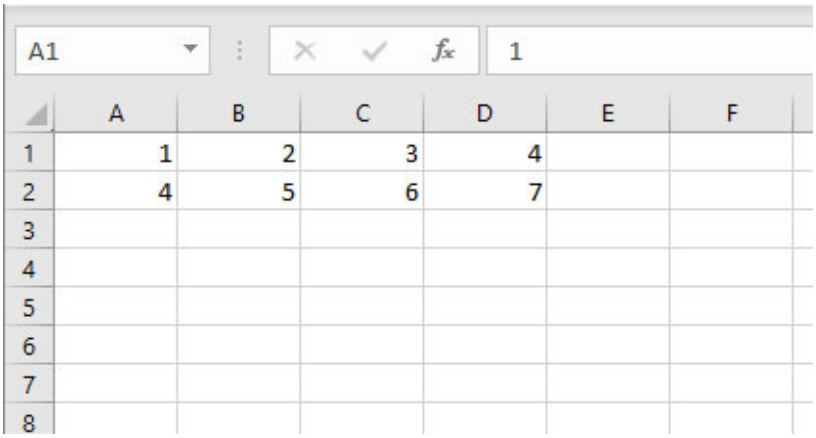


Figure 4.17: *CSV file before update*

Figure 4.18 shows the CSV file after the ***csv_automation*** program was executed adding new data on *row number 3*:

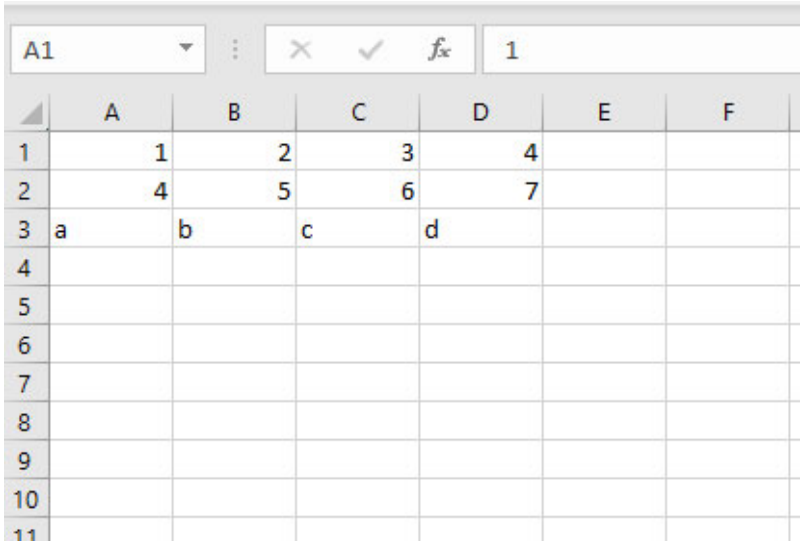


Figure 4.18: *Openpyxl being installed in Mu*

You can also create new CSV files, manipulate data inside a CSV file, and convert a CSV file to JSON format or Python object using the CSV library.

Conclusion

In this chapter, we looked at basic Excel methods to manipulate and automate Excel-based tasks. We looked at different ways to use the ***openpyxl*** library to read, write, and update Excel files. We also went through the CSV module in Python to read, write, and update CSV files.

In the next chapter, we will take a look at the various techniques to implement automation with a variety of online websites. We will discuss the Python modules to help with website-based dataset automations and various examples of automations that can be performed for different types of websites.

Further reading

There are a lot of online resources to help you learn more about Excel and CSV automation with Python. The following table lists some of the best resources to further improve your learning on Excel and CSV libraries in Python:

Resource Name	Link
`openpyxl` - A Python library to read/write Excel files	https://openpyxl.readthedocs.io/en/stable/
A guide to Excel spreadsheets in Python with `openpyxl`	https://realpython.com/openpyxl-excel-spreadsheets-python/
CSV file reading and writing	https://docs.python.org/3/library/csv.html
Excel automation with OPENPYXL in Python	https://www.topcoder.com/thrive/articles/excel-automation-with-openpyxl-in-python

Table 4.1: *Resources on CSV and Excel libraries in Python*

Questions

1. What is the most popular package in Python for Excel automation?
2. How can you create Excel documents in Python?
3. How do you build an automation to transfer data between multiple Excel sheets?
4. How to you read data from Excel documents into Python data structure?

CHAPTER 5

Automating Web-Based Tasks

Introduction

In this chapter, we will go through automation for websites and web-based tasks. We will look at how to download data from websites and automate data extraction from websites by parsing HTML documents. We will also look at the **Selenium** framework to automate web actions such as mouse click and keyboard actions on different websites.

Structure

In this chapter, we will cover the following topics:

- Downloading files from the Internet
- Introduction to HTML, CSS, and JavaScript
- Extracting data from websites
- Controlling the browser with Selenium

Objectives

After studying this chapter, you will be able to automate web-based tasks such as extracting data from webpages, downloading files, and performing search. You will also get an understanding of the Python libraries for working with websites and HTML documents.

Downloading files from the Internet

Python allows you to download web pages, HTML documents, PDF documents, videos, and other file types from the Internet. We will use `requests` which is a Python library that allows you to perform HTTP requests. One of its applications is to download a file from the web using the file URL.

To install requests, use the `mu` package manager, type `requests`, and click on `OK`, as shown in the following figure:

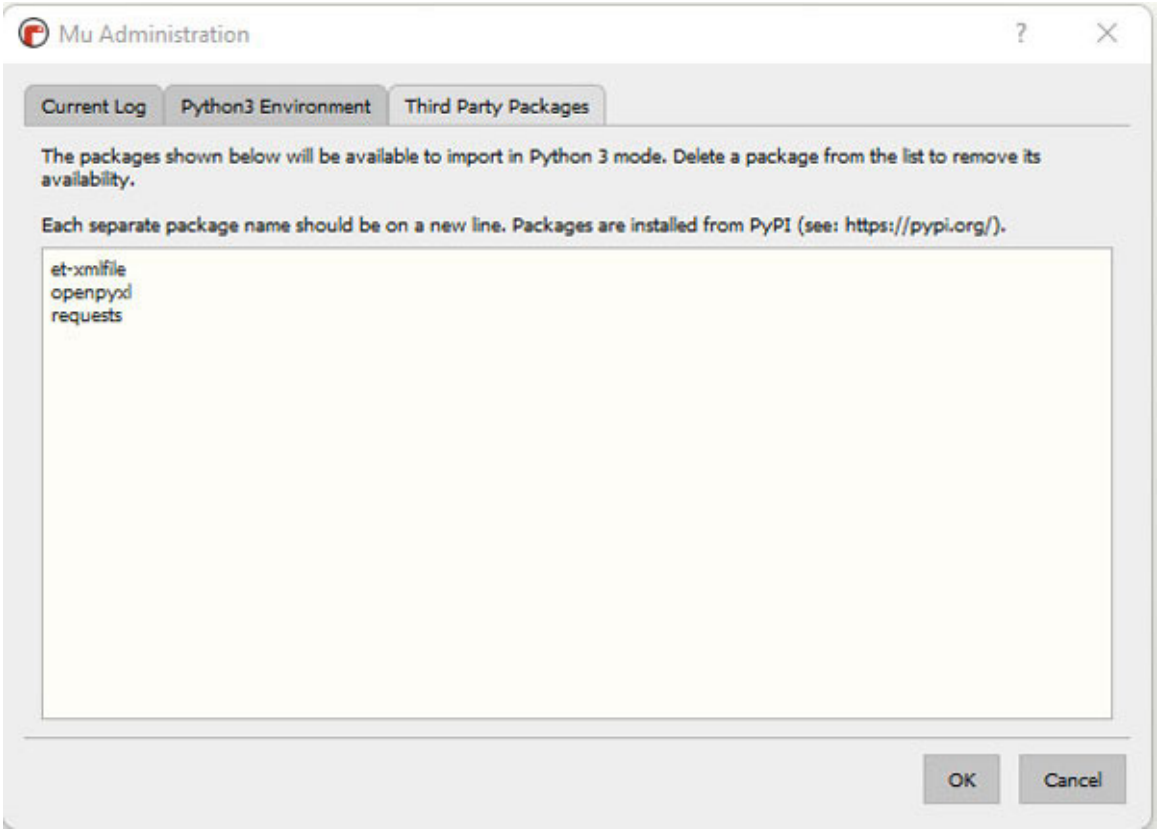


Figure 5.1: *Mu package manager*

Once the library is installed, you can import it using the **`import`** statement. The **`Requests`** library allows you to send HTTP requests, and there's no need to manually add query strings to your URLs, or to form-encode your POST data.

HTTP defines the methods to indicate the action that needs to be performed on the web service. The HTTP methods available with the **`Requests`** library are as follows:

- **`GET`**: This allows you to retrieve data from the given web link.
- **`HEAD`**: This is similar to the **`GET`** request but it does not include a response body.
- **`POST`**: This submits data to the specified web link, often causing something to happen in the server.
- **`PUT`**: This replaces the current representations on the server with uploaded data.
- **`DELETE`**: This deletes the specified data.
- **`CONNECT`**: This establishes a tunnel to the server identified by the web link.

- **`OPTIONS`**: This describes the communication options for the target web link.
- **`TRACE`**: This performs a message loop-back test.
- **`PATCH`**: This applies partial modifications to the data.

To download the files from the Internet, we will use the **`HTTP GET`** method. As shown in *Figure 5.2*, we can use the **`requests`** library with the syntax **`requests.get(FILE_LINK)`** to download the file from the Internet.

We are also using the Python file write in the following figure which allows us to create a new file or add data to an existing file. Python has the **`open()`** function which is the key function for working with files. The **`open()`** function takes two parameters as arguments which are file location and mode. There are four different modes for opening a file using the **`open`** function:

- **`r`**: **Read** – opens a file for reading.
- **`a`**: **Append** – opens a file for adding more data and creates a new file if it does not exist.
- **`w`**: **Write** - opens a file for writing and creates a new file if it does not exist.
- **`x`**: **Create** - creates a new file.

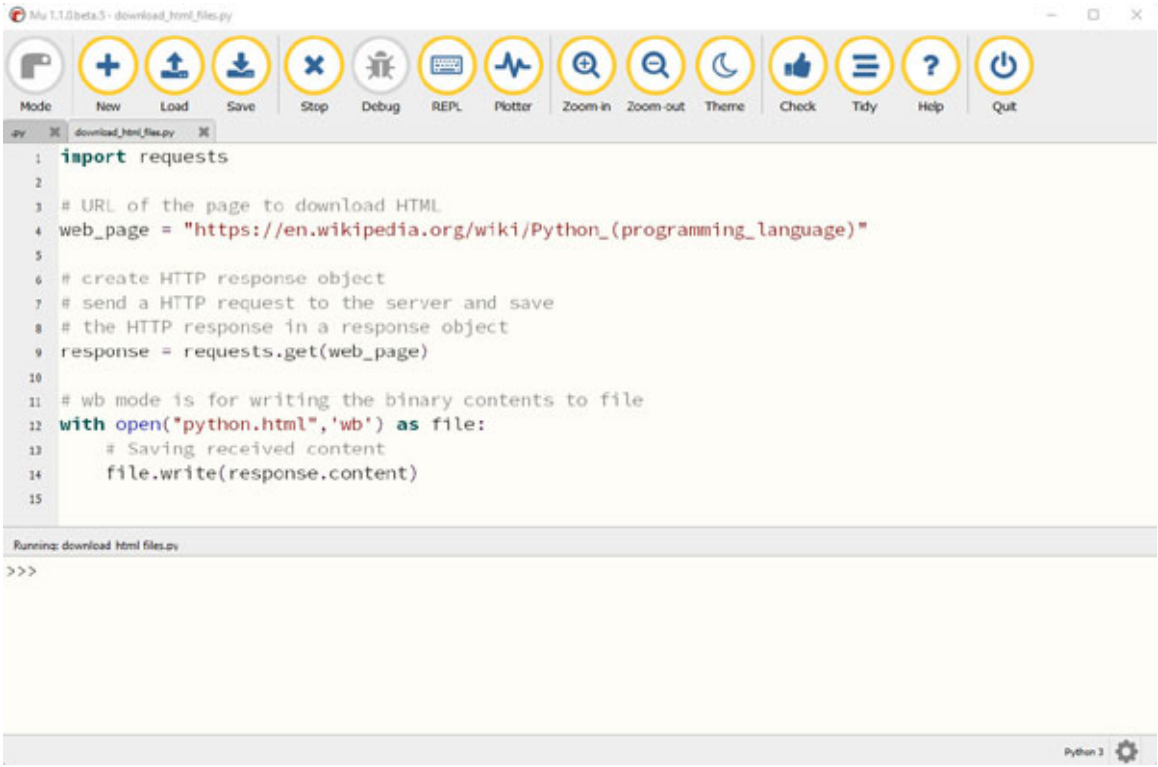


Figure 5.2: *Downloading simple HTML web page*

To download multiple files from the Internet, we can add multiple URLs in a Python list, and use the `for` loop to loop through the links, and download the required files as shown in *Figure 5.3*:

```
# importing the requests library
import requests

# Multiple URLs
multiple_urls = {
    "python_logo.png": "https://www.python.org/static/community_logos/python-logo-master-v3-TM.png",
    "python.html": "https://www.python.org/"
}
for file_name, url in multiple_urls.items():
    response = requests.get(url)
    with open(file_name, "wb") as file:
        file.write(response.content)
```

Figure 5.3: *Downloading multiple files from the internet*

Unless a save location is specified, the files are downloaded in the folder where the code file is present as shown in *Figure 5.3*:

Name	Date modified	Type	Size
download_html_files.py	12/18/2021 11:52 AM	PY File	1 KB
download_multiple_files.py	12/18/2021 12:01 PM	PY File	1 KB
python.html	12/18/2021 12:01 PM	Chrome HTML Docume...	49 KB
python_logo.png	12/18/2021 12:01 PM	PNG File	82 KB

Figure 5.4: *Files downloaded in the code folder*

To download large files from the Internet, we can set the **`stream`** parameter to **`True`** in the **`requests`** function. This will download the response headers only and the connection will remain *open*. This avoids reading the content all at once into memory for large responses. A fixed chunk is loaded into memory each time while **`r.iter_content`** is iterated.

As shown in *Figure 5.5*, we loop through the response and write the large PDF document in the required file:

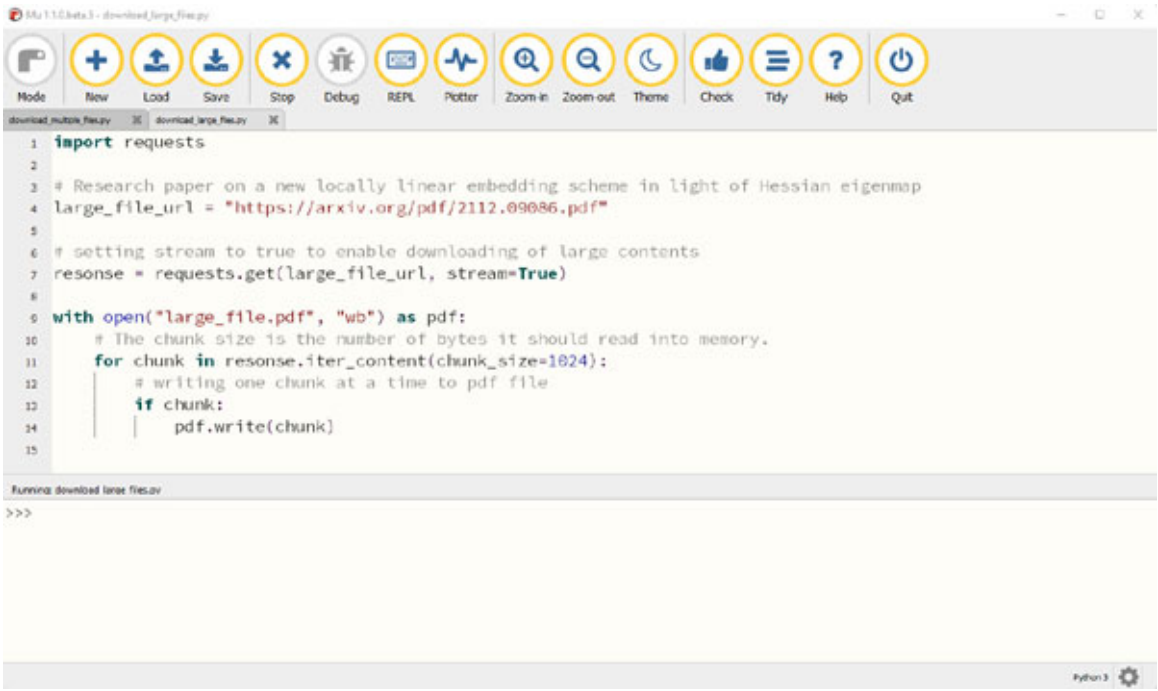


Figure 5.5: *Downloading large files*

In the next section, we will go through the basic introduction of HTML, CSS, and JavaScript which are used for creating web pages and websites available on the Internet. A basic knowledge of HTML, CSS, and JavaScript is essential to be able to successfully automate the web data extraction tasks.

Introduction to HTML, CSS, and JavaScript

In this section, we will go through the building blocks and components of a web page. When we visit a web page, our web browser makes a **`GET`** request to a web server. The server then sends back files that tell our browser how to render the page for us. These files typically include:

- **HTML**: The main content of the page to be displayed in the browser.
- **CSS**: This is used to add styling to make the web page look nicer.

- **JavaScript**: JavaScript files add interactivity and additional functionality to web pages.
- **Images**: Image files such as JPG and PNG allow web pages to show pictures.
- **Other files formats**: These can be videos, documents, audio files, or any other file types.

After our browser receives all the files, it renders the page and displays it.

HTML

When we perform web scraping, we are mainly interested in the main content of the web page which is an HTML document. **HTML** stands for **hypertext markup language** and is the language used for building websites. HTML code is based on tags that provide instructions for formatting and displaying the document. A tag starts with the *less than* sign: **<** and ends with the *greater than* sign **>**.

For example, to make the word **`Hello`** *bold*, you can use the opening **bold** tag **`<b>`** and then the closing bold tag **`</b>`**, like this:

```
<b>Hello</b>
```

HTML documents can be created using the **`<html>`** and **`</html>`** tags.

There is a **`head`** tag which contains data about the title of the page and other top-level information, and a **`body`** tag which contains the main content of the page. For web scraping, we will mostly be interested in content within the **`body`** tag of the HTML page.

The commonly used HTML tags are:

- **`<!--...-->`**: Defines a comment.
- **`<!DOCTYPE>`**: Defines the document type.
- **`<a>`**: Defines a hyperlink.
- **`<audio>`**: Defines embedded sound content.
- **`<b>`**: Defines bold text.
- **`<body>`**: Defines the document's body.
- **`<br>`**: Defines a single line break.
- **`<button>`**: Defines a button.
- **`<caption>`**: Defines a table caption.

- **`<dialog>`**: Defines a dialog box or window.
- **`<div>`**: Defines a section in a document.
- **`<footer>`**: Defines a footer for a document or section.
- **`<form>`** : Defines an HTML form for user input.
- **`<h1>`** to **`<h6>`**: Defines HTML headings of different sizes with **`h1`** being the largest.
- **`<head>`** : Contains metadata/information for the document.
- **`<html>`**: Defines the root of an HTML document.
- **`<img>`**: Defines an image.
- **`<input>`**: Defines an input control.
- **`<label>`**: Defines a label for an **`<input>`** element.
- **`<li>`**: Defines a list item.
- **`<ol>`**: Defines an ordered list.
- **`<option>`**: Defines an option in a drop-down list.
- **`<p>`**: Defines a paragraph.
- **`<pre>`**: Defines preformatted text.
- **`<select>`**: Defines a drop-down list.
- **`<span>`**: Defines a section in a document.
- **`<style>`**: Defines style information for a document.
- **`<table>`**: Defines a table.
- **`<tbody>`**: Groups the body content in a table.
- **`<td>`**: Defines a cell in a table.
- **`<th>`**: Defines a header cell in a table.
- **`<title>`**: Defines a title for the document.
- **`<tr>`**: Defines a row in a table.
- **`<ul>`**: Defines an unordered list.
- **`<video>`**: Defines embedded video content.

The HTML document has an **`id` attribute** which is used to specify a unique ID for an HTML element. The **`id`** property is particularly useful for automating web-based tasks. The value of the **`id`** is unique within the HTML document. In the HTML document, **`id`** is declared for a particular tag as shown in the following example:

```
<h1 id="myId">My Id</h1>
```

The HTML document also has a **`class` attribute** that can be used to identify elements. Multiple elements can have the same class in the HTML document. In the HTML document, **`class`** is declared for a particular tag as shown in the following example:

```
<div class="myClass"> </div>
```

A simple HTML code snippet is shown as follows which will print **`Hello World`** when it is displayed on the browser:

```
1. <html>
2. <head>
3. </head>
4. <body>
5.    <h1>Hello World<h1>
6. </body>
7. </html>
```

CSS

CSS is used to style a web page and it stands for **Cascading Style Sheets**. It describes how HTML elements are to be displayed on screen, paper, or in other media. It can be reused across different web pages and are generally stored in CSS files. CSS documents are generally not useful for web automation purposes as they just define the style of the webpage and not its content.

The following example is a sample CSS file where all **`<p>`** elements are center-aligned with black text color:

```
1. p {
2.   color: black;
3.   text-align: center;
4. }
```

JavaScript

JavaScript is a programming language similar to Python and is the main programming language used for designing web pages. JavaScript is used to change the HTML content and manipulate HTML documents.

In HTML documents, the JavaScript code is added within the **`script`** tag shown as follows where **`main.js`** is the name of the JavaScript file containing the JavaScript code:

```
<script src="main.js"></script>
```

The following is an example of simple JavaScript code that can be used to change the heading of the HTML document:

```
1. const docHeading = document.querySelector('h1');
2. docHeading.textContent = 'New Heading';
```

When we add this code to the HTML document, when the code is executed it will change the heading of the HTML document to the **`New Heading`** value.

An in-depth knowledge of JavaScript is not required to perform web-based automation but the following **w3schools** tutorial (**https://www.w3schools.com/js/**) is a great place to start to learn more about the language.

In the next section, we will use the basic knowledge of HTML documents to extract data from web pages and automate web-based tasks.

Extracting data from websites

Extracting data from websites is called **web scraping** and it involves getting the HTML page from the web and extracting required data from the HTML document. In Python, we will use the **Beautiful Soup** library which makes it very easy to extract data from HTML documents. **Beautiful Soup** allows us to write custom code that filters through the specific elements that we specified and extracts the required content as instructed.

To install **Beautiful Soup**, use the **`mu`** package manager, type **`beautifulsoup4`**, and click on **`OK`**, as shown in the following figure:

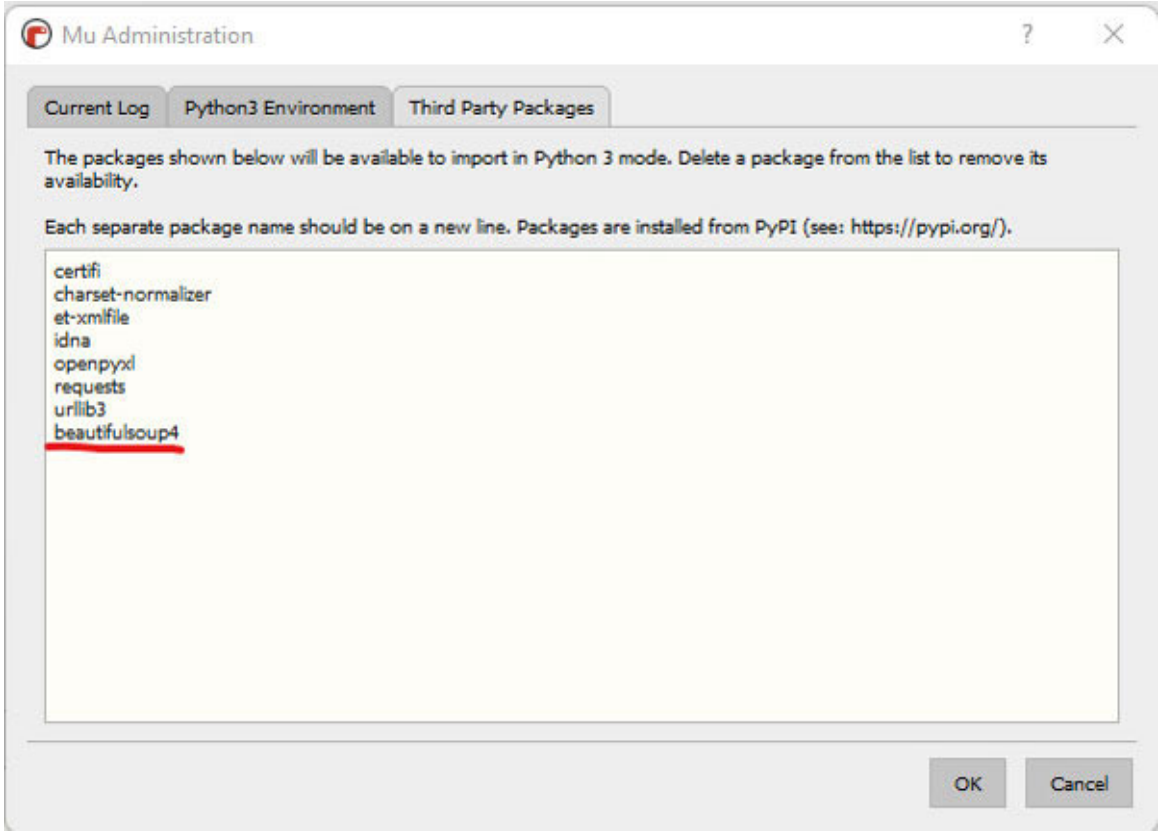


Figure 5.6: *Mu package manager*

Once the Beautiful Soup library is installed, you can import it with the statement **`from bs4 import BeautifulSoup`** where **`bs4`** stands for **`beautifulsoup4`**, as shown in *Figure 5.7*:

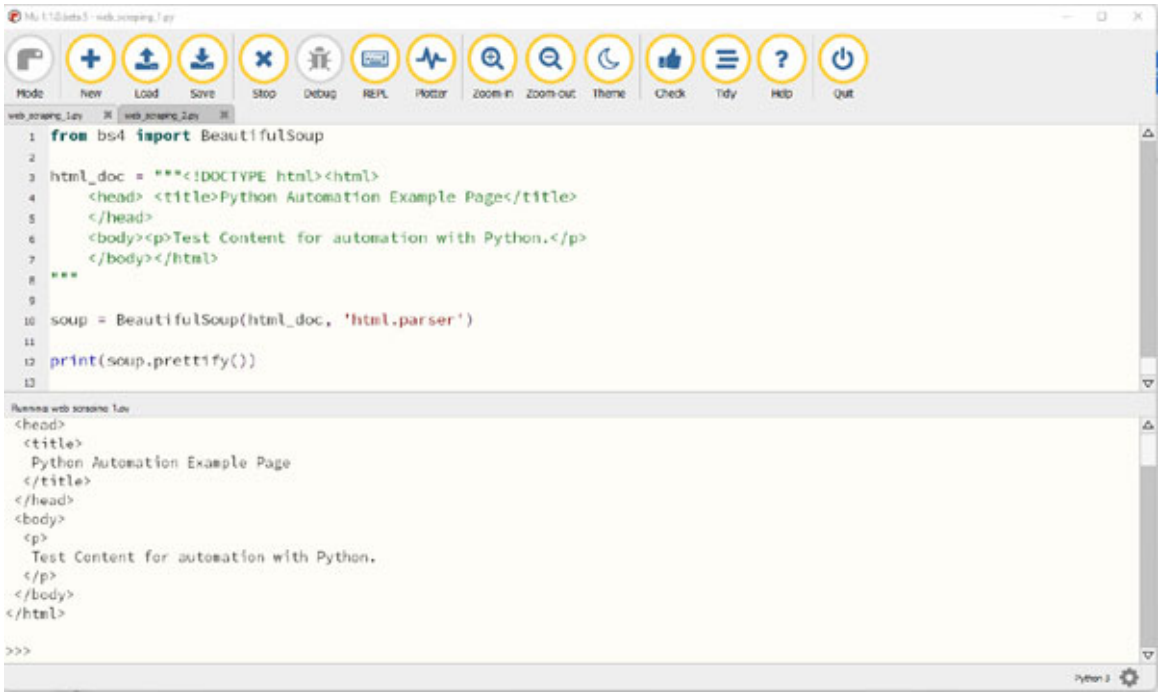


Figure 5.7: *Using BeautifulSoup library*

Beautiful Soup transforms a complex HTML document into a tree of Python objects. There are four main types of objects that we will use for extracting data from web pages: **Tag**, **NavigableString**, **BeautifulSoup**, and **Comment**. The main properties and objects we will use for data extraction are:

- **Tag**: The **`tag`** object corresponds to the HTML tag in the original document. For example:

```
soup = BeautifulSoup('<b class="bold">bold text</b>',
'html.parser')
tag = soup.b
type(tag)
# returns <class 'bs4.element.Tag'> as output
```

- **Name**: Every HTML tag has a name which can be accessed using **`.name`**. For example:

```
tag.name
# returns 'b' as output
```

- **Attributes**: A tag can have any number of attributes. The **`<b id="bold">`** tag has an attribute **`id`** whose value is **`bold`**. You can access a **`tag`**'s attributes by treating the **`tag`** like a dictionary. For example:

```
tag = BeautifulSoup('<b id="bold">bold text</b>',
'html.parser')
```

```
tag['id']
# returns bold as output
```

You can access that dictionary containing all attributes using the **`.attrs`** function.

- **NavigableString**: A string that contains the text within a tag. For example:

```
tag = BeautifulSoup('<b id="bold">bold text</b>',
'html.parser')
bold = tag .b
bold.string
# returns 'bold text' string as output
```

- **BeautifulSoup**: The **`BeautifulSoup`** object represents the parsed HTML document. It is similar to the **`tag`** object, and supports the preceding methods to navigate and search the document.

As shown in *Figure 5.8*, we are converting the HTML document using the **`BeautifulSoup`** function, and then access its property directly by using the **`tag`** object properties:

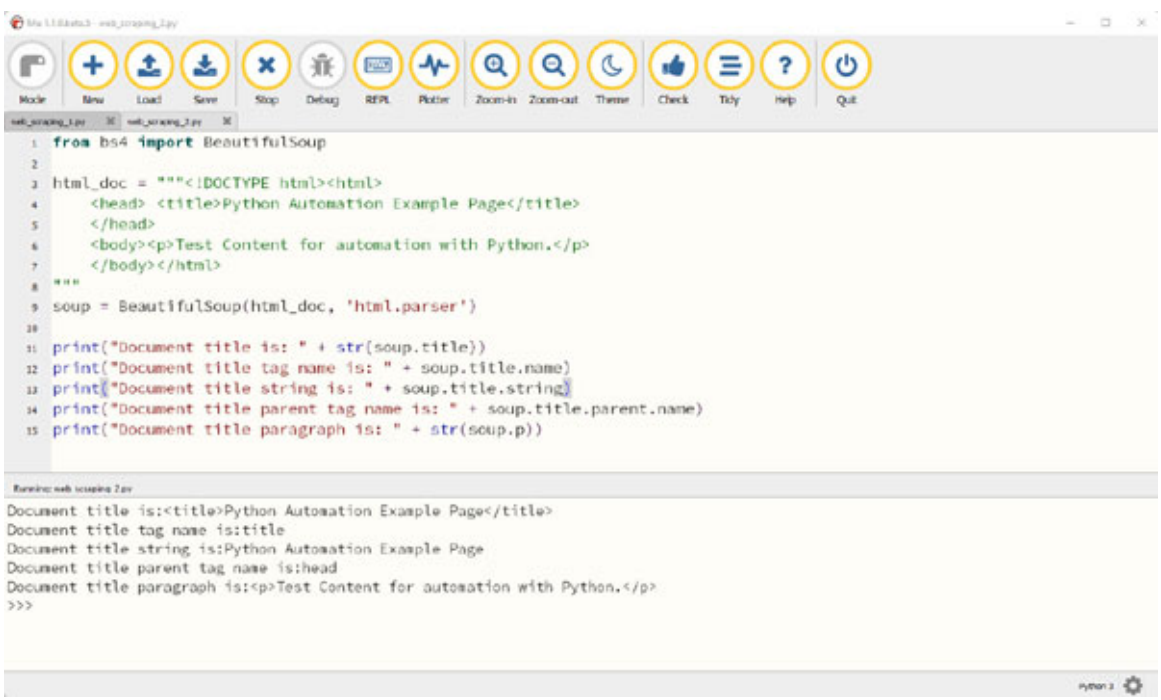


Figure 5.8: *Extracting HTML elements*

We can also download a webpage using the **`requests`** library and extract data by converting the downloaded document into a **`BeautifulSoup`** object as

shown in *Figure 5.9*:

```
from bs4 import BeautifulSoup
import requests
page = requests.get("https://www.python.org/")
soup = BeautifulSoup(page.content, 'html.parser')
print(soup.prettify())
```

```
<!DOCTYPE html>
<!--[if lt IE 7]>   <html class="no-js ie6 lt-ie7 lt-ie8 lt-ie9">   <![endif]-->
<!--[if IE 7]>      <html class="no-js ie7 lt-ie8 lt-ie9">          <![endif]-->
<!--[if IE 8]>      <html class="no-js ie8 lt-ie9">                 <![endif]-->
<!--[if gt IE 8]><!-->
<html class="no-js" dir="ltr" lang="en">
 <!--<![endif]-->
 <head>
  <meta charset="utf-8"/>
  <meta content="IE=edge" http-equiv="X-UA-Compatible"/>
  <link href="//ajax.googleapis.com/ajax/libs/jquery/1.8.2/jquery.min.js" rel="prefetch"/>
  <link href="//ajax.googleapis.com/ajax/libs/jqueryui/1.12.1/jquery-ui.min.js" rel="prefetch"/>
  <meta content="Python.org" name="application-name"/>
  <meta content="The official home of the Python Programming Language" name="msapplication-tooltip"/>
  <meta content="Python.org" name="apple-mobile-web-app-title"/>
  <meta content="yes" name="apple-mobile-web-app-capable"/>
  <meta content="black" name="apple-mobile-web-app-status-bar-style"/>
  <meta content="width=device-width, initial-scale=1.0" name="viewport"/>
  <meta content="True" name="HandheldFriendly"/>
  <meta content="telephone=no" name="format-detection"/>
```

Figure 5.9: *Downloading and parsing online documents*

To extract elements of a particular type, we can use the **`find_all()`** function to get the elements for that type. We can use this function to extract all the external links from a particular HTML page as shown in *Figure 5.10*:

```
from bs4 import BeautifulSoup
import requests

page = requests.get("https://www.python.org/")
soup = BeautifulSoup(page.content, 'html.parser')

# Get all the external links contained in the HTML page
for link in soup.find_all('a'):
    print(link.get('href'))
```

```
https://devguide.python.org/
https://bugs.python.org/
https://mail.python.org/mailman/listinfo/python-dev
/dev/core-mentorship/
/dev/security/
#python-network
/about/help/
/community/diversity/
https://github.com/python/pythondotorg/issues
https://status.python.org/
/psf-landing/
/about/legal/
/privacy/
/psf/sponsorship/sponsors/#heroku
>>>
```

Figure 5.10: *Extracting web links from website*

To extract elements from the tag ID, we can use the **`find()`** function with the required **`id`** value as shown in *Figure 5.11*:

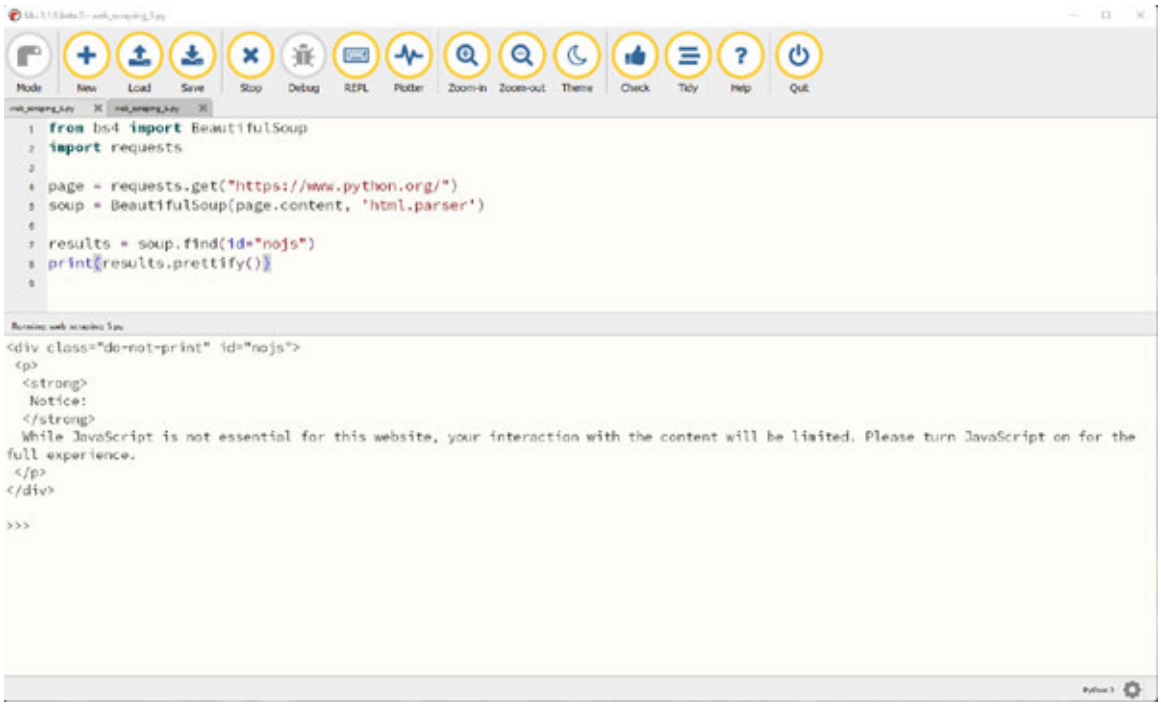


Figure 5.11: *Extracting data via HTML tag ID*

To extract elements from the class name, we can use the **`find_all()`** function with the required **`class`** and **`tag`** values as shown in *Figure 5.12*:

```
from bs4 import BeautifulSoup
import requests

page = requests.get("https://www.python.org/")
soup = BeautifulSoup(page.content, 'html.parser')

results = soup.find_all("div", {"class": "do-not-print"})
print(results)
```

```
[<div class="do-not-print" id="nojs">
<p><strong>Notice:</strong> While JavaScript is not essential for this website, your interaction with the content will be limited. Please
turn JavaScript on for the full experience. </p>
</div>, <div class="top-bar do-not-print" id="top">
<nav class="meta-navigation container" role="navigation">
<div class="skip-link screen-reader-text">
<a href="#content" title="Skip to content">Skip to content</a>
</div>
<a aria-hidden="true" class="jump-link" href="#python-network" id="close-python-network">
<span aria-hidden="true" class="icon-arrow-down"><span>▾</span></span> Close
                </a>
<ul class="menu" role="tree">
<li class="python-meta current_item selectedcurrent_branch selected">
<a class="current_item selectedcurrent_branch selected" href="/" title="The Python Programming Language">Python</a>
</li>
<li class="psf-meta">
<a href="/psf-landing/" title="The Python Software Foundation">PSF</a>
</li>
<li class="docs-meta">
```

Figure 5.12: *Extracting data via HTML class*

We can also use the **`select_one()`** function with the class name as a parameter to extract data as shown in *Figure 5.13*:

```
from bs4 import BeautifulSoup
import requests

page = requests.get("https://www.python.org/")
soup = BeautifulSoup(page.content, 'html.parser')

results = soup.select_one('.do-not-print')
print(results)
```

```
<div class="do-not-print" id="nojs">
<p><strong>Notice:</strong> While JavaScript is not essential for this website, your interaction with the content will be limited. Please
turn JavaScript on for the full experience. </p>
</div>
>>>
```

Figure 5.13: *Extracting one element data by class name*

In this section, we looked at a few examples on how we can use the **`beautifulsoup`** and **`requests`** library to extract the required data from the web pages. In the next section, we will look at the **`Selenium`** library that allows you to automate the browser mouse and keyboard actions.

Controlling the browser with Selenium

Selenium is a library that allows you to automate the browser action. It provides extensions to emulate the user interaction with browsers and allows you to write the code to automate all major web browsers.

We will look at automation on the *Chrome* browser using Selenium but the automations can be easily imported to other browsers as well. To install **`selenium`**, use the **`mu`** package manager, type **`selenium`**, and click on **`OK`** as shown in the following figure:

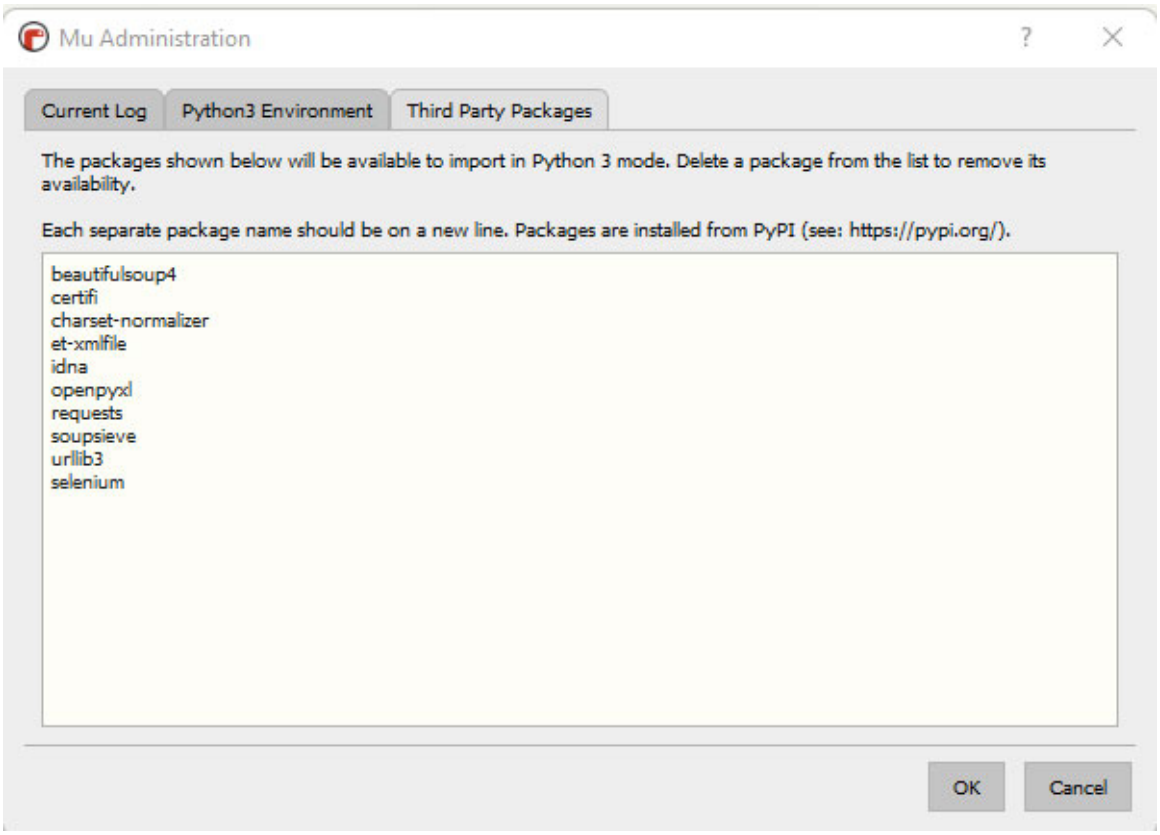


Figure 5.14: *Mu package manager*

We will also need to download the Chrome driver with the **`selenium`** package to be able to automate Chrome actions as per the following steps:

- Download the *Chrome* driver from the *chromium* website, and select the right version and operating system for your Chrome browser as required (**https://chromedriver.chromium.org/downloads**).
- Extract the downloaded folder using any ZIP extractor tool and copy the path of the location of the **`chromedriver.exe`** file.
- You can also move the file to **`c`** drive or any other path that is accessible by the system variables.

After this, we can perform the browser automations by importing the web driver from the **`selenium`** library using the **`from selenium import webdriver`** statement. We will also need to import the Chrome service using **`from selenium.webdriver.chrome.service import Service`**.

To create a **`selenium`** service, use the **`Service()`** function with the path of the **`chromedriver`** as shown in *Figure 5.15*:

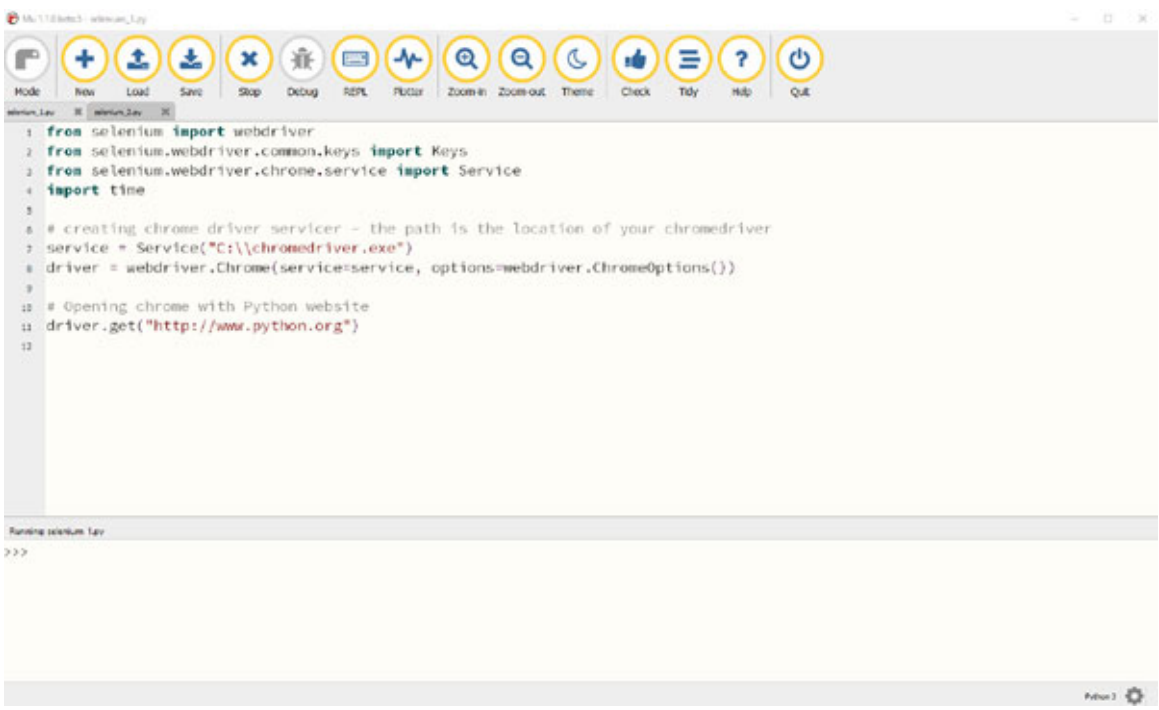


Figure 5.15: *Opening Chrome with Selenium*

The **`selenium`** has a **`get()`** function that takes the argument of the URL of the page to be opened and opens the requested page as shown in *Figures 5.15* and *5.16*:

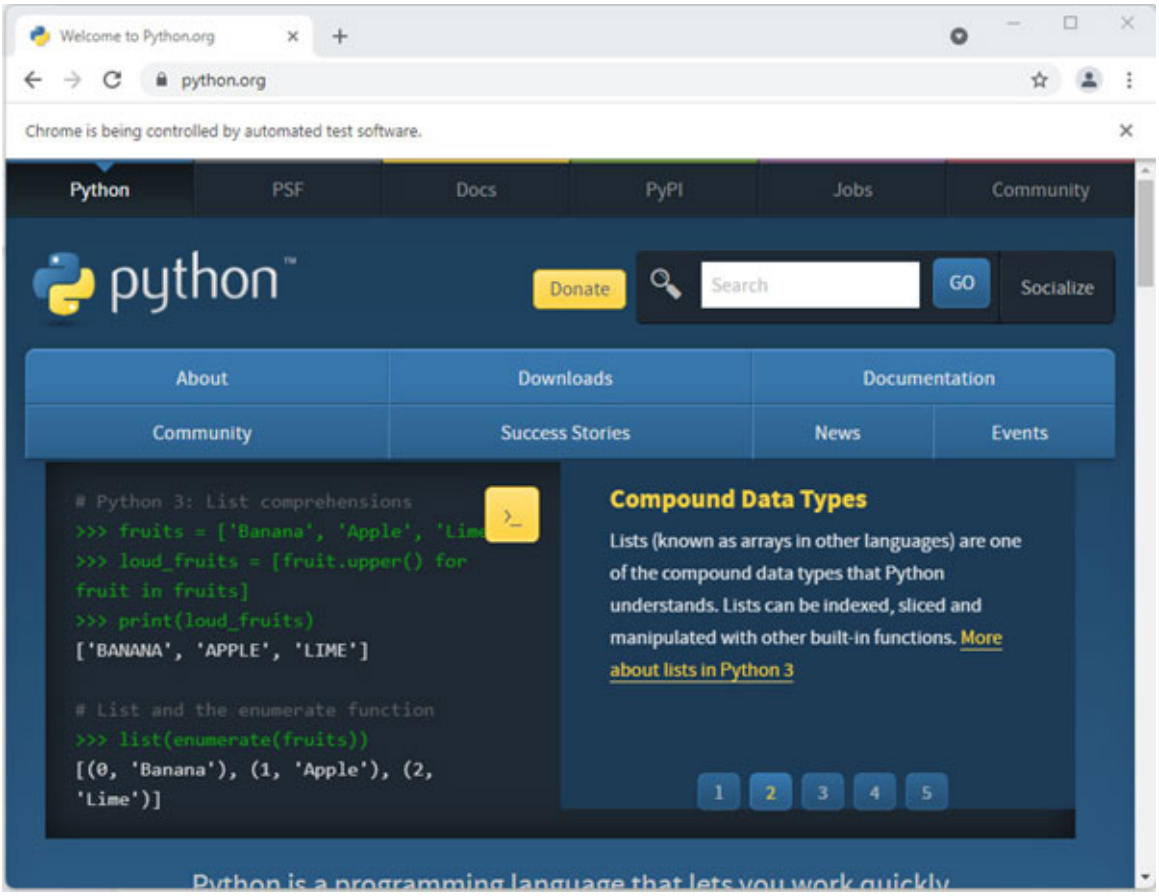


Figure 5.16: *Python home page*

There are two methods in selenium that we would be using to locate web page elements and perform mouse or keyboard actions on them. These methods are **`find_element`** and **`find_elements`**. They can be used as per the following example:

1. `from selenium.webdriver.common.by import By`
2. `driver.find_element(By.XPATH, '//button[text()="text"]')`
3. `driver.find_elements(By.XPATH, '//button')`

These are the attributes available for the **`By`** class:

- **`ID`** = id
- **`XPATH`** = xpath
- **`LINK_TEXT`** = link text
- **`PARTIAL_LINK_TEXT`** = partial link text
- **`NAME`** = name

- **`TAG_NAME`** = tag name
- **`CLASS_NAME`** = class name
- **`CSS_SELECTOR`** = CSS selector

As shown in *Figure 5.17*, we can use the **`find_element()`** function with the **`By.NAME`** parameter as **`q`** and send keys to that element in the Chrome window:

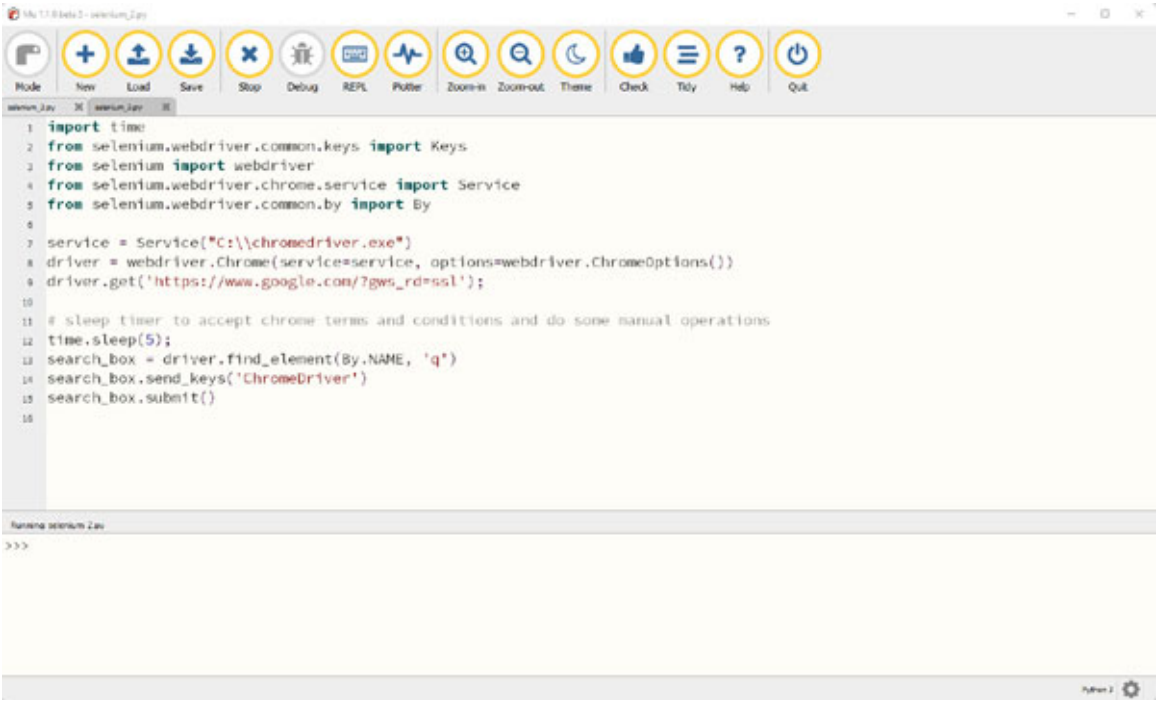


Figure 5.17: *Automating keyboard actions in Chrome*

Once the script shown in *Figure 5.17* is executed, the *Chrome* driver opens a *Google* search page and searches for **`ChromeDriver`** automatically using the **`send_keys()`** and **`submit()`** functions as shown in *Figure 5.18*:

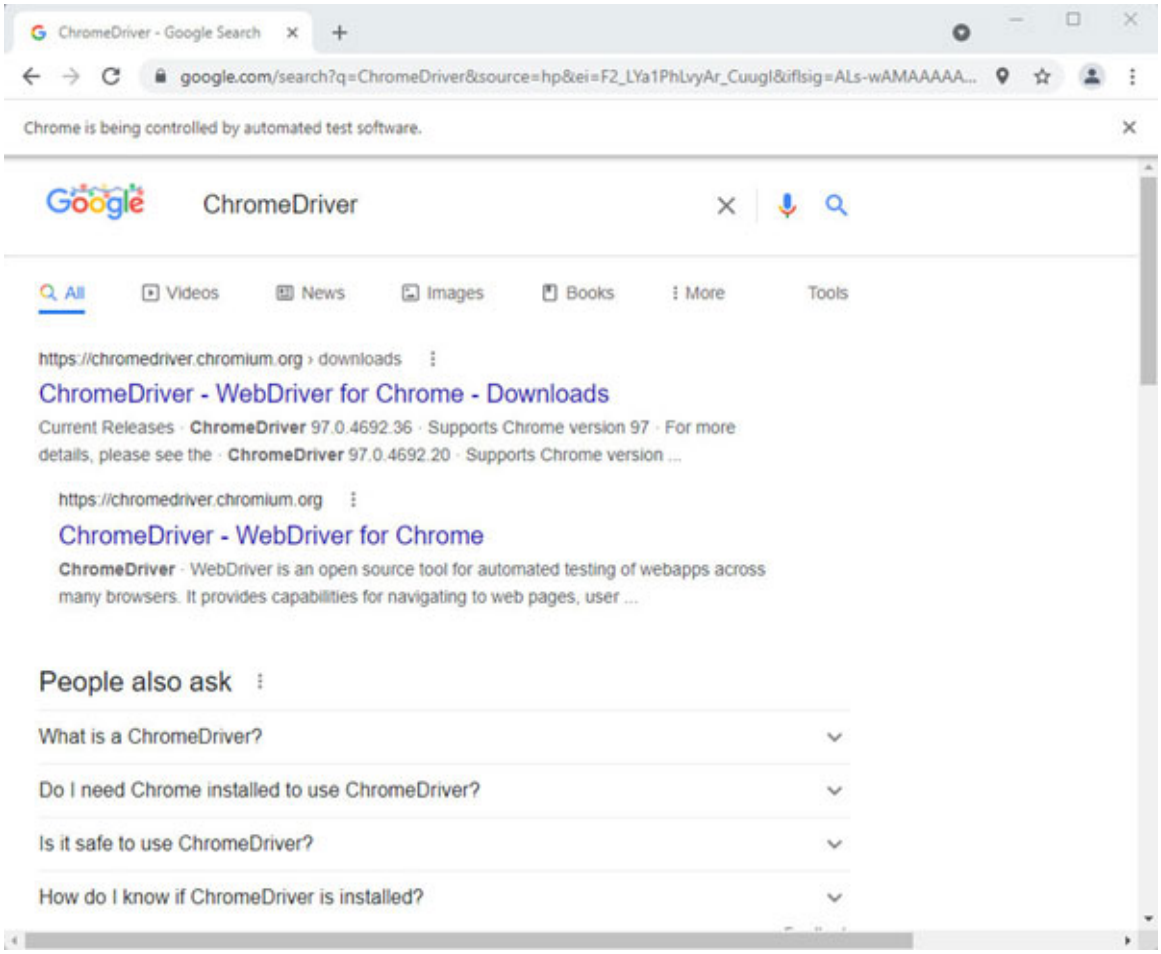
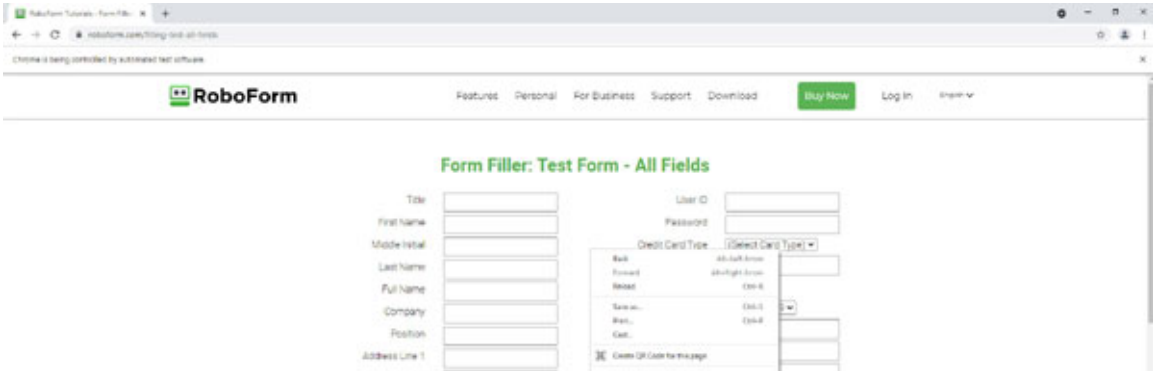


Figure 5.18: *Performing automated Chrome search*

We can also automate tasks involving filling up forms or copying data from in-house applications to forms using **Selenium**. To achieve this, you can identify elements by **XPATH**, **ID** or any other tags that are accepted by the `By` function.

To identify the HTML element's name, perform the following steps:

1. Right click on the web page that you want to automate and select the `Inspect` option as shown in *Figure 5.19*:



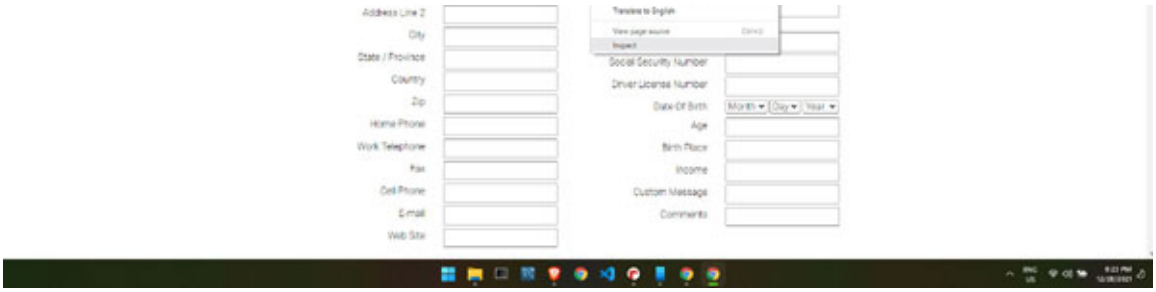


Figure 5.19: *Example form page*

2. Once you select the `Inspect`... option, you will see the elements panel that opens on the right-hand side of the browser as shown in *Figure 5.20*:

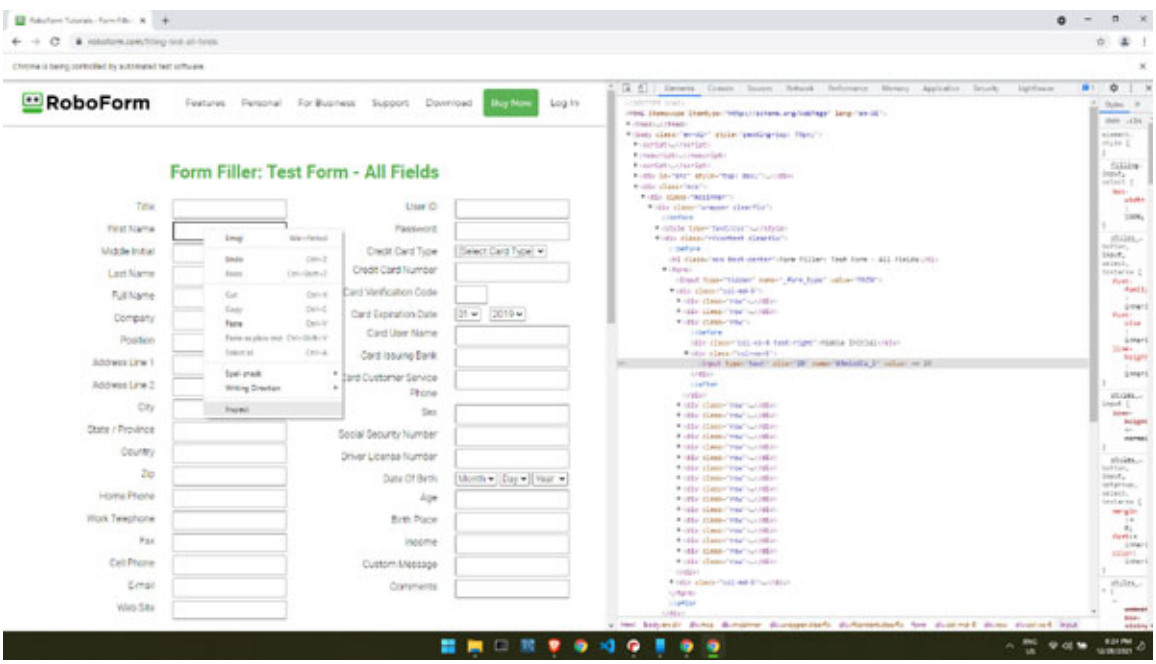


Figure 5.20: *Inspecting page data*

3. Hover over the required element to get the name for that element highlighted in the elements panel as shown in *Figure 5.21*. Take a note of this name as you would need to use it in your code to automate actions:

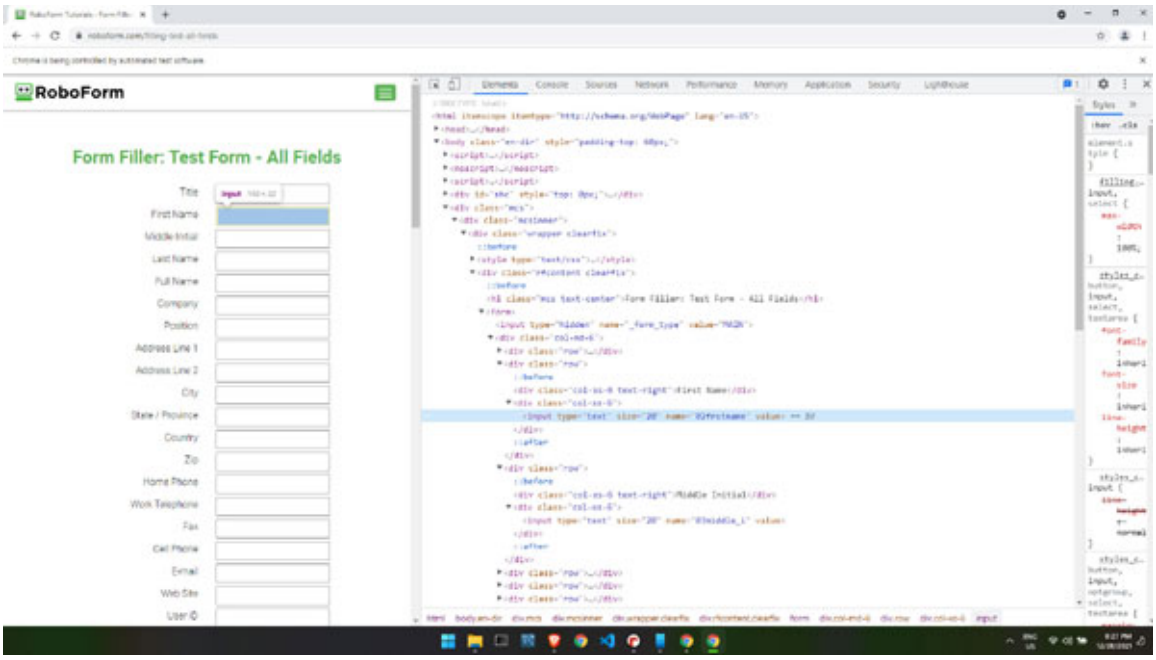


Figure 5.21: *Getting element tags*

4. Once you have identified the element names, you can get the elements by using the **`find_element()`** function and using the **`send_keys`** function to send specific data to this element as shown in *Figure 5.22*:

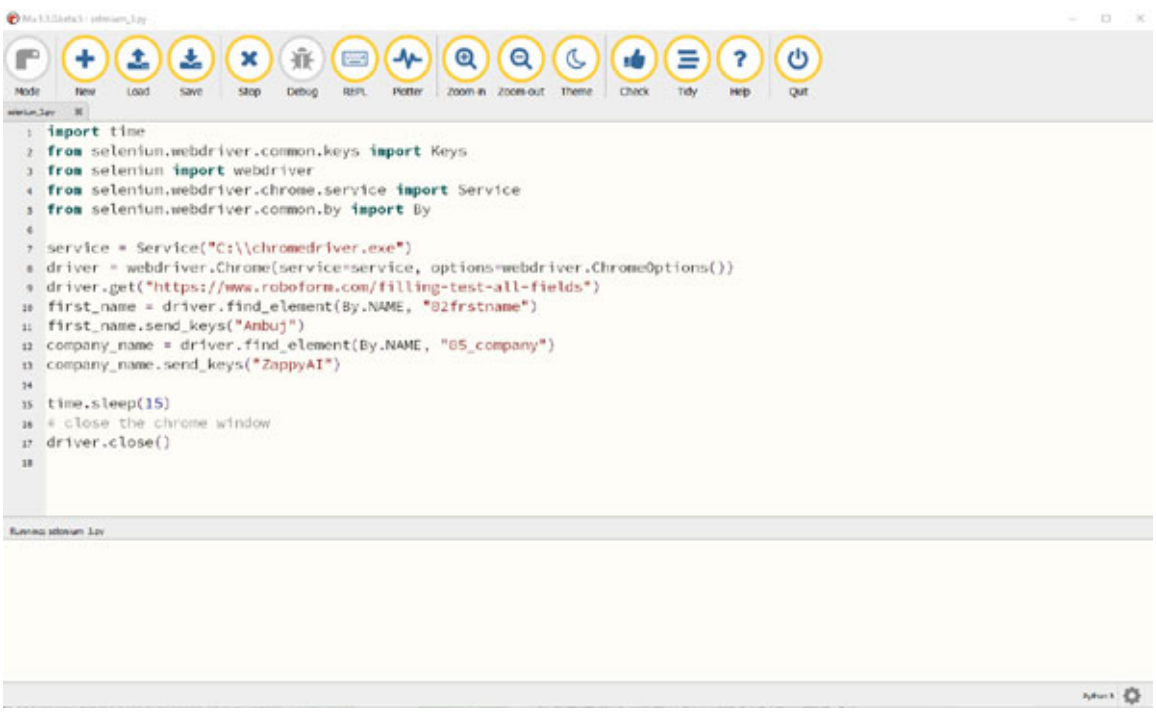


Figure 5.22: *Filling form data*

5. Once the script shown in *Figure 5.22* is executed, the Chrome driver opens a form page and fills up the form with the required data as shown in *Figure 5.23*:

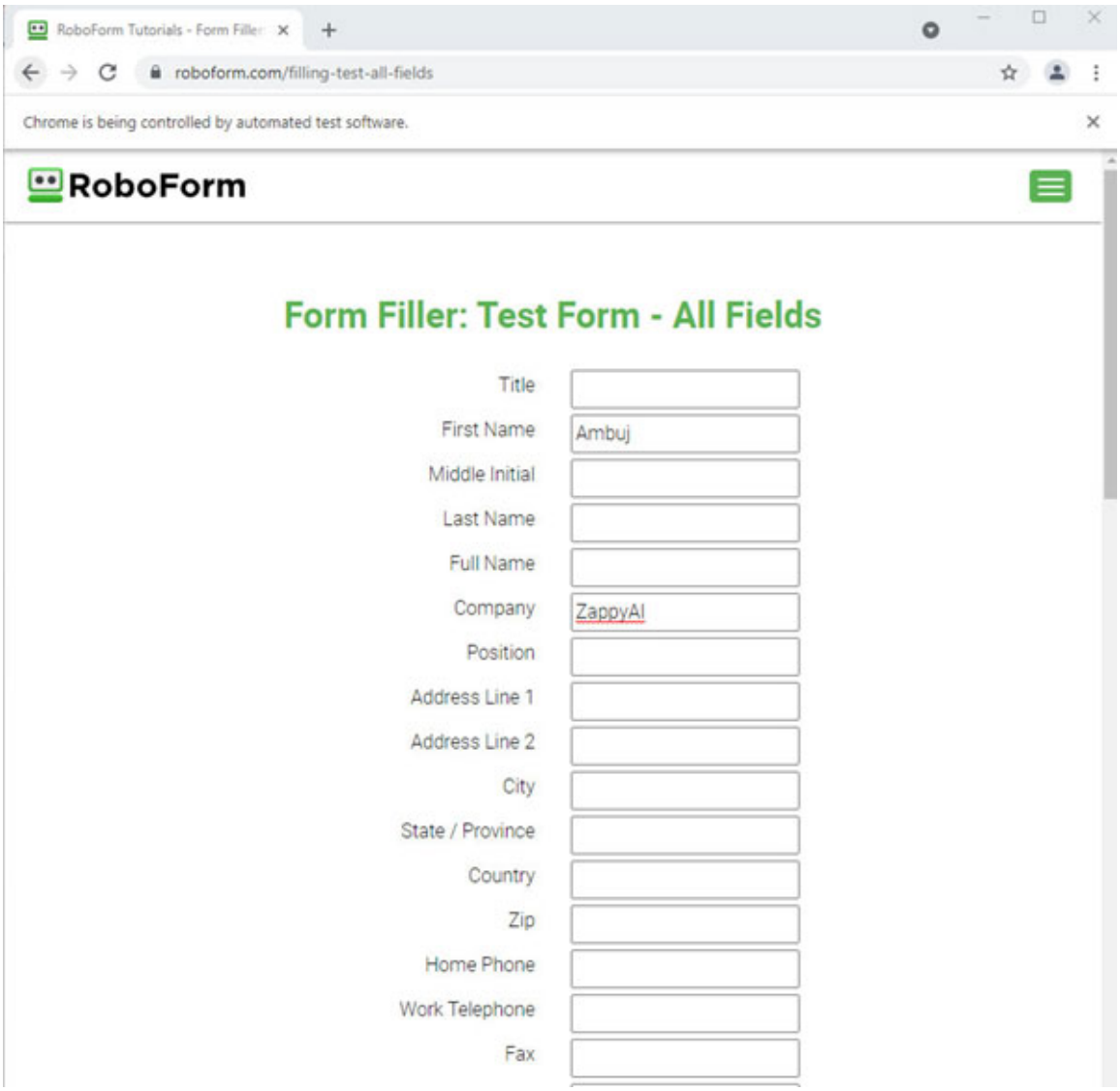


Figure 5.23: *Output after the form data is automatically filled*

6. When we send keys using **`selenium`**, it is similar to typing the keys using your keyboard. Special keys can also be sent using the **`Keys`** class imported from **`selenium.webdriver.common.keys`**. For example, to press *Enter*, you can use the **`send_keys(Keys.RETURN)`** function.
7. Finally, to close the browser window, you can use the **`driver.close()`** function which will close the browser window and end the program.

Conclusion

In this chapter, we covered a lot of content with regards to web automation in Python. We looked at ways to download files from the Internet, extract data from websites, and control browser actions with Selenium. We also went through the basics of HTML, CSS, and JavaScript to help you automate web-based tasks successfully.

In the next chapter, we will look at various files-based automation with Python. In particular, we will look at automations involving reading, writing, and creating PDF documents, Word documents, and other file types.

Further reading

There are a lot of online resources to help you learn more about web automation with Python. The following table lists some of the best resources to further improve your learning on web libraries in Python:

Resource name	Link
Requests: HTTP for humans	https://requests.readthedocs.io/en/latest/
Downloading files from web using Python	https://www.geeksforgeeks.org/downloading-files-web-using-python/
Beautiful Soup documentation	https://beautiful-soup-4.readthedocs.io/en/latest/
Tutorial: Web Scraping with Python Using Beautiful Soup	https://www.dataquest.io/blog/web-scraping-python-using-beautiful-soup/
Beautiful Soup: Build a Web Scraper with Python	https://realpython.com/beautiful-soup-web-scraper-python/
Selenium with Python	https://selenium-python.readthedocs.io/
Selenium automates browsers	https://www.selenium.dev/
ChromeDriver	https://chromedriver.chromium.org/getting-started

Table 5.1: *Resources on web automation in Python*

Questions

1. What languages are used by a web browser to render a webpage?
2. How can you automate filling up on online forms?

3. What is Selenium?
4. How do you build a web scrapper in Python?

CHAPTER 6

Automating File-Based Tasks

Introduction

In this chapter, we will look at various file-based automations for different file types in Python. We will discuss some of the Python libraries that are used to automate different file types. We will also look at ways to extract data from PDF documents and Word documents type file structure.

Structure

In this chapter, we will cover the following topics:

- Reading and writing files
- PDF documents automation
- Word documents automation
- Convert a PDF to a Word document

Objectives

After studying this chapter, you will be able to extract the text from PDF documents and generate new PDF documents. You will also be able to read and create new Word documents. You will further have the skills and understanding of Python libraries for working with a variety of file types.

Reading and writing files

A computer file is a contiguous set of bytes that is used to store data. The data is organized into the required format and can be anything from a simple text file to a computer application. These byte files are translated into **1** and **0** to be used by the computer.

Most of the file types contain three main parts:

- **Header:** Metadata containing information about the file such as file type, size, file name, and so on.
- **Data:** Contents of the file in bytes.
- **End of file (EOF):** A special character indicating the end of the file.

Python has many libraries that will help you work with different types of files. Some of the popular Python libraries for different file types are as follows:

- **`wave`:** Read and write WAV audio files (**https://docs.python.org/3/library/wave.html**).
- **`zipfile`:** Work with ZIP archives (**https://docs.python.org/3/library/zipfile.html**).
- **`configparser`: Create and read configuration files (https://docs.python.org/3/library/configparser.html**).
- **`xml.etree.ElementTree`**: Create and read XML-based files (**https://docs.python.org/3/library/xml.etree.elementtree.html**).
- **`PyPDF2`: PDF toolkit for reading and writing PDF documents (https://pypi.org/project/PyPDF2/**).
- **`openpyxl`:** Read and write Excel files (**https://openpyxl.readthedocs.io/en/stable/**).
- **`Pillow`:** Reading and manipulating image-based files (**https://pillow.readthedocs.io/en/stable/**).

We will use many of these libraries in this book to build work automations.

For working with text files in Python, it has an inbuilt **`open()`** function which is the key function for working with files. The **`open()`** function takes two parameters as arguments which are file location and mode. There are four different modes for opening a file using the **`open`** function:

- **`r`**: **Read** - Opens a file for reading.
- **`a`**: **Append** - Opens a file for adding more data and creates a new file if it does not exist.
- **`w`**: **Write** - Opens a file for writing and creates a new file if it does not exist.
- **`x`**: **Create** - Creates a new file.

In addition, you can specify if the file should be handled in the **binary** mode or **text** mode:

- `t`: Text mode.
- `b`: Binary mode (for example, for opening images).

As shown in *Figure 6.1*, we can use the `open` function to open a file for writing in the binary mode using the argument `wb` after the file name. The default file path is the path where the script is running if no specific file path is specified:

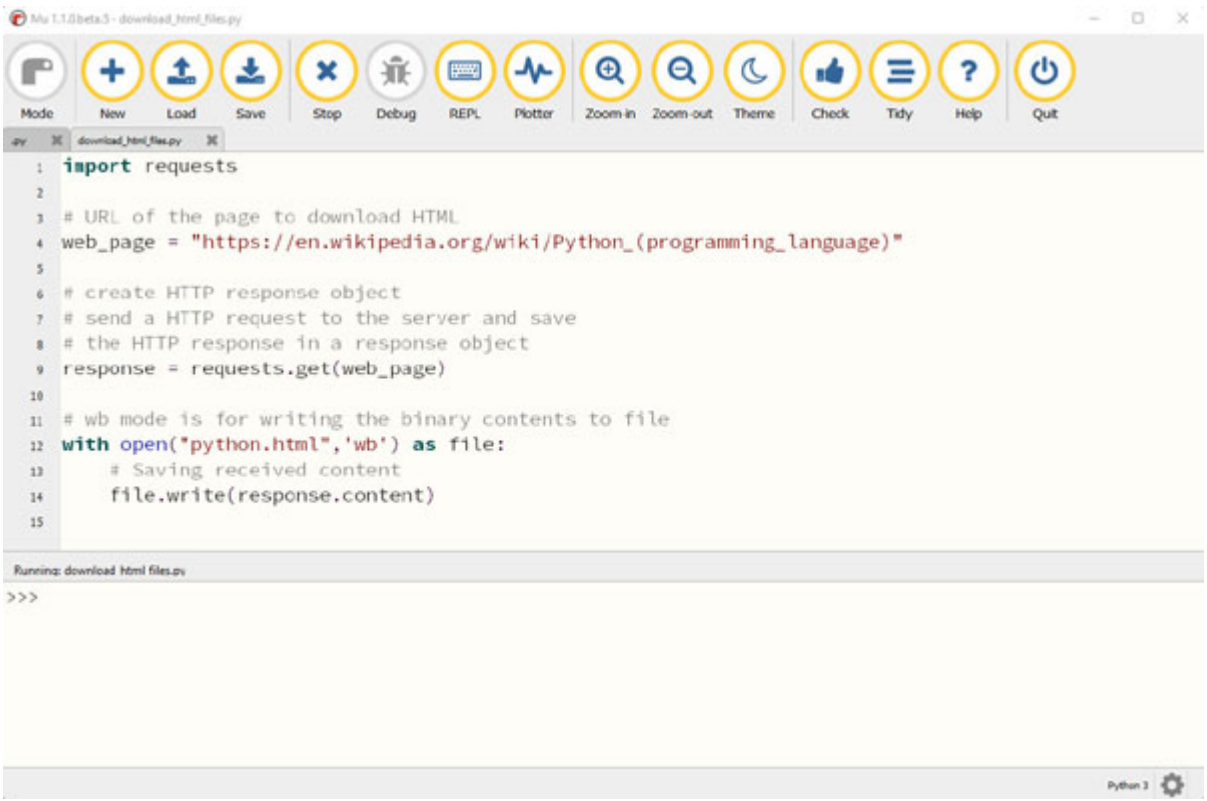


Figure 6.1: *Opening a new file with the binary mode*

In the next section, we will take a look at PDF document automations, including ways to create and extract data from PDF documents.

PDF documents automation

PDF documents are widely used in a day-to-day work environment for a variety of purposes to present and exchange documents. In this section, we

will look at Python libraries that help with PDF-based task automation such as extracting PDF data and creating new PDF documents.

For extracting text from PDF documents, Python has two main libraries **`Pdfminer.six`** and **`PyPDF2`**. **`Pdfminer.six`** is one of the best Python packages for extracting information from PDF documents and has features to extract text, images, and tables from PDF documents. **`PyPDF2`** can do much more than just extracting text from PDF documents such as creating PDF documents, splitting documents, merging documents, cropping pages, merging multiple pages into a single page, and encrypting, and decrypting PDF files.

To install **`Pdfminer`**, use the **`mu`** package manager, type **`Pdfminer.six`**, and click on **`OK`** as shown in the following figure:

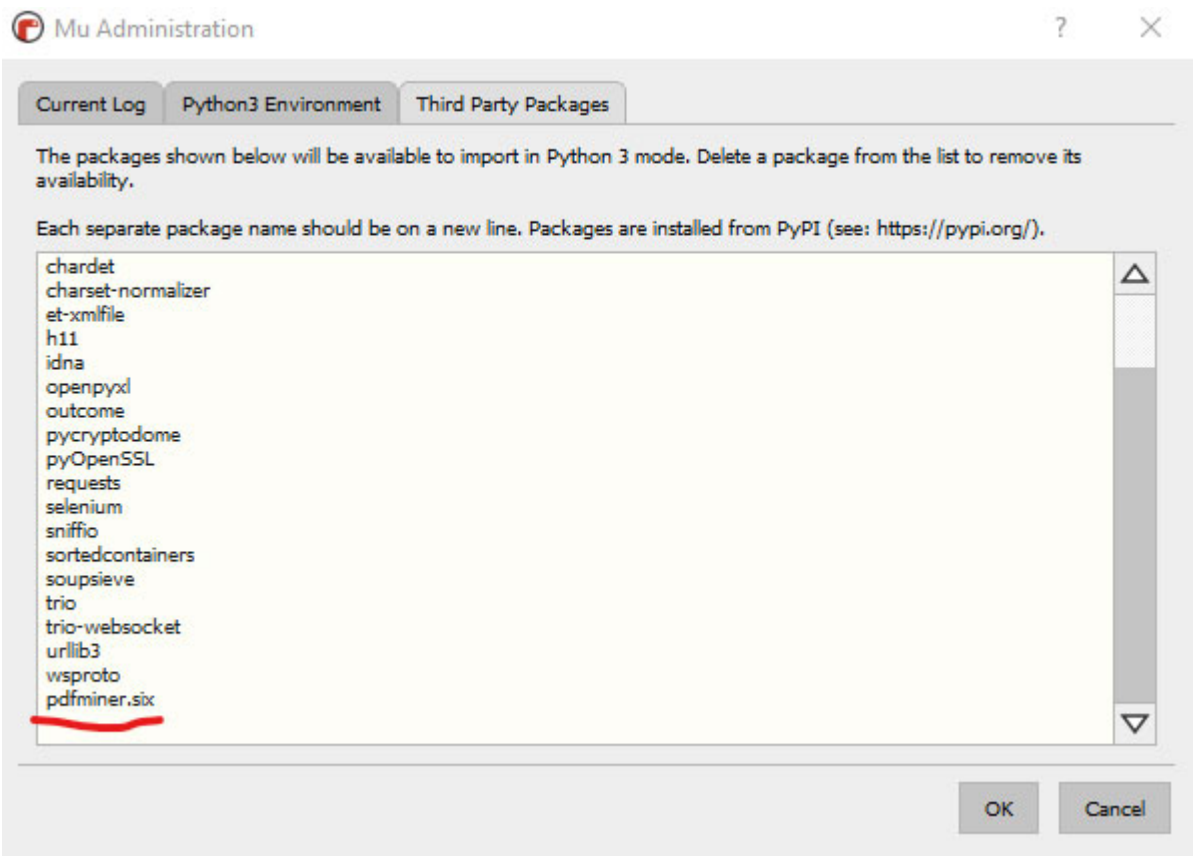


Figure 6.2: *Mu package manager*

PDF miner has the **`extract_text`** function which is used for extracting text from PDF documents. It takes the following parameters to extract the text data:

- **`pdf_file`**: PDF file path or file object.
- **`password`**: For encrypted PDFs, the password to decrypt the document.
- **`page_numbers`**: The page numbers to extract the text from (index starts from **0**).
- **`maxpages`**: The maximum number of pages to extract the text from.
- **`caching`**: If resources should be cached.
- **`codec`**: Text character encoding (by default, it used **UTF-8**).
- **`laparams`**: An **LAParams** object from **`pdfminer.layout`** to send the layout of the document.

The functions return a string containing all the text data extracted as shown in *Figure 6.3*:

Figure 6.3: *Extract the text from the PDF document*

With the Python library **`PyPDF2`**, you can create PDF documents as well. To install **`PyPDF2`**, use the **`mu`** package manager, type **`PyPDF2`**, and click on **`OK`** as shown in the following figure:

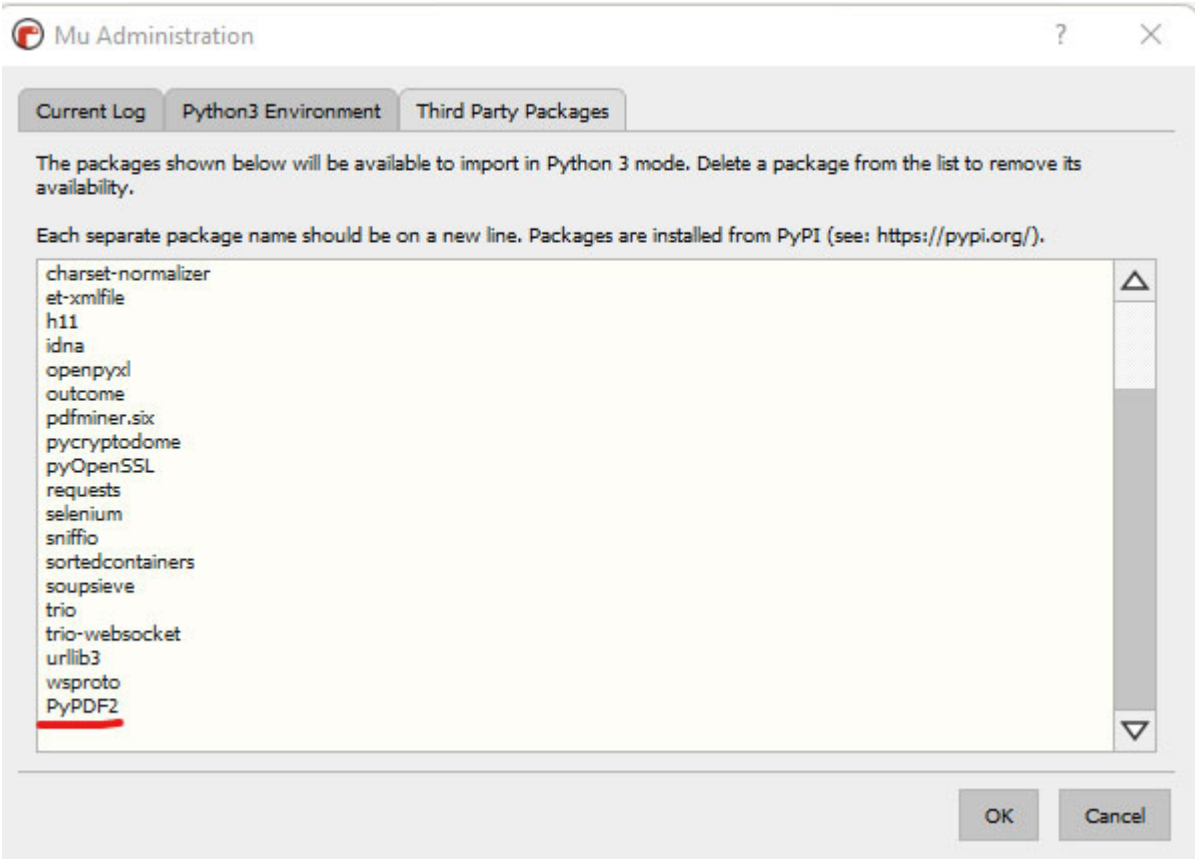


Figure 6.4: *Mu package manager*

`PyPDF2` can allow you to extract useful data from any PDF. For example, you can extract details like the author of the document, title and subject, and number of pages. As shown in *Figure 6.5*, use the **`getNumPages()`** function to get the number of pages in the PDF document and the **`documentInfo`** function to get more information on the PDF document:

```python
from PyPDF2 import PdfFileReader

pdf = PdfFileReader("large_file.pdf")

# getting the number of pages in PDF
print("Number of pages: " + str(pdf.getNumPages()))

# Printing document information
print(pdf.documentInfo)
```

```
Number of pages: 13
{'/Producer': 'dvips + GPL Ghostscript GIT PRERELEASE 9.22', '/CreationDate': "D:20211216205629-05'00'", '/ModDate': "D:
20211216205629-05'00'", '/Creator': 'LaTeX with hyperref', '/Title': '', '/Subject': '', '/Author': '', '/Keywords': ''}
>>>
```

Figure 6.5: *Extracting the PDF information*

`PyPDF2` can also be used to extract the text from PDF documents using the **`extractText()`** function as shown in *Figure 6.5*:

```python
from PyPDF2 import PdfFileReader

pdf_reader = PdfFileReader("large_file.pdf")

with open("large_file.txt",'w', encoding='utf-8') as file:
    for page in pdf_reader.pages:
        text = page.extractText()
        file.write(text)
```

```
>>>
```

Figure 6.6: *Extracting the text using PyPDF2*

PyPDF2 supports creation of new PDF documents using the **PdfFileWriter** class. **PdfFileWriter** provides functions to create and add data to the new PDF documents such as:

- **addAttachment**: This function embeds a file inside the PDF taking in parameters as the filename and the data to be stored in the file.
- **addBlankPage**: This function appends a blank page to the PDF file and returns it with **width** and **height** as parameters.
- **appendPagesFromReader**: This function copies pages from the **PdfFileReader** reader to the writer. It takes the **PdfFileReader** object as a parameter.

More functions available for the **PdfFileWriter** class can be found on the **PyPDF2** documentation (**https://pypdf2.readthedocs.io/en/latest/modules/PdfWriter.html**). As shown in *Figure 6.7*, we can use the **addBlankPage** function to create a blank PDF with the specified **width** and **height**:

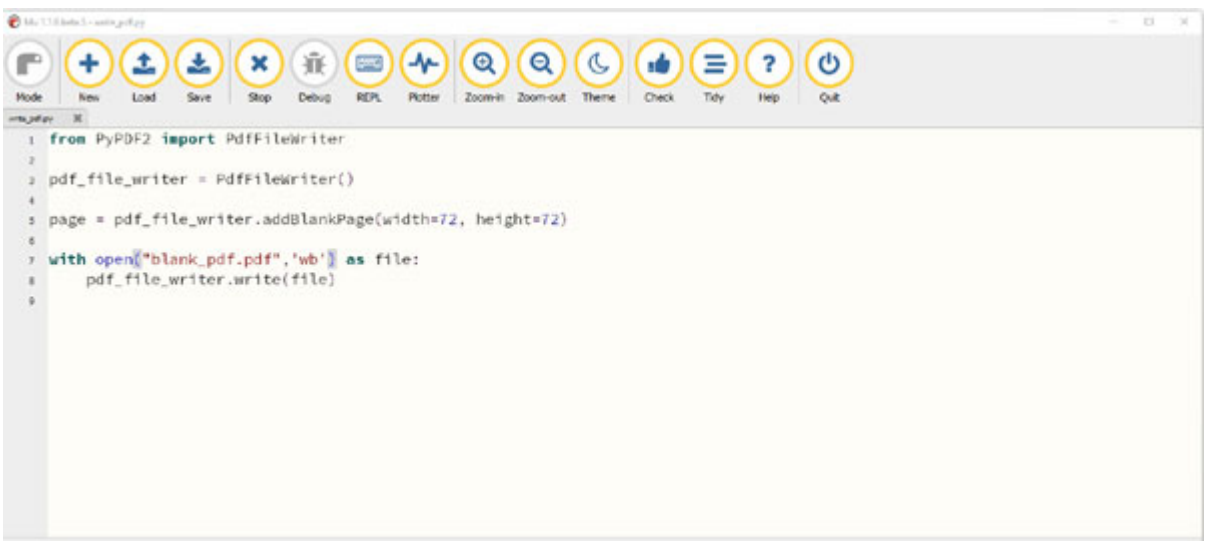


Figure 6.7: *Creating a new PDF document*

We can also copy the PDF data from one PDF document to another PDF document. We can selectively add pages to the PDF document using the **PdfFileWriter.addPage()** function as shown in *Figure 6.8*:

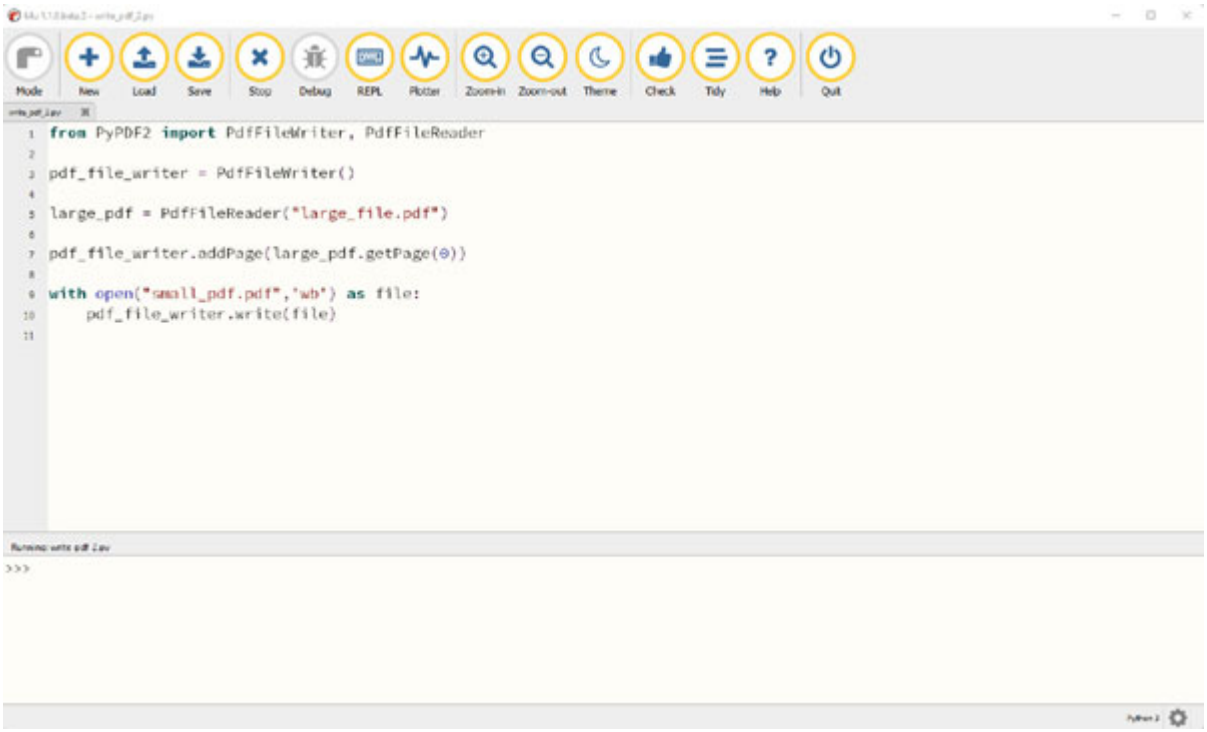
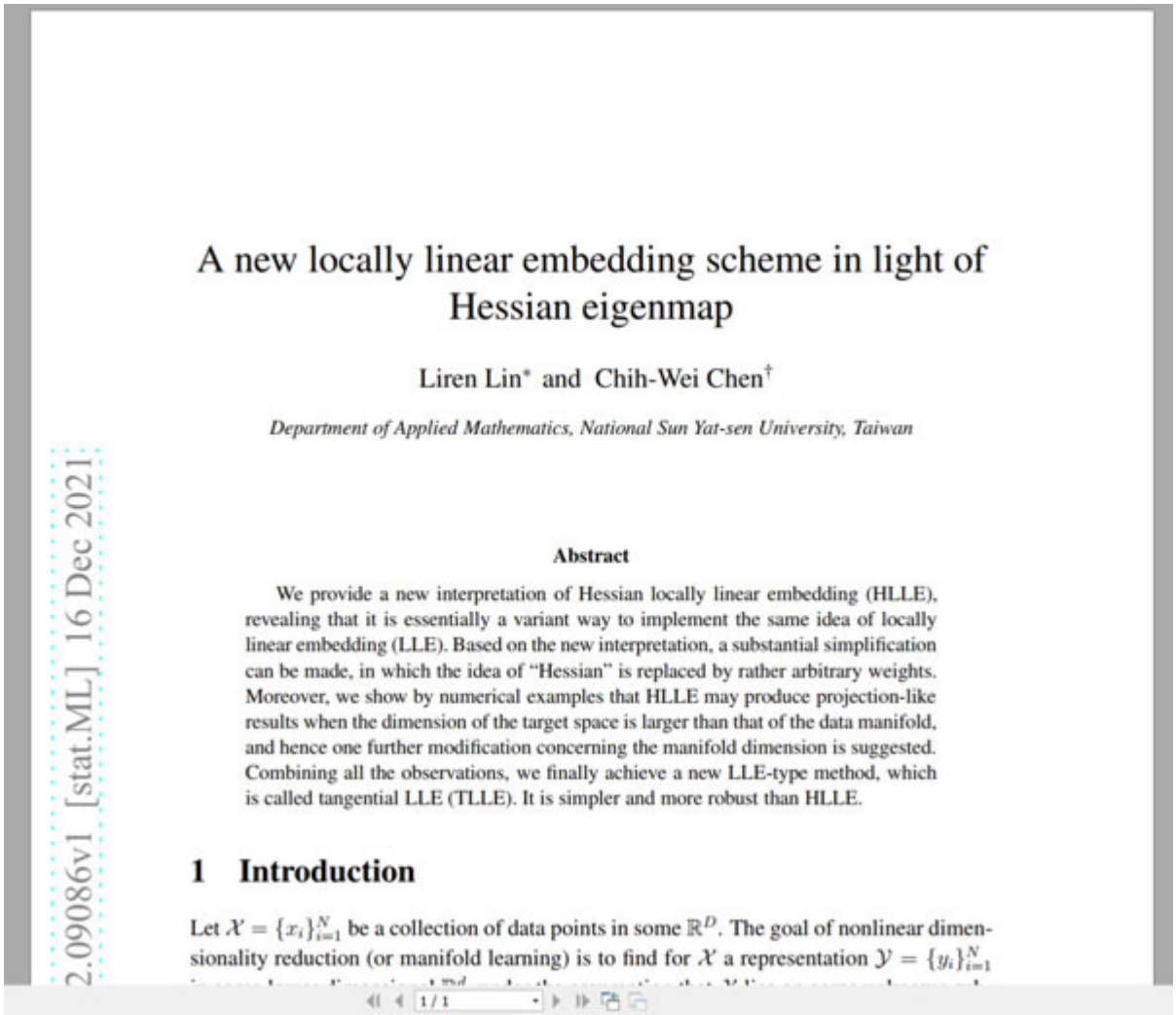


Figure 6.8: *Wringing a page from an existing PDF document*

In *Figure 6.9*, we can see the new PDF created by the **`PdfFileWriter`**:



A new locally linear embedding scheme in light of Hessian eigenmap

Liren Lin* and Chih-Wei Chen†

Department of Applied Mathematics, National Sun Yat-sen University, Taiwan

2.09086v1 [stat.ML] 16 Dec 2021

Abstract

We provide a new interpretation of Hessian locally linear embedding (HLLE), revealing that it is essentially a variant way to implement the same idea of locally linear embedding (LLE). Based on the new interpretation, a substantial simplification can be made, in which the idea of "Hessian" is replaced by rather arbitrary weights. Moreover, we show by numerical examples that HLLE may produce projection-like results when the dimension of the target space is larger than that of the data manifold, and hence one further modification concerning the manifold dimension is suggested. Combining all the observations, we finally achieve a new LLE-type method, which is called tangential LLE (TLLE). It is simpler and more robust than HLLE.

1 Introduction

Let $\mathcal{X} = \{x_i\}_{i=1}^N$ be a collection of data points in some $\mathbb{R}^D$. The goal of nonlinear dimensionality reduction (or manifold learning) is to find for $\mathcal{X}$ a representation $\mathcal{Y} = \{y_i\}_{i=1}^N$

Figure 6.9: *New PDF document*

In the next section, we will look at how to create and read Word documents in Python.

Word documents automation

Word documents are widely used to generate reports, research material, and keeping notes in our day-to-day work. Python has a **`python-docx`** library to read and write Microsoft Word (**`.docx`**) files.

To install **`python-docx`**, use the **`mu`** package manager, type **`python-docx`**, and click on **`OK`** as shown in the following figure:

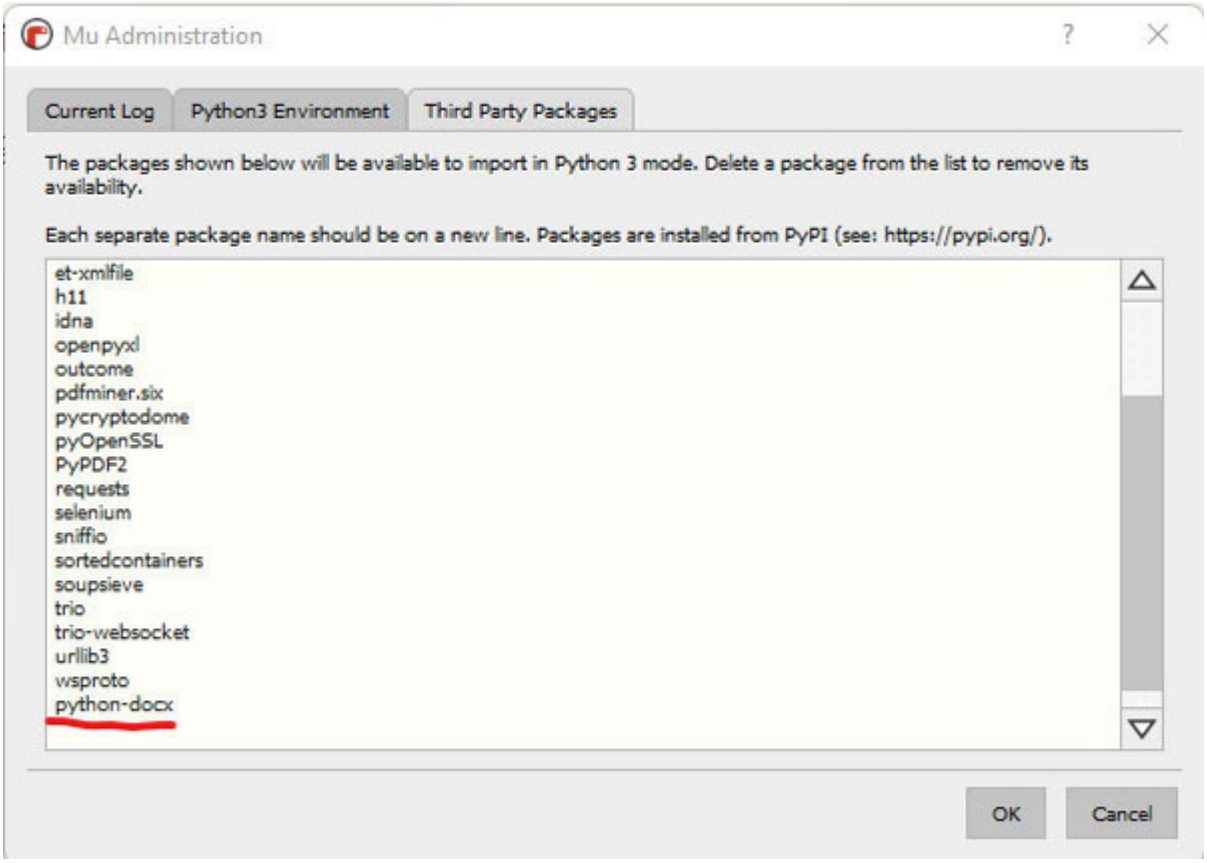


Figure 6.10: *Mu package manager*

The `python-docx` library has the `Document` class to create a blank document. The `Document` class has the following functions mentioned to create a new Word document:

- `add_paragraph()`: This function creates a new paragraph at the end of the document with taking the paragraph text as an argument and an optional `style` tag specifying the style for the Word document.
- `add_heading()`: By default, this function adds a top-level heading, what appears in Word as **Heading 1**. When you need a heading for a sub-section, just specify the level you want as an integer between **1** and **9**: `document.add_heading('The role of dolphins', level=2)`. If you specify a level of `0`, a `Title` paragraph is added. This can be handy to start a relatively short document that doesn't have a separate title page.
- `add_page_break()`: This function adds a page break to your document.

- **`add_table(rows=2, cols=2)`**: With the **`add_table`** function, you can create a new table in Word document. It takes the number of rows and columns as arguments. To add data to a particular cell, use the **`cell()`** function with the row and column as parameters or a **`for`** loop with the **`table.rows`** and **`row.cells`** properties.

  The following code adds data to the specified table cell:

  ```
  table.cell(0, 0)
  cell.text = 'My table'
  ```

  The following code allows you to loop through the rows and cells of the table:

  ```
  for row in table.rows:
   for cell in row.cells:
    cell.text = 'My Text
  ```

- **`document.add_picture(picture path)`**: This function allows you to add a picture to the Word document with the specified picture path.

In *Figure 6.11*, we can see an example of creating a new Word document by using the functions discussed previously:

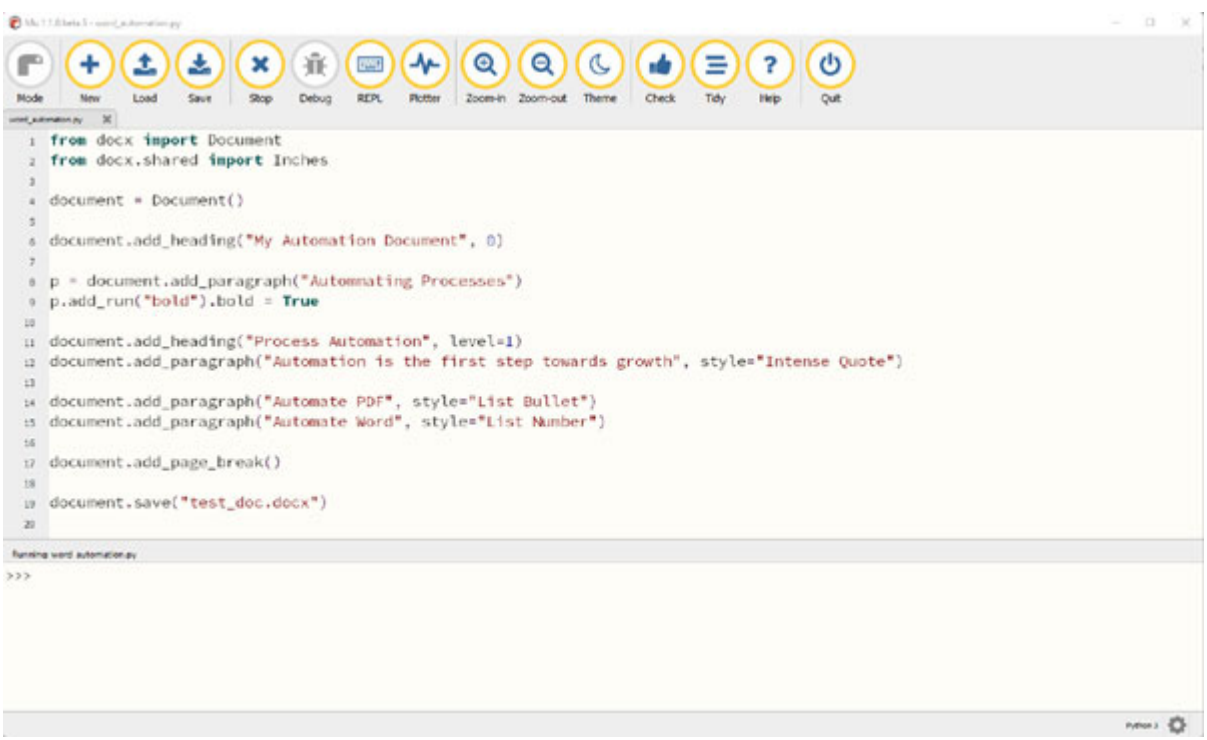


Figure 6.11: *Creating a new Word document*

After executing the document creation code, a new Word document is created with the specified styles as shown in *Figure 6.12*:

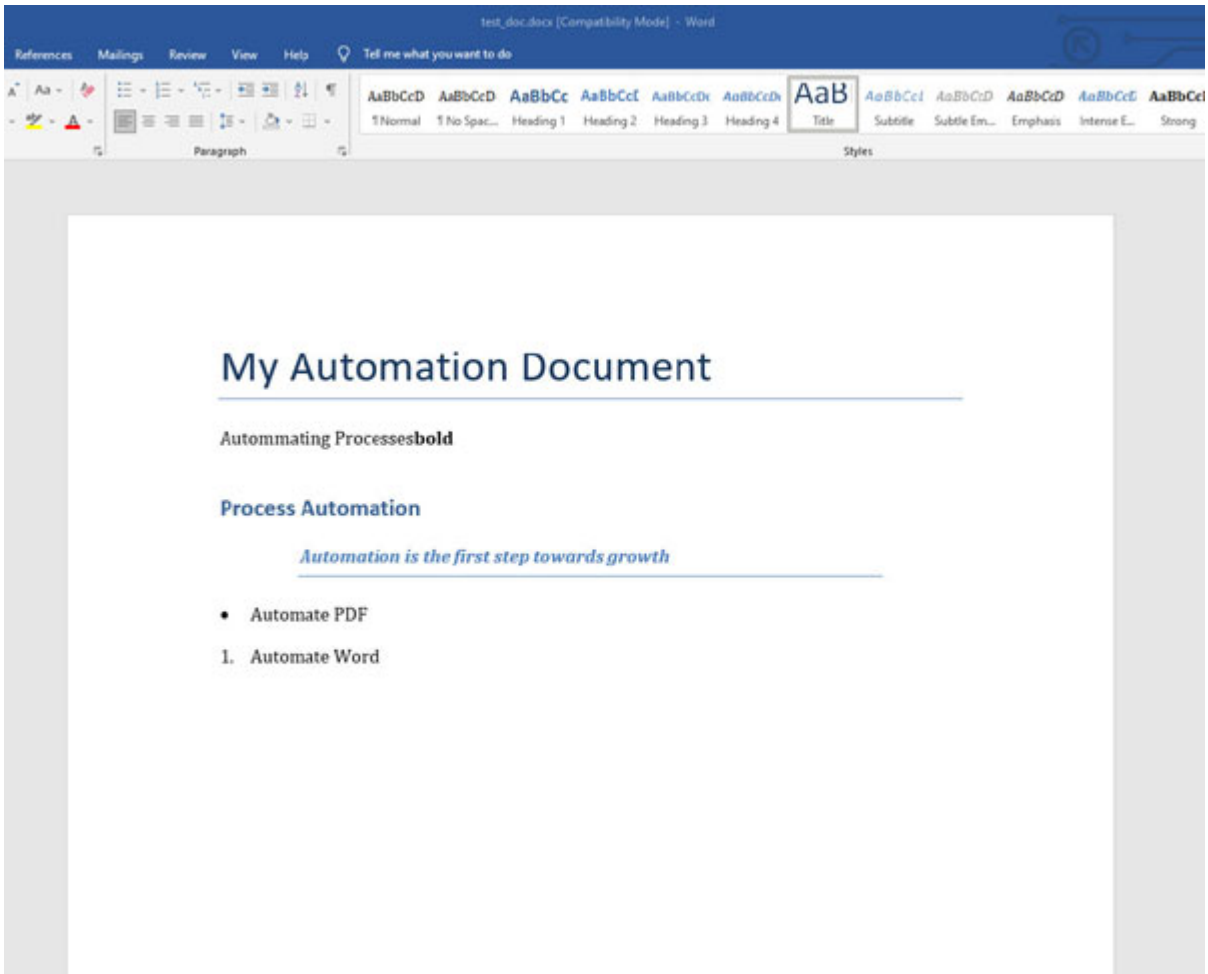


Figure 6.12: *New Word document*

The **`python-docx`** library also has a function to iterate and read the existing Word documents. To iterate through paragraphs, use the **`document.paragraph`** parameter as shown in *Figure 6.13*:

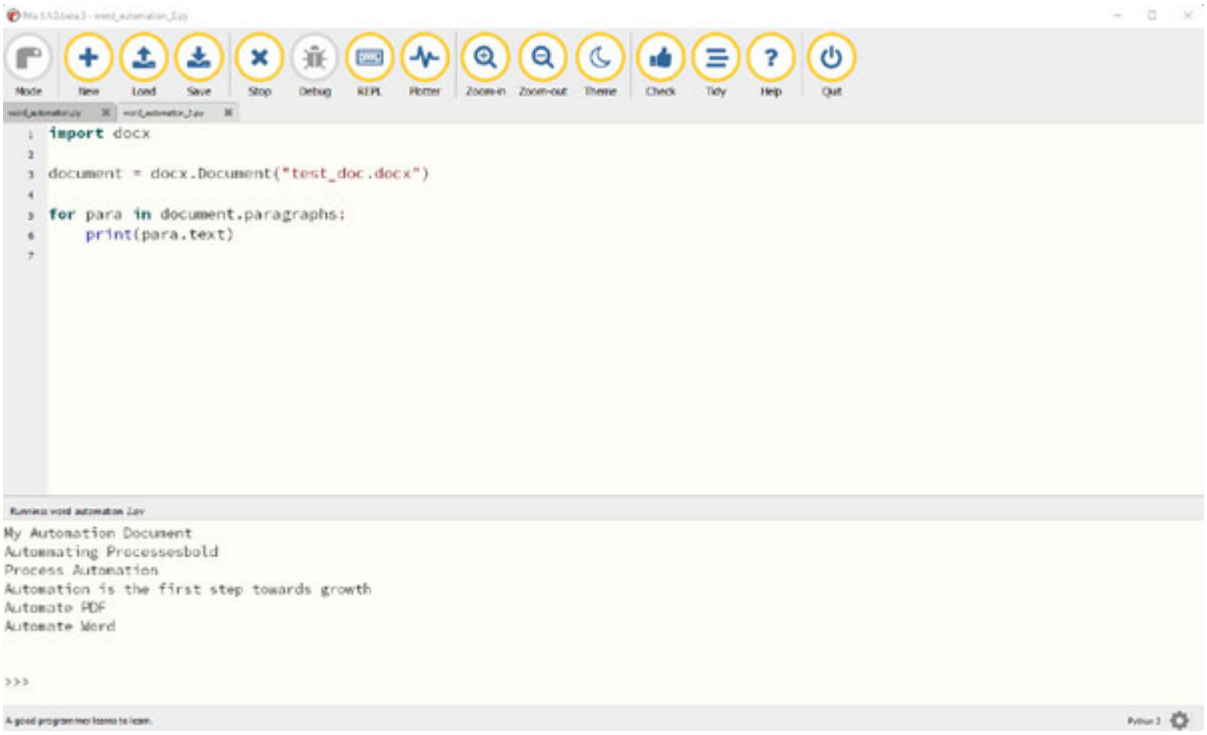


Figure 6.13: *Reading data from a Word document*

In the next section, we will look at a common automation requirement which is to convert a PDF document to a Word document to be able to easily read and manipulate the data contained inside the PDF document.

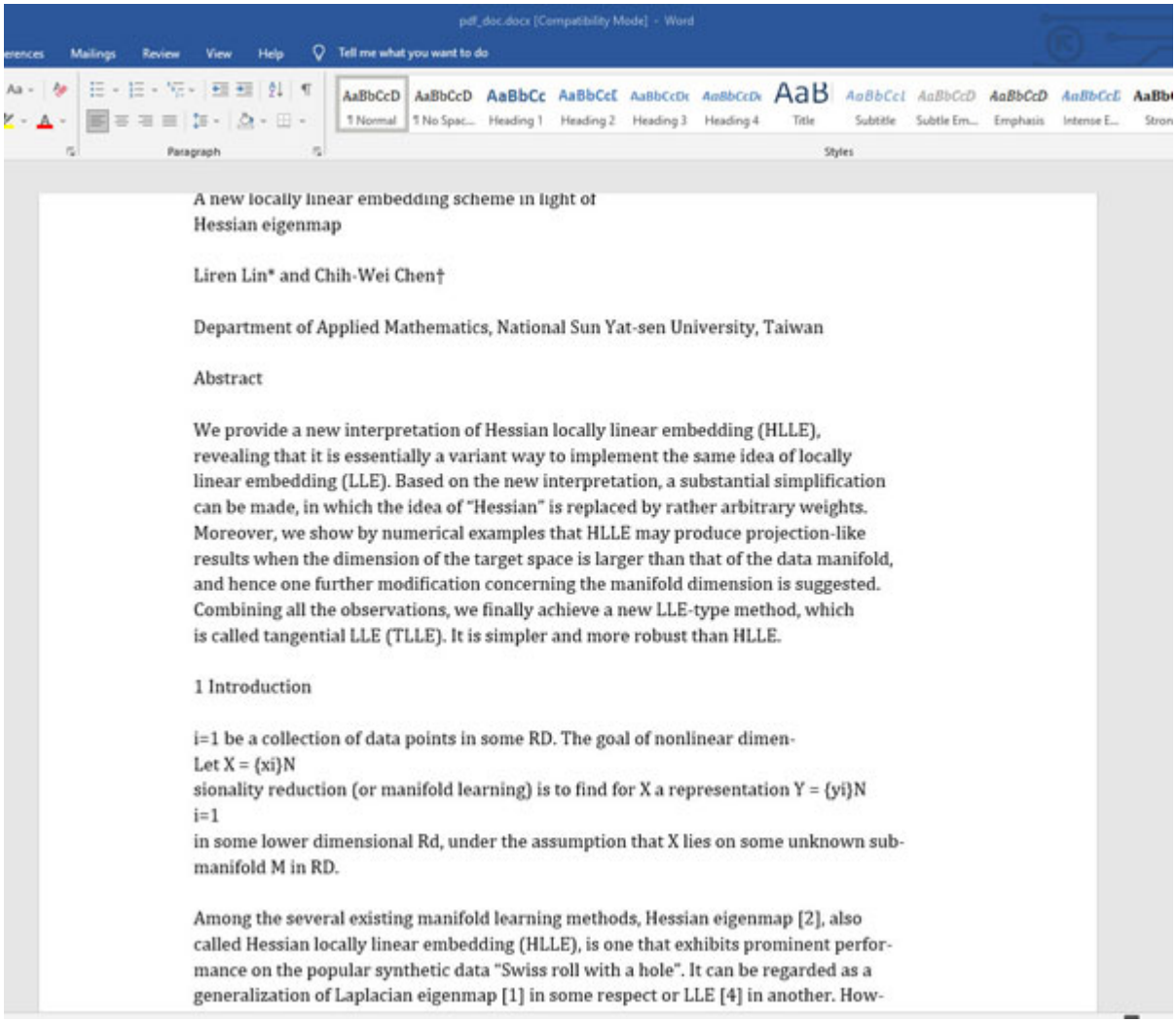
Convert a PDF to a Word document

We can easily convert a PDF document to a Word document using the **`Pdfminer`** and **`python-docx`** libraries. If the PDF document text contains invalid characters, we can remove those characters to support the Word document encoding format by a custom function. As shown in *Figure 6.14*, we will first read the PDF document using the **`extract_text()`** function and then add the extracted string to the Word document using the **`add_paragraph()`** function:

```python
import pdfminer
import pdfminer.layout
import pdfminer.high_level
from docx import Document
from docx.shared import Inches

text = pdfminer.high_level.extract_text('small_pdf.pdf')

def valid_xml_char_ordinal(c):
    codepoint = ord(c)
    # conditions ordered by presumed frequency
    return (
        0x20 <= codepoint <= 0xD7FF or
        codepoint in (0x9, 0xA, 0xD) or
        0xE000 <= codepoint <= 0xFFFD or
        0x10000 <= codepoint <= 0x10FFFF
        )
cleaned_string = ''.join(c for c in text if valid_xml_char_ordinal(c))

document = Document()
p = document.add_paragraph(cleaned_string)
document.add_page_break()
document.save("pdf_doc.docx")
```

Figure 6.14: *PDF to Word document*

Figure 6.15 shows the converted PDF document to the Word document by executing the PDF to Word conversion script:



A new locally linear embedding scheme in light of
Hessian eigenmap

Liren Lin* and Chih-Wei Chen†

Department of Applied Mathematics, National Sun Yat-sen University, Taiwan

Abstract

We provide a new interpretation of Hessian locally linear embedding (HLLE), revealing that it is essentially a variant way to implement the same idea of locally linear embedding (LLE). Based on the new interpretation, a substantial simplification can be made, in which the idea of "Hessian" is replaced by rather arbitrary weights. Moreover, we show by numerical examples that HLLE may produce projection-like results when the dimension of the target space is larger than that of the data manifold, and hence one further modification concerning the manifold dimension is suggested. Combining all the observations, we finally achieve a new LLE-type method, which is called tangential LLE (TLLE). It is simpler and more robust than HLLE.

1 Introduction

i=1 be a collection of data points in some RD. The goal of nonlinear dimen-
Let X = {xi}N
sionality reduction (or manifold learning) is to find for X a representation Y = {yi}N
i=1
in some lower dimensional Rd, under the assumption that X lies on some unknown sub-
manifold M in RD.

Among the several existing manifold learning methods, Hessian eigenmap [2], also called Hessian locally linear embedding (HLLE), is one that exhibits prominent perfor-
mance on the popular synthetic data "Swiss roll with a hole". It can be regarded as a generalization of Laplacian eigenmap [1] in some respect or LLE [4] in another. How-

Figure 6.15: *PDF to Word document converted file*

Conclusion

In this chapter, we covered a lot of content on file-based automations for different file types in Python. We looked at ways to extract data from PDF documents and create new PDF documents. We also looked at ways to create new Word documents and convert PDF documents to Word documents.

In the next chapter, we will look at ways to automate email-based tasks using **Gmail**, **Outlook**, and **SMTP** clients. We will also look at text message automation using the **Twilio** API and messaging automation using Slack APIs.

Further reading

There are a lot of online resources to help you learn more about file automation with Python. The following table lists some of the best resources to further improve your learning on File automation in Python:

Resource name	Link
Reading and writing files in Python	https://realpython.com/read-write-files-python/
Python PDF parser	https://github.com/euske/pdfminer
Extract text from a PDF using Python	https://pdfminersix.readthedocs.io/en/latest/tutorial/highlevel.html
PyPDF2 documentation	https://pypdf2.readthedocs.io/en/latest/

Table 6.1: *Resources on web automation in Python*

Questions

1. How do you read different types of files in Python?
2. How can you extract data from PDF document?
3. What are different Python libraries for working with PDF documents?
4. How can you build an automation to convert a PDF document into Word document?

CHAPTER 7

Automating Email, Messenger Applications, and Messages

Introduction

In this chapter, we will learn how to automate email-based tasks using *Gmail*, *Outlook*, and other SMTP clients. We will also look at text message and *WhatsApp* automation using the *Twilio* API.

Structure

In this chapter, we will cover the following topics:

- Simple Mail Transfer Protocol
- Sending emails using Gmail
- Outlook email automation
- Text and WhatsApp message automation

Objectives

After studying this chapter, you will be able to automatically read and send emails through Gmail and Outlook applications in Python. You will also be able to automatically send text messages using the *Twilio* APIs and WhatsApp messages using the WhatsApp web application.

Simple Mail Transfer Protocol

Simple Mail Transfer Protocol (**SMTP**) is the protocol system to send emails on the web. It is used by many email applications to send and receive emails on the web. The SMTP protocol ensures that the message is sent to the right receiving server and the receiver server makes sure the message is delivered to the correct end recipient.

Python has a built-in library called **`smtplib`** that is used for sending emails using the SMTP protocol. The **`smtplib`** can be imported using the **`import smtplib`** statement. The **`smtplib`** library has an SMTP function to connect to the server with the parameters as:

1. `smtpObj = smtplib.SMTP( [host [, port [, local_hostname]]] )`

The parameters used in the SMTP function are as follows:

- **`host`**: This is the IP address or a domain name of the SMTP server running your email service.
- **`port`**: This is the port number required with the host argument to point to the port where the SMTP server is listening. Generally, this is set to **`25`**.
- **`local_hostname`**: If your SMTP server is running on your local machine, then you can specify just local host to refer to the local server.

The SMTP object has a method called **`sendmail`** that is used for sending the email. It takes the following parameters:

- **The sender**: This is a string with the address of the sender.
- **The receivers**: This is a list of strings, one for each recipient.
- **The message**: This is a message as a formatted string (can be an HTML string as well).

The **`smtplib`** client can communicate with a remote SMTP server by supplying the outgoing mail server as given in the following statement - **`smtplib.SMTP('mail.your-domain.com', 25)`**.

In the next section, we will look at a real-work example of automating the sending of the emails using Gmail.

Sending emails using Gmail

We will use the **`ssl`** library and SMTP library for sending emails using Gmail. To use these libraries, we need to allow a less secure app option to be **ON** (**https://myaccount.google.com/lesssecureapps**) to allow the use of these libraries with password-based authentication. This setting is not

available for Gmail accounts with *two-step verification* enabled. If you do not want to enable this option in the Gmail account, then you can use the OAuth2 authorization framework and follow the Gmail API documentation (**https://developers.google.com/gmail/api/quickstart/python**).

The `ssl` library has the `create_default_context()` function which returns a new `SSLContext` object with default settings. The `smtplib` has `SMTP_SSL()` function behaves exactly the same as the SMTP function taking arguments as per this definition: `smtplib.SMTP_SSL(host='', port=0, local_hostname=None, keyfile=None, certfile=None, [timeout, ]context=None, source_address=None)`. `SMTP_SSL` is used for situations where SSL is required from the beginning of the connection. We can create a connection to Gmail using these functions as shown in *Figure 7.1*:

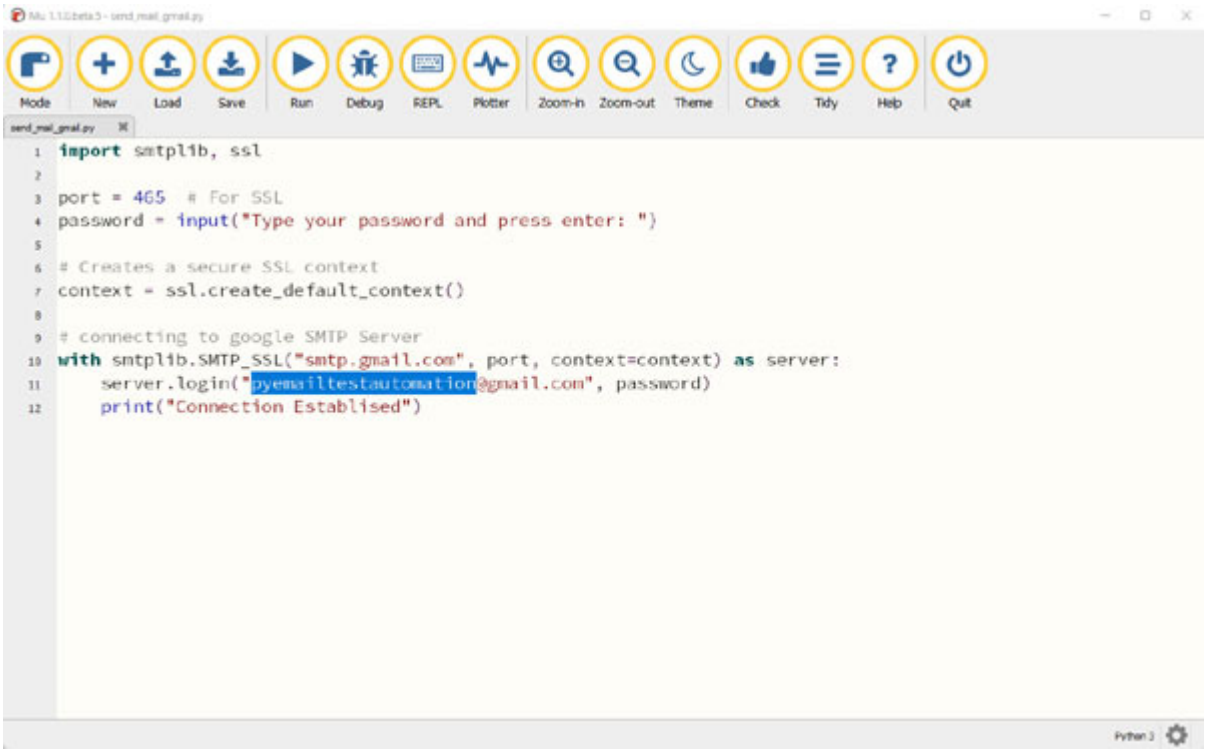


Figure 7.1: *Establishing a connection with Gmail*

Once the connection is established, you can send emails using the `sendmail()` function with the sender email, receiver email, and message as arguments as shown in *Figure 7.2*:

```python
import smtplib, ssl

port = 465  # For SSL
password = input("Type your password and press enter: ")

# Creates a secure SSL context
context = ssl.create_default_context()

message = """\
Subject: Hi there

I am automating sending of email."""

sender_email = "pyemailtestautomation@gmail.com"
receiver_email = "pyemailtestautomation@gmail.com"

# connecting to google SMTP Server
with smtplib.SMTP_SSL("smtp.gmail.com", port, context=context) as server:
    server.login(sender_email, password)
    server.sendmail(sender_email, receiver_email, message)
```

Figure 7.2: *Send emails using Gmail*

You can verify that the message is correctly sent with a proper subject and message by opening Gmail in the browser as shown in *Figure 7.3*:

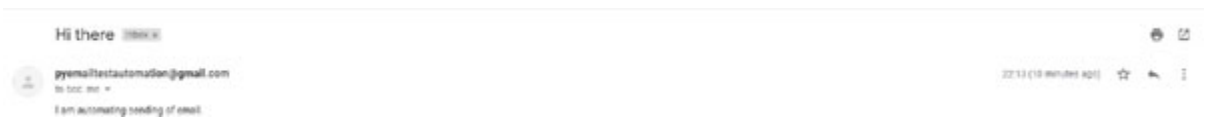


Figure 7.3: *Gmail message sent by an automation script*

The SMTP library has the **Multipurpose Internet Mail Extensions** (**MIME**) object support that is used to send attachments and HTML messages. We will use **`MIMEMultipart`** and **`MIMEText`** to send HTML-based emails. The **`MIMEMultipart`** object can be created using the **`MIMEMultipart()`** and the HTML message can be attached using the **`MIMEText(html_message, 'html')`** function which takes **`html_message`** and message type as **`html`** as arguments. You can attach the **`MIMEText`** to the **`MIMEMultipart`** object using the **`attach`** function to send HTML-based emails as shown in *Figure 7.4*:

```python
import smtplib, ssl
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText

password = input("Type your password and press enter: ")
context = ssl.create_default_context()
sender_email = "pyemailtestautomation@gmail.com"
receiver_email = "pyemailtestautomation@gmail.com"
# Creation of the MIMEMultipart Object
message = MIMEMultipart()
message["Subject"] = "Automation Bot"
message["From"] = sender_email
message["To"] = receiver_email
# HTML message
html_message = """<html>
    <head></head>
    <body>
        <p style='color:red;'>I am automating sending of email.</p>
        <p>Automation Bot</p>
    </body>
</html>"""
# Creation of a MIMEText Part
htmlPart = MIMEText(html_message, 'html')
# Part attachment
message.attach(htmlPart)

# connecting to google SMTP Server
with smtplib.SMTP_SSL("smtp.gmail.com", 465, context=context) as server:
    server.login(sender_email, password)
    server.sendmail(sender_email, receiver_email, message.as_string())
```

Figure 7.4: *Sending an HTML email message on Gmail*

You can verify that the HTML message is formatted correctly by opening this email in the browser as shown in *Figure 7.5*:

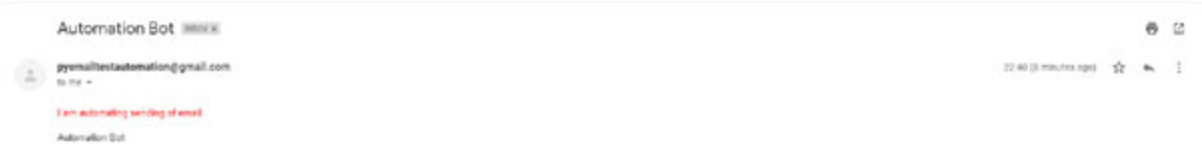


Figure 7.5: *HTML message sent by an automation bot*

You can also send the email with an attachment using the MIME object. You will need to open the file and attach the file to the **`MIMEBase("application", "octet-stream")`** part using the **`set_payload()`** function. You will also need to encode the file to send by an email and add a header as *key/value pair* to the attachment part as shown in *Figure 7.6*:

```python
import smtplib, ssl
from email.mime.multipart import MIMEMultipart
from email.mime.text import MIMEText
from email import encoders
from email.mime.base import MIMEBase

password = input("Type your password and press enter: ")
context = ssl.create_default_context()
sender_email = "pyemailtestautomation@gmail.com"
receiver_email = "pyemailtestautomation@gmail.com"
message = MIMEMultipart()
message["Subject"] = "Automation Bot"
message["From"] = sender_email
message["To"] = receiver_email
# File name located in same directory as this script
filename = "large_file.pdf"
with open(filename, "rb") as attachment:
    part = MIMEBase("application", "octet-stream")
    part.set_payload(attachment.read())
# Encode file in ASCII characters to send by email
encoders.encode_base64(part)
# Add header as key/value pair to attachment part
part.add_header(
    "Content-Disposition",
    f"attachment; filename= {filename}",
)
message.attach(part)

# connecting to google SMTP Server
with smtplib.SMTP_SSL("smtp.gmail.com", 465, context=context) as server:
    server.login(sender_email, password)
    server.sendmail(sender_email, receiver_email, message.as_string())
```

Figure 7.6: *Attaching the file in the Gmail*

You can verify that the attachment is sent correctly by opening the email in the browser as shown in *Figure 7.7*:

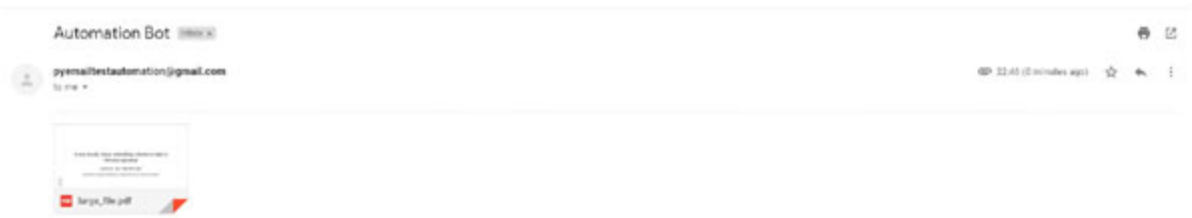


Figure 7.7: *Attachment received via an automation bot*

When you use the Python automation to send emails, Gmail adds all these emails to the **`Send`** folder and you can audit this folder to verify that all the messages are sent as per the automation requirement as shown in *Figure 7.8*:

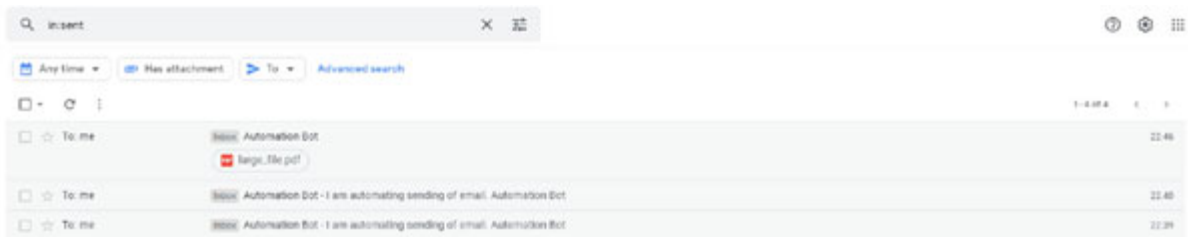


Figure 7.8: *List of emails sent by an automation bot*

In the next section, we will look at Outlook application email automations. The Outlook application automation can be used with any email provider as long as the email is configured in the Outlook application.

Outlook email automation

For automation of Outlook applications, we will use the `pywin32` library which provides access to Windows APIs functions. The Windows API (also known as **Win32**) is an application programming interface written by Microsoft to allow access to Windows features. The main components of the Windows API are as follows:

- **WinBase**: Windows kernel functions, `CreateFile`, `CreateProcess`, and so on.
- **WinUser**: Windows GUI functions, `CreateWindow`, `RegisterClass`, and so on.
- **WinGDI**: Windows graphics functions, `Ellipse`, `SelectObject`, and so on.

To install `pywin32`, use the `mu` package manager, type `pywin32`, and click on `OK` as shown in the following *Figure in 7.9*:

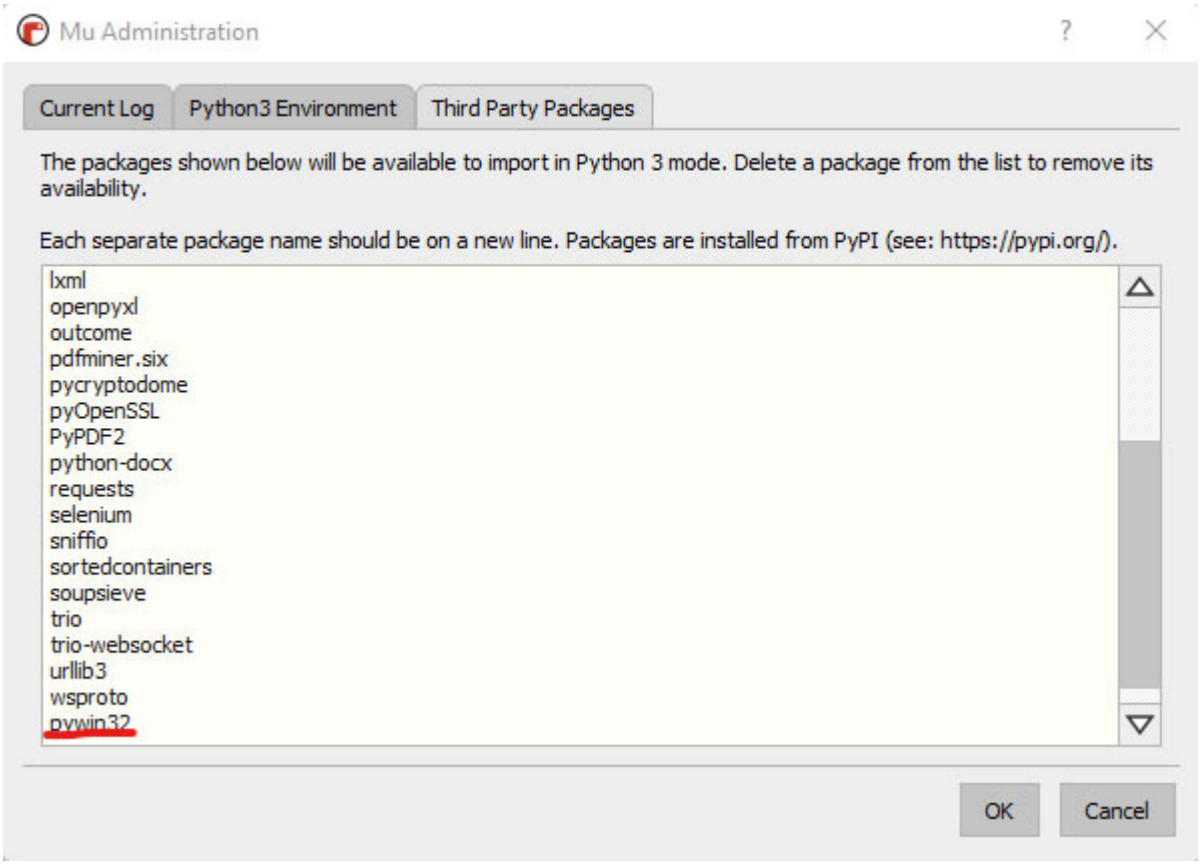


Figure 7.9: *Mu package manager*

We will use the **`win32com.client.DispatchEx()`** function with **`Outlook.Application`** as an argument which will open the Outlook application on your computer. The **`CreateItem`** method creates and returns a new Microsoft Outlook item which can be used to create a new email to send to the desired recipient. You can add **`mail.To`**, **`mail.Subject`**, and **`mail.HtmlBody`** to the Outlook item, and send the email using the **`Send`** function as shown in *Figure 7.10*:

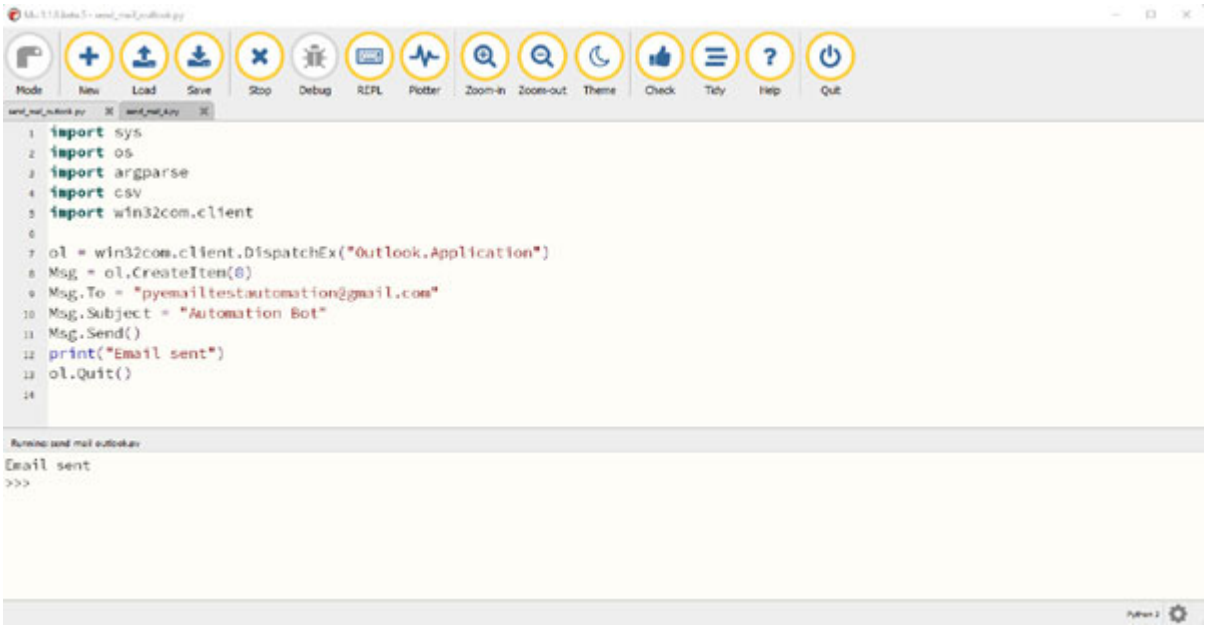


Figure 7.10: *Send an email using the Outlook application*

In the next section, we will look at text and WhatsApp automations. We will use *Twilio* APIs for text message automation and the **`pywhatkit`** library for WhatsApp automation.

Text and WhatsApp message automation

Twilio provides communication APIs for sending and receiving SMS messages, making voice calls, and video calls, and accessing other communication tools such as **chat** and **emails**. They have multiple product lines to automate a number of communication channels and the platform is used by businesses worldwide as a customer engagement platform.

In this chapter, we will only use the Twilio APIs for SMS automation. To install *Twilio*, use the **`mu`** package manager, type **`twilio`**, and click on **`OK`** as shown in the following *Figure 7.11*:

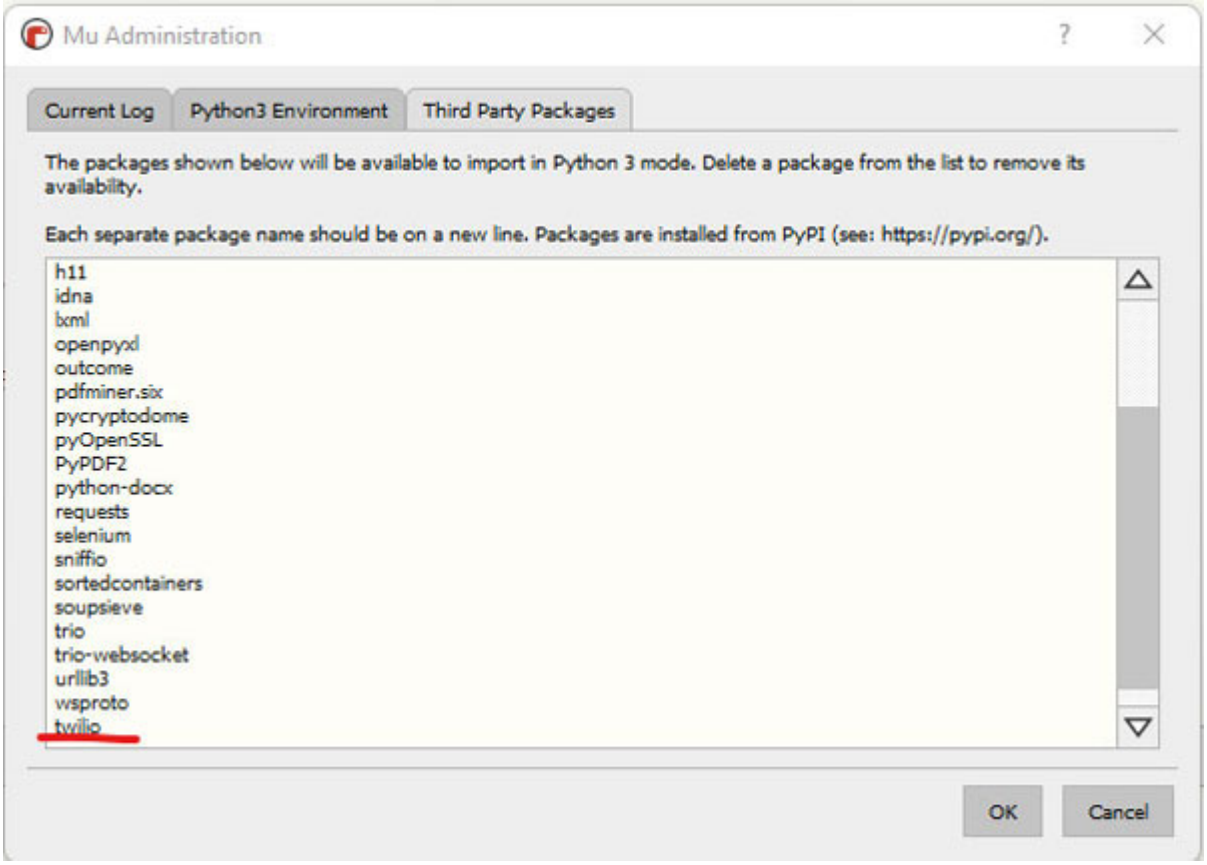


Figure 7.11: *Mu package manager*

To use the Twilio APIs for SMS automation, you will have to register for a Twilio account at the Twilio website (**https://www.twilio.com/**) and create a trial number for testing the SMS automation as shown in *Figure 7.12*:

Figure 7.12: *Getting a Twilio phone number*

You will also get a trial balance when you create a Twilio account which can be used for testing SMS automations (see *Figure 7.13*). There are also

other API providers which provide SMS automation services and you can use them as well instead of Twilio if that better suits your requirements:

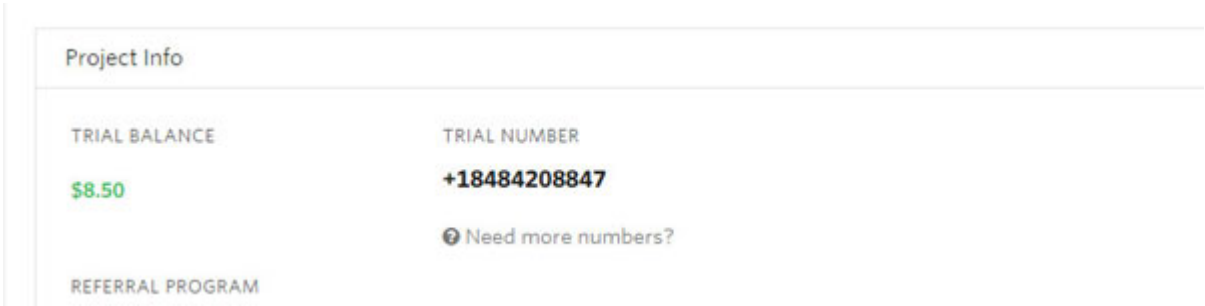


Figure 7.13: *Validating a Twilio number and balance*

When you have a Twilio Account, you will get an **account SID** and **Auth token**. These are your API credentials that will allow you to authenticate and use the Twilio API. Once you are authenticated with the API, you can send the message with the `message.create()` function passing the message text with senders and recipient number as shown in *Figure 7.14*:

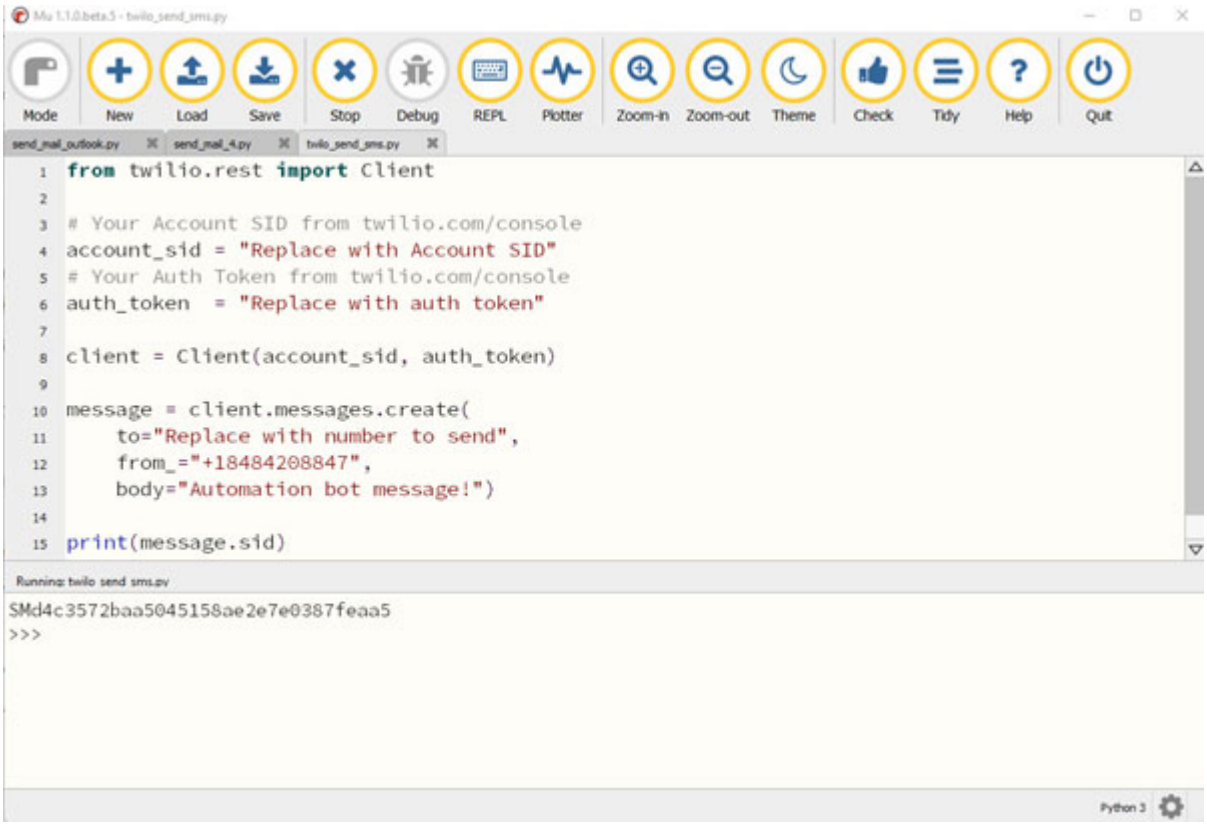


Figure 7.14: *Sending a text message using Twilio*

Once you run the code, the message would be sent to the desired recipient, and you can verify by checking this message on the recipient's phone as shown in *Figure 7.15*.

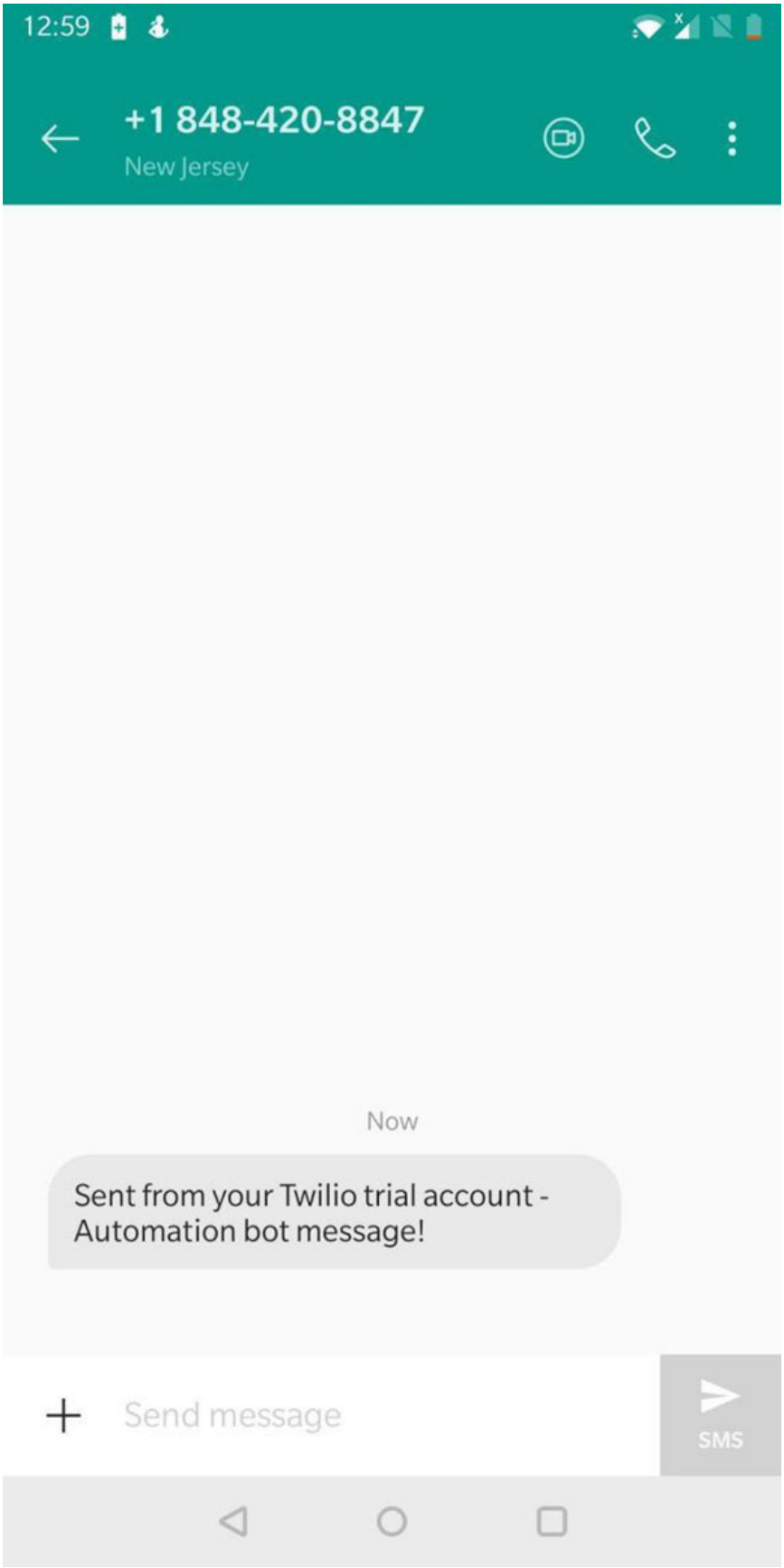

12:59
+1 848-420-8847
New Jersey
Now
Sent from your Twilio trial account - Automation bot message!
Send message
SMS

Figure 7.15: Verifying the message received on the mobile

WhatsApp Messenger is another popular messaging application which is used by users worldwide. We can automate WhatsApp messages using the Twilio APIs or using the **`pywhatkit`** library. **`PyWhatKit`** is a Python library that allows you to easily automate sending messages or images to a WhatsApp group or contact. It does not require any external API access and is easy to set up to automate simple messaging tasks on WhatsApp. To install **`PyWhatKit`**, use the **`mu`** package manager, type **`pywhatkit`**, and click on **`OK`** as shown in the following figure:

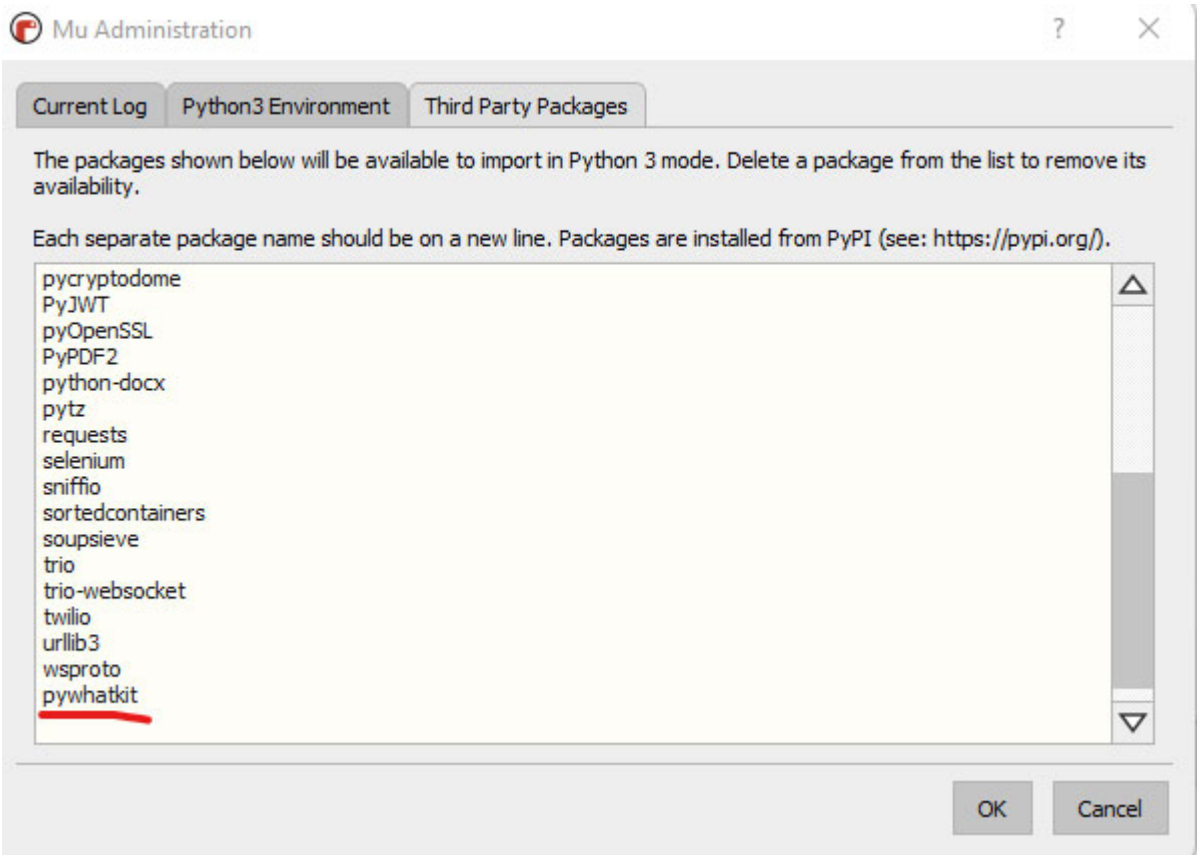


Figure 7.16: Mu package manager

Once the library is installed, sign in on the WhatsApp web using your WhatsApp account on your default browser (**https://web.whatsapp.com/**). **PyWhatKit** uses the WhatsApp web account to automate sending of WhatsApp messages.

The **`sendwhatmsg()`** function is used to send WhatsApp messages to a given contact at a particular time taking in arguments as the recipient number

(write the phone number with the international code (+...) of the country you want to send the automated message.), message, and time. The time is denoted using the *24 hour format*; for example, to send a message at *1:17 pm,* you would use arguments as `13: 17` as shown in *Figure 7.17*:

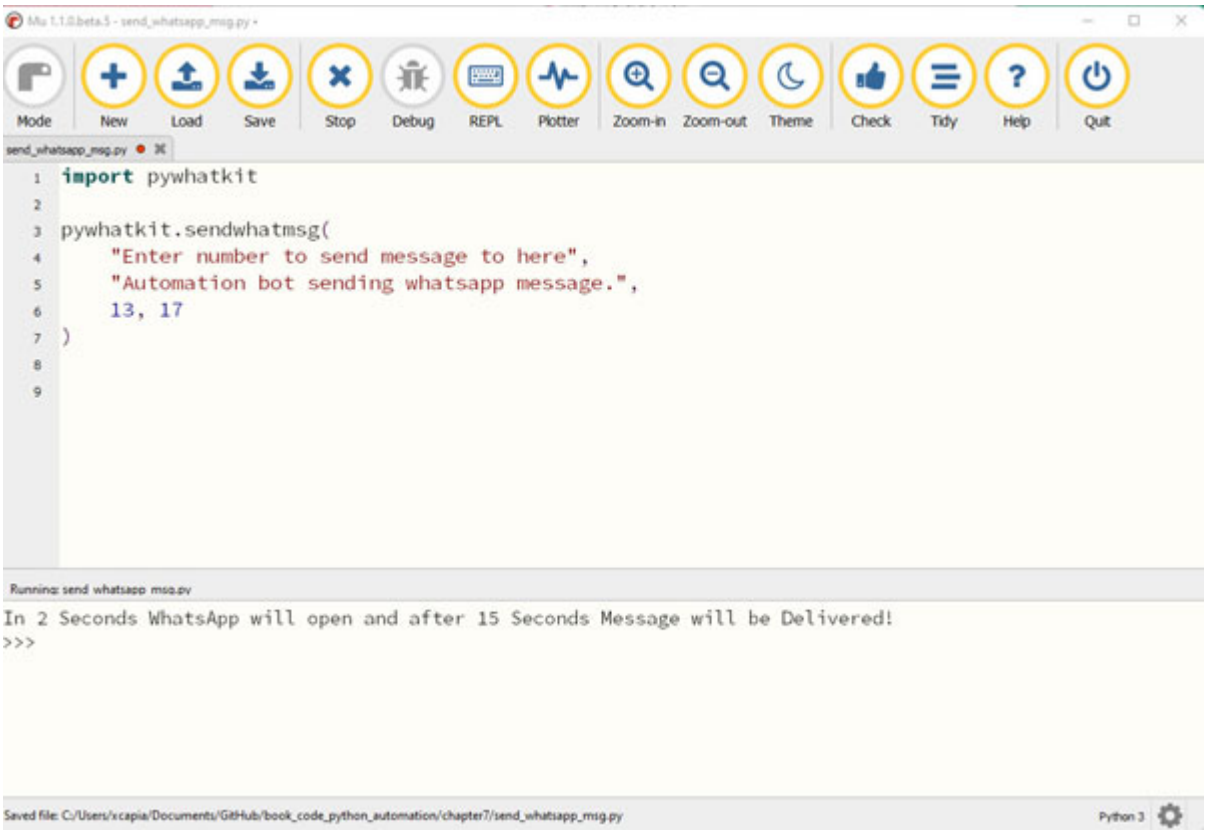


Figure 7.17: *Sending a WhatsApp message at 1:17 pm*

Once you run the code, the message would be sent to the desired recipient, and you can verify by clicking on the contact and seeing the message as sent in your WhatsApp message history. As the WhatsApp application keeps on updating to newer versions, you may find instances where the message was typed in the chat but is not sent and you may have to manually hit the *Sent* button. You can automate the clicking of the *Send* button using the Selenium web automation library discussed in *Chapter 5: Automating Web-Based Tasks*.

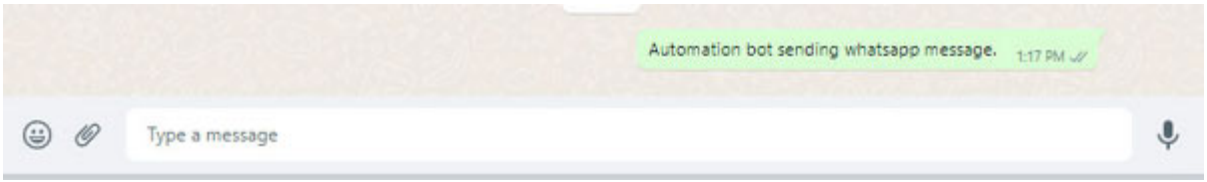


Figure 7.18: Verifying that the WhatsApp message is sent using the WhatsApp web application

In this chapter, we saw the simple example of sending WhatsApp messages but there are other tools such as **Twilio** WhatsApp APIs (https://www.twilio.com/whatsapp) that can be used to automate more complex workflows on WhatsApp business accounts. It can be used to provide customer care, customer service, and notifications.

Conclusion

In this chapter, we learned about different libraries to automate email-based tasks in Python. We learned the basics of SMTP and Gmail APIs for sending emails. We also looked at some APIs to automate SMS messaging and libraries to automate WhatsApp messenger.

In the next chapter, we will look at ways to automate different applications on your computer using the **Graphical User Interface**. This will allow you to automate a wide range of applications and allow you to control keyboard and mouse actions through a Python program.

Further reading

There are a lot of online resources to help you learn more about emails and messenger applications automation with Python. The following table lists some of the best resources to further improve your learning to build more complex email and messenger application automations:

Resource name	Link
Sending emails with Python	https://realpython.com/python-send-email/
Programming reference for the Win32 API	**https://docs.microsoft.com/en-us/windows/win32/api/**
How to automate mass SMS, push, and chat notifications	https://www.twilio.com/learn/notifications/automate-mass-sms-push-and-chat-notifications
Build workflow automation	https://www.twilio.com/docs/sms/tutorials/workflow-automation
How to send a WhatsApp message in 30 seconds with Python	https://www.twilio.com/blog/send-whatsapp-message-30-seconds-python
Gmail API Python quick start	**https://developers.google.com/gmail/api/quickstart/python**

Table 7.1: *Resources on email and messenger automation in Python*

Questions

1. What is SMTP?
2. How can you automate the sending of emails?
3. What are different Python libraries for working WhatsApp application?
4. How can you send a text message using Python?

CHAPTER 8

GUI – Keyboard and Mouse Automation

Introduction

In this chapter, we would learn to automate the **Graphical User Interface** (**GUI**) by controlling the keyboard and mouse actions. We will use the Python library `PyAutoGUI` which works with Windows, Mac, and Linux and provides automations for GUI elements within the application.

Structure

In this chapter, we will cover the following topics:

- Introduction to the PyAutoGUI module
- Controlling mouse actions
- Controlling keyboard actions
- Automation using screenshots

Objectives

After studying this chapter, you will be able to automate all kinds of applications you use on your work computer. We will go through the examples on a Windows machine but the automations would work even if you have a Mac or Linux computer.

Introduction to the PyAutoGUI module

We will use the **PyAutoGUI** module that allows your Python scripts to control the mouse and keyboard to automate computer applications. PyAutoGUI also works across operating systems such as Windows, macOS, and Linux.

PyAutoGUI provides features stated as follows:

- Controlling the mouse movement and clicking on the window (user-interface) of the required application.
- Sending keyboard letters to applications (for example, to fill out data).
- Take screenshots and search for buttons and other controls using the image.
- Display message boxes.
- Locate an applications window and resize the application (works only on Windows operating system).

Sometimes, you might want to stop the automation running with PyAutoGUI due to an error in your code. The PyAutoGUI has a safety feature called **FailSafe** that is enabled by default. If you move your mouse in any of the four corners of your monitor, and if the PyAutoGUI function is running, it will raise a `pyautogui.FailSafeException`. There is also a `0.1` second delay after calling every PyAutoGUI function so that you have the time to slam the mouse in the corner to trigger the fail safe exception.

To install `pyautogui`, use the `mu` package manager, type `pyautogui`, and click on `OK` as shown in the following figure:

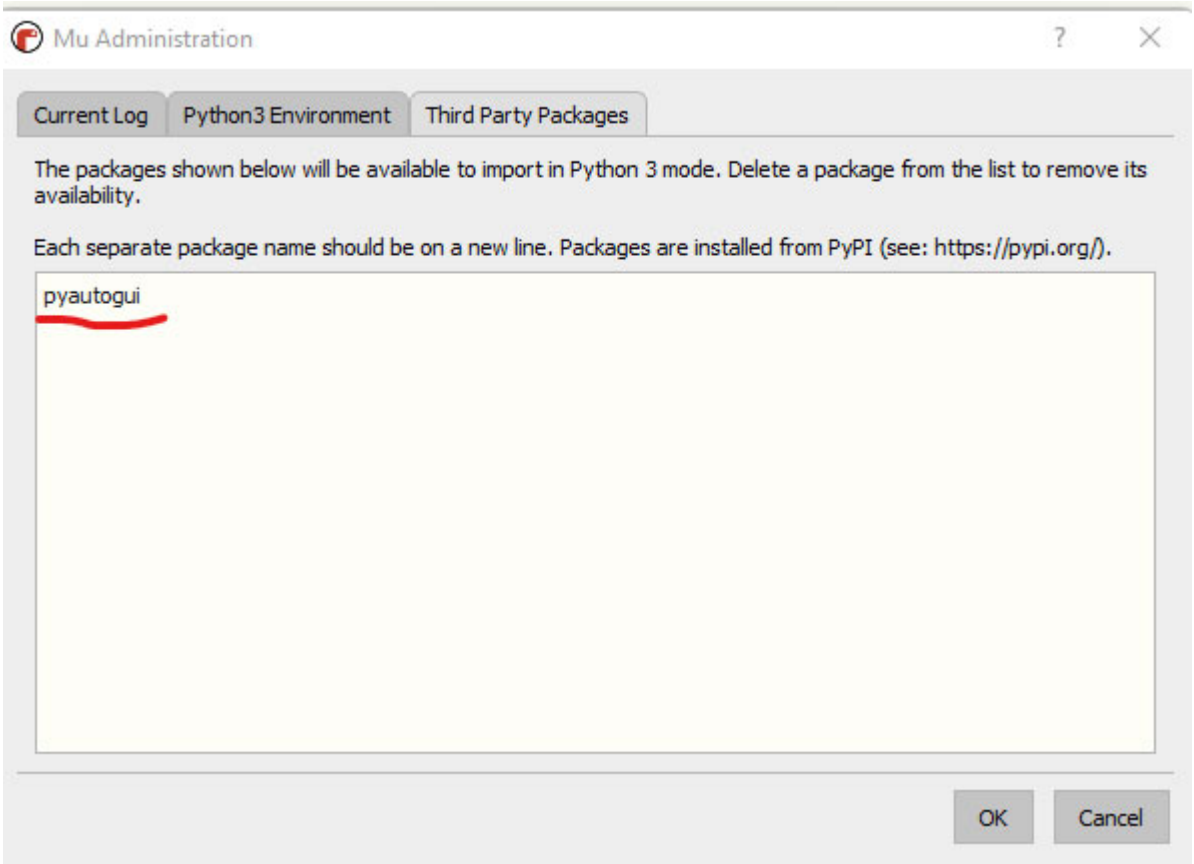


Figure 8.1: *Mu package manager*

PyAutoGUI has functions to help you get the screen coordinates and screen resolution. The location at the top-left corner of the screen is at coordinates **0, 0**. The location of the lower right-hand corner depends on your screen's resolution (for example, if the screen resolution is *1920 x 1080*, then the location of the lower right corner will be *1919, 1079*).

PyAutoGUI has the `size()` function that returns the screen resolution size, the `position()` function that returns the current **X** and **Y** coordinates of the mouse cursor, and the `onScreen()` function that can check whether the **X** and **Y** coordinates are on the screen as shown in *Figure 8.2*. The **x** and **y** coordinates that we see in this case are showing the position of the *Run* button that we clicked when we ran the code:

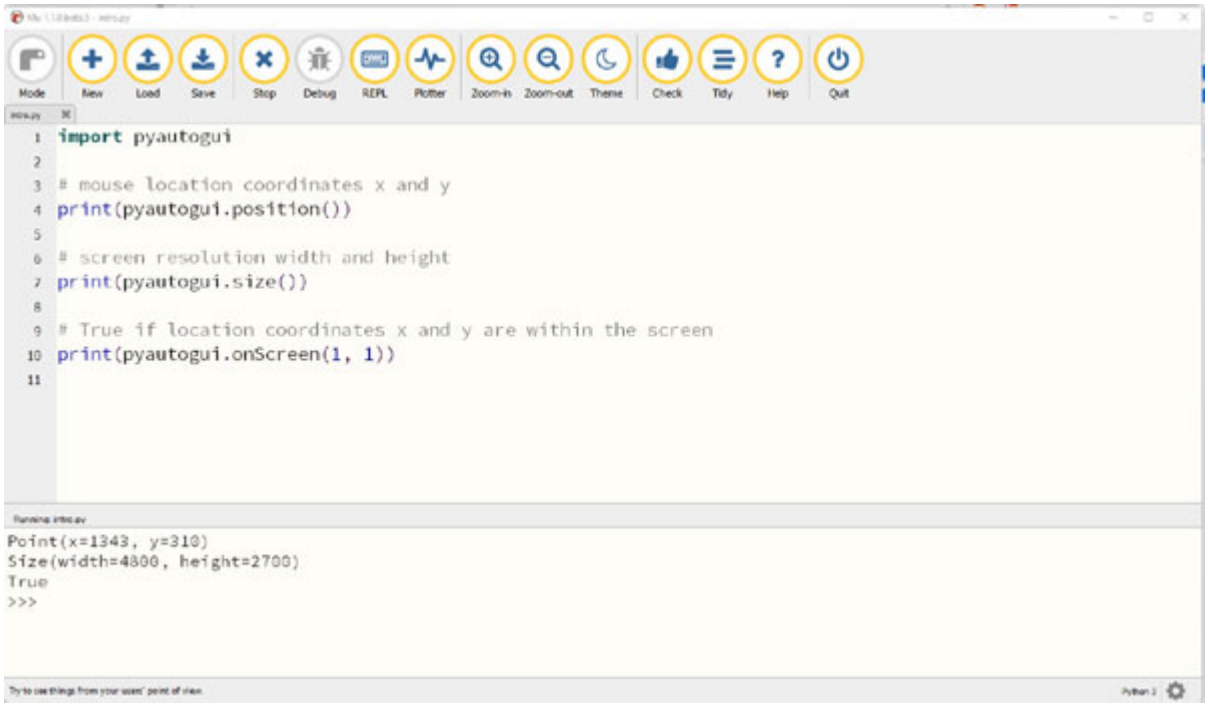


Figure 8.2: *Using pyautogui basic functions*

In the next section, we will look at controlling mouse actions with the PyAutoGUI library. In particular, we will look at how we can use the library to automatically click on the *application and use the mouse drag function to drag* the *mouse pointer on applications.*

Controlling mouse actions

PyAutoGUI provides various functions to control different types of mouse actions. The most commonly used mouse automation functions in PyAutoGUI are as follows:

- **`moveTo()`**: The **`moveTo()`** function moves the mouse cursor to the **X** and **Y** integer coordinates passed to it. For example, **`pyautogui.moveTo(200, 400)`** will move the mouse cursor to **X** coordinates at **`200`** and **Y** coordinates at **`400`**. The mouse pointer will immediately move to these new coordinates.
    - To add a delay, you can pass a third parameter for the delay (in seconds).
    - The **`move()`** function moves the mouse to position relative to the current position.

- **`dragTo()`** and **`drag()`**: The **`dragTo()`** and **`drag()`** take **`X`** and **`Y`** integer coordinates similar to the **`moveTo()`** and **`move()`** functions, but it drags the mouse pointer instead of moving the mouse pointer. They can also take a **`button`** keyword which can be set to *left*, *middle*, and *right* for specifyi the mouse button to hold down while dragging.
- **`scroll()`**: The **`scroll()`** function simulates the mouse scroll wheel by taking the argument as the integer number of *clicks* to scroll. For example, **`pyautogui.scroll(5)`** will scroll up *5 clicks* and **`pyautogui.scroll(-5)`** will scroll down *5 clicks*.
- **`click()`**: The **`click()`** function simulates a left-button mouse click (pushing the button down and releasing it up) at the mouse's current position:

  You can also specify **X** and **Y** integer coordinates to move the mouse at the location and then click on the left mouse button.

  - To specify different mouse buttons for click, you can pass arguments such as **`left`**, **`middle`**, or **`right`** on the **`button`** keyword argument. For example, **`pyautogui.click(button='right')`** will use the right-click button of the mouse.
  - To do multiple clicks, you can pass an integer to the **`click`** keyword argument. For example, **`pyautogui.click(clicks=2)`** will perform a double-click on the left mouse button.
  - There are **`doubleClick()`** and **`rightClick()`** functions as well to simulate double click and mouse right button clicks as well.

Figure 8.3 shows an example of the **`moveTo`** function to move the mouse to the specified coordinates and the **`click`** function to click at the correct position as required by the automation. When you run this code, the mouse pointer would move to the specified coordinates in the code, and then perform click, and double click actions. After this, the mouse pointer would move to coordinates specified in the **`moveTo`** function using the specified animation which in this case is **`pyautogui.easeInOutQuad`** that starts and ends fast and is slow in the middle:

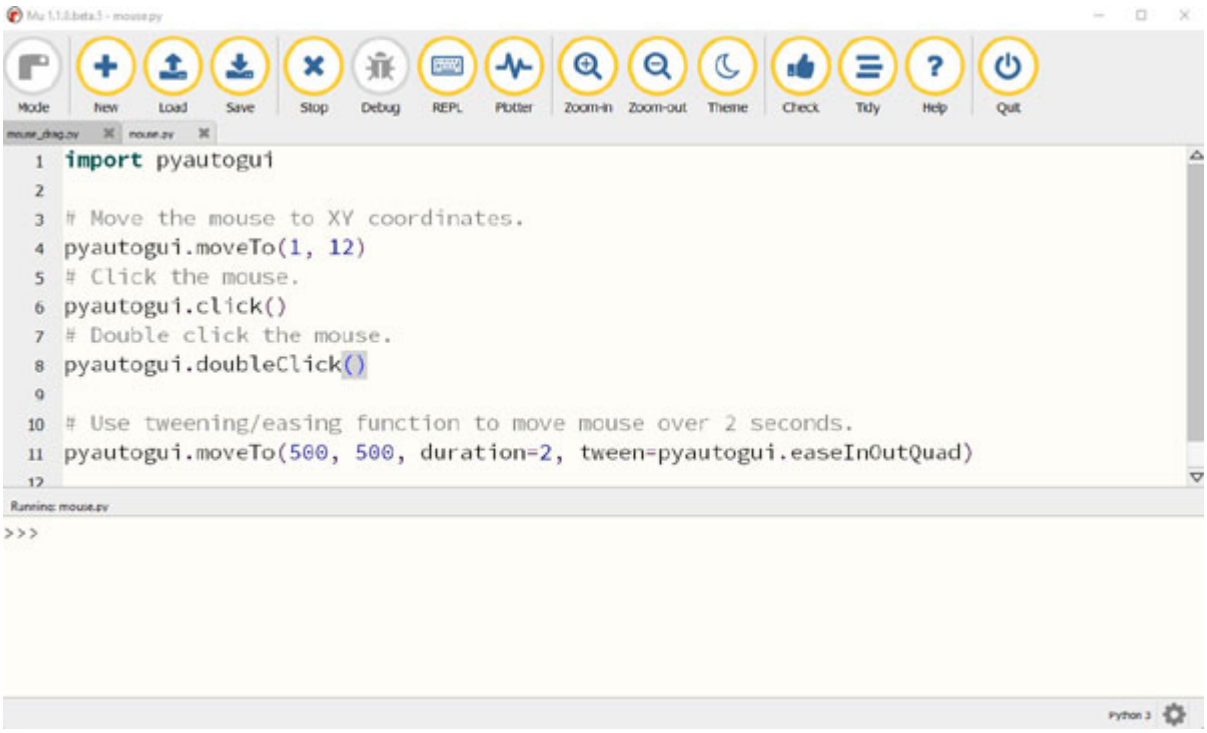


Figure 8.3: *Automating mouse actions*

You can also use the **`os.startfile`** function to launch a new program passing in as parameters the program name or the program file location. Once the program is launched, you can perform the automation on the newly launched program using the mouse automation functions. *Figure 8.4* shows an example of launching the **`mspaint`** program and using the **`drag`** function to paint on the MS paint program:

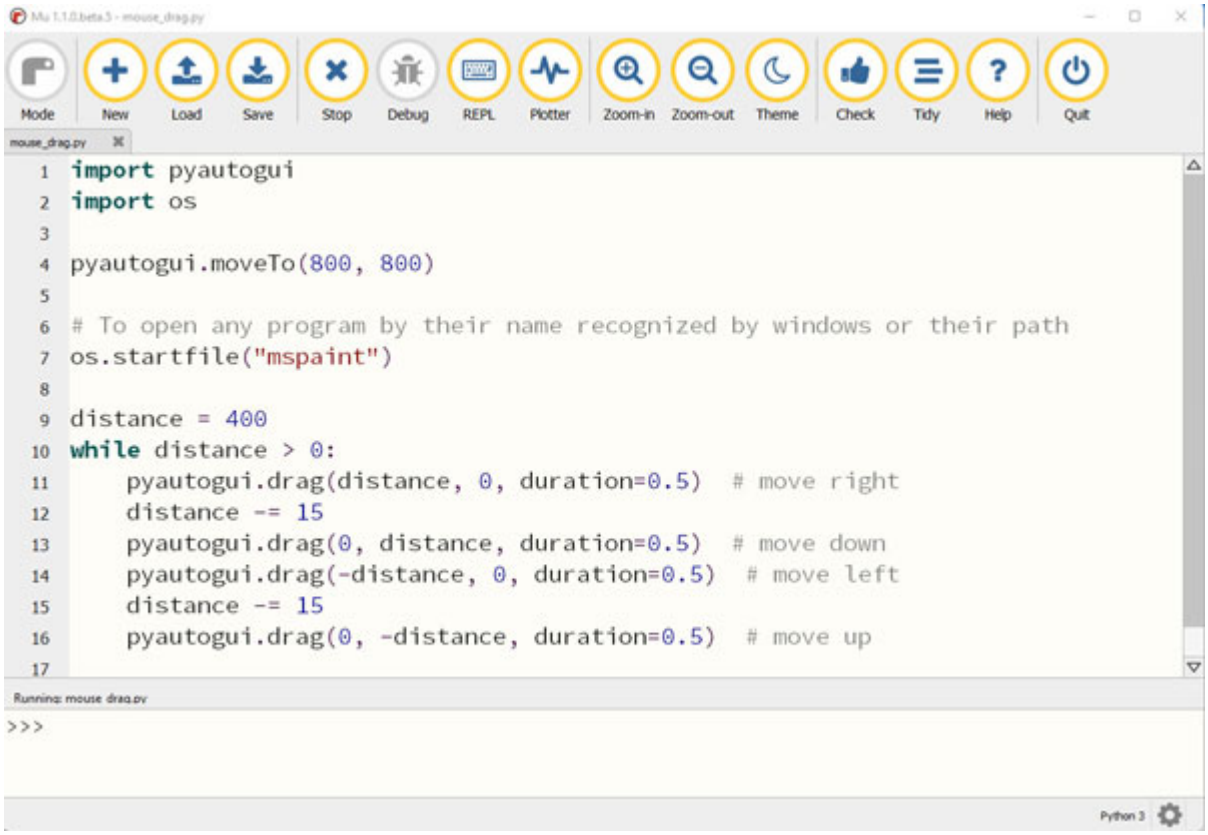


Figure 8.4: *Automating the MS Paint application*

Figure 8.5 shows the output of running the paint automation shown previously *at two different start locations and creating a square spiral diagram*:

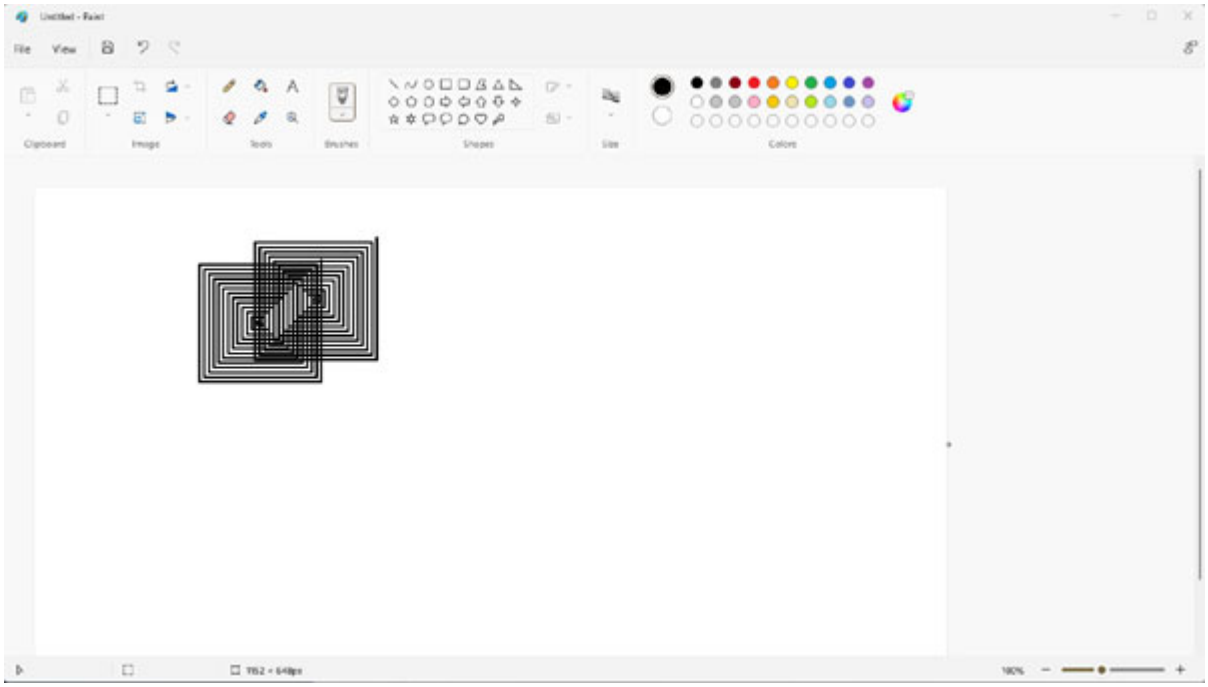


Figure 8.5: *Automating paint application*

In the next section, we will look at controlling keyboard actions with the **PyAutoGUI** library. In particular, we will look at how we can use the library to automatically type on different applications and use the hot keys functions to send commands such as **copy** and **paste**.

Controlling keyboard actions

PyAutoGUI provides various functions to control different types of keyboard actions. The most commonly used keyboard automation functions in PyAutoGUI are stated as follows:

- **`write()`**: The **`write()`** function is the primary keyboard function that is used to type the characters in the string that is passed. To add delay between pressing each character key, an interval keyword argument is passed with the required delay. For example, **`pyautogui.write('hello from bot')`** will write the text **`hello from bot`** on the focused application.
- **`press()`**: The **`press()`** function is used to press a particular key from **`pyautogui.KEYBOARD_KEYS`** such as *Enter*, *esc*, *F1*. For example, **`pyautogui.press('enter')`** will press the *Enter* key. The **`press()`**

function calls the **keyDown()** and **keyUp()** functions that simulate pressing a key down and then releasing it up.

- **keyDown()** and **keyUp()**: **keyDown()** is used to simulate pressing a key down and **keyUp()** is used to simulate releasing the key up. For example, **pyautogui.keyDown('shift')** holds down the *Shift* key and **pyautogui.keyUp('shift')** releases the *Shift* key. You can add other key presses in between these functions to keep on holding the shift key while other keys are typed.
- **hotkey()**: The **hotkey()** function is used to make the pressing of hotkeys or keyboard shortcuts convenient. The **hotkey()** takes key strings as arguments that will be pressed down in order and then released in reverse order. For example, **pyautogui.hotkey('ctrl', 'a')** will perform the select all command by pressing *ctrl* then *a*, and then releasing *a*, then *ctrl*.

There are multiple valid **KEYBOARD_KEYS** defined at the **PyAutoGUI** documentation (**https://pyautogui.readthedocs.io/en/latest/keyboard.html**) that can be passed to the **write()**, **press()**, **keyDown()**, **keyUp()**, and **hotkey()** of the **PyAutoGUI** keyboard function. For example, for passing function keys, here are the following **KEYBOARD_KEYS**:

```
[
alt, altleft, altright, backspace, capslock, ctrl, ctrlleft,
ctrlright, delete, enter, esc, escape, insert, numlock, print,
shift, shiftleft, shiftright, tab]
```

PyAutoGUI also has an **alert()** function that can be used to display a message box when an automation has been completed. Furthermore, PyAutoGUI has the window handling function which is useful during the application automation as follows:

- **pyautogui.getWindows()**: This gets a **dict** of window titles mapped to window IDs.
- **pyautogui.getWindow(str_title_or_int_id)**: This gets a **Win** object that can be used to perform various operations on the selected window.
- **pyautogui.getWindowsWithTitle()**: This gets the windows with the title supplied in the argument.

- **`win.move(x, y)`**: This moves the window to **X** and **Y** location.
- **`win.resize(width, height)`**: This resizes the window to the given **`width`** and **`height`**.
- **`win.maximize()`**: This maximizes the window.
- **`win.minimize()`**: This minimizes the window.
- **`win.restore()`**: This restores the window.
- **`win.close()`**: This closes the window.
- **`win.position()`**: This gets the **X** and **Y** location of the top-left corner of the window.

Figure 8.6 shows an example of using the **`pyautogui.getWindowsWithTitle()`** to get the current Mu code window and minimizing it. The **`keyboard.py`** specified in this function is the name of the **`Mu`** file. We then write **`I am automation bot`** on the Notepad application which was already opened and then use the hotkey function to select, copy, and paste the text. After that, we use the **`alert()`** function to alert the user that the automation is completed:

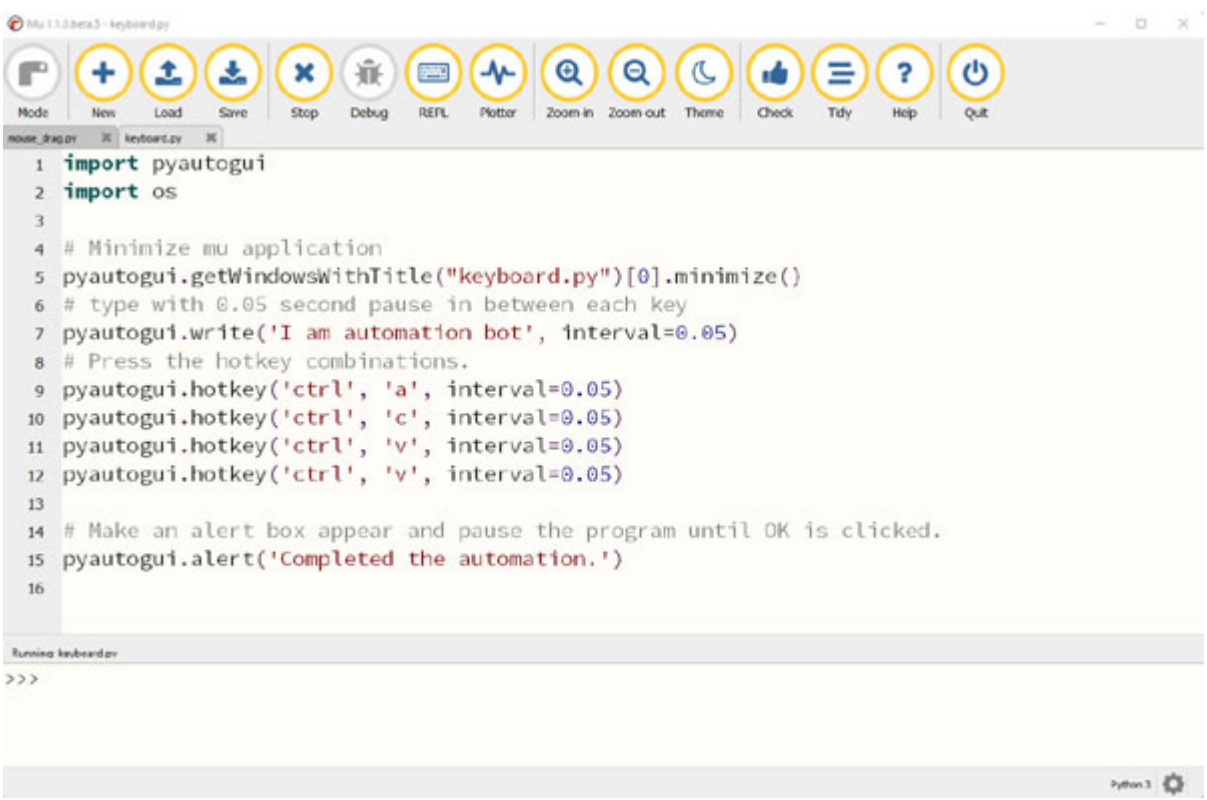


Figure 8.6: *Automating keyboard actions*

Figure 8.7 shows the output of running the keyboard automation on the Notepad application:

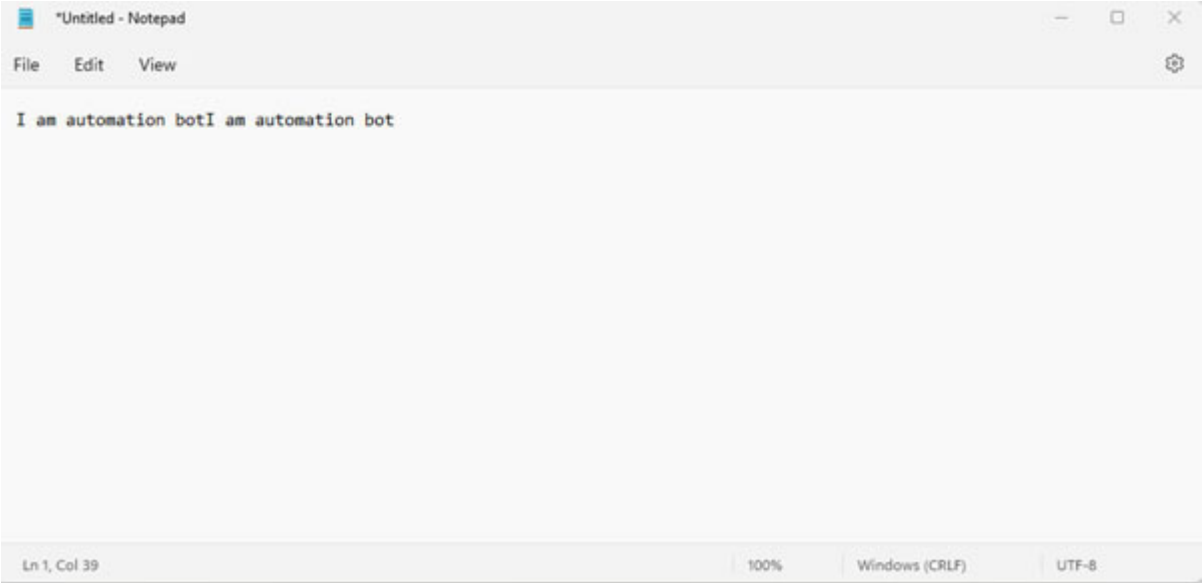


Figure 8.7: *Typing stuff on Notepad using automation*

In the next section, we will look at identifying windows and buttons using the screenshots identification tools in the PyAutoGUI library. In particular, we will look at how we can use the library to identify different buttons, areas, and windows where we want our automation to work on.

Automation using screenshots

PyAutoGUI provides functions to identify windows and buttons using the screenshots. PyAutoGUI has the functionality to take screenshots, save them to files, and locate images within the screen. You can also use the **Snipping Tool** in windows to take a snapshot of the required button or windows, and save it to be used by the automation program.

The most commonly used screenshot-based functions in PyAutoGUI are as follows:

- **`screenshot()`**: The **`screenshot()`** function returns an **`Image`** object of the captured screen. You can also pass file path to save the screenshot to a file. For example, **`pyautogui.screenshot('automation_screenshot.png')`** will capture the full screen and save it to the current Python folder with the

filename `automation_screenshot`. You can also provide a `region` keyword argument to capture the subset of the screen by passing the four-integer tuple of the `left`, `top`, `width`, and `height` of the region to be captured.

- `locate` functions: There are three main locate functions that are used to find the location of the captured image on the screen. They are mentioned as follows:
  - `locateOnScreen(image, grayscale=False)`: This function returns the **left**, **top**, **width**, and **height** coordinates of the first found instance of the image on the screen. It raises `ImageNotFoundException` if not found on the screen.
  - `locateCenterOnScreen(image, grayscale=False)`: This function returns the **X** and **Y** coordinates of the center of the first found instance of the image on the screen. It raises `ImageNotFoundException` if not found on the screen.
  - `locateAllOnScreen(image, grayscale=False)`: This function returns a tuple of **left**, **top**, **width**, and **height** coordinates for the images found on the screen.

Figure 8.8 shows an example of using the `pyautogui.locateOnScreen()` to get the current location of the load image on the screen. Once we get the image location, we use the `pyautogui.center()` function to get the center of the image location and pass it to **X** and **Y** coordinate variables. We can use these **X** and **Y** coordinate variables and call the `pyautogui.click()` to click on the `load` button. The `loadImage.png` file used in this code should reside in the same folder as the Python code otherwise you will need to specify the full file path of the image file. Also, the image file should contain the image of the location where you want the automation to work on. Further documentation on using the image location's function is available on the PyAutoGUI documentation page (**https://pyautogui.readthedocs.io/en/latest/**):

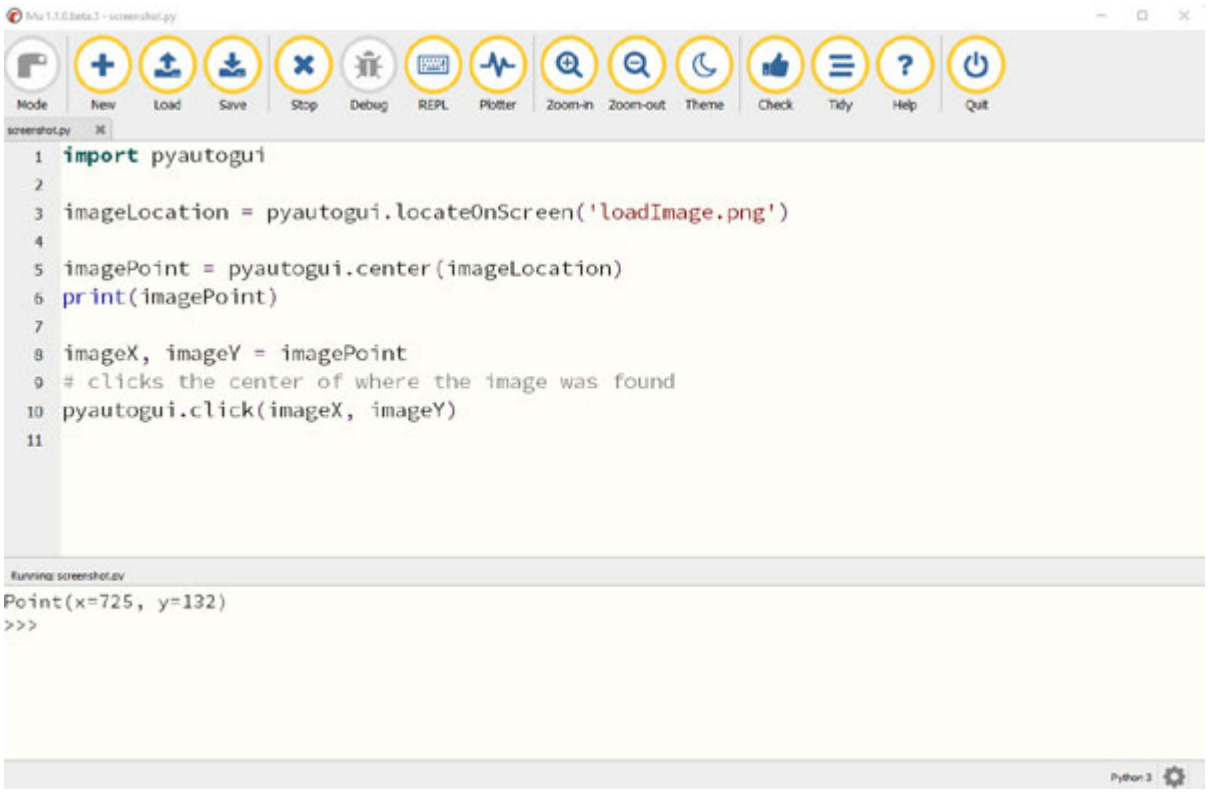


Figure 8.8: *Automating a click using the image screenshot*

With PyAutoGUI, you can automate a wide variety of applications in Windows, Mac, and Linux machines. If you want to just automate applications in Windows, then **`pywinauto`** is another library that provides functions to automate the Microsoft Windows GUI. It allows you to send mouse and keyboard actions to windows dialogs and controls, and it also has support for more complex actions like getting text data from different applications. To learn more about the **`pywinauto`**, see the **`pywinauto`** documentation available online (**https://pywinauto.readthedocs.io/en/latest/**).

Conclusion

In this chapter, we learned about the Python library PyAutoGUI to control mouse and keyboard actions and automation applications using the **Graphical User Interface** (**GUI**). We learned the functions available to perform click operations, type operations, and identifying controls of applications using images.

In the next chapter, we will look at image fundamentals and the `pillow` Python library for manipulating images. We would also look at the `Tesseract` library which can be used to extract the text within images and scanned documents.

Further reading

There are a lot of online resources to help you learn more about GUI, keyboard, and mouse automation with Python. The following table lists some of the best resources to further improve your learning on GUI automation in Python:

Resource Name	Link
PyAutoGUI's documentation	https://pyautogui.readthedocs.io/en/latest/
Use Snipping Tool to capture screenshots	**https://support.microsoft.com/en-us/windows/use-snipping-tool-to-capture-screenshots-00246869-1843-655f-f220-97299b865f6b**
What is pywinauto?	https://pywinauto.readthedocs.io/en/latest/
PyautoGUI: Three Great Uses	**https://www.youtube.com/watch?v=o0OySmkZo8g**

Table 8.1: *Resources on GUI automation in Python*

Questions

1. What is the use of PyAutoGUI module?
2. What are the different types of mouse actions in PyAutoGUI module?
3. How do you simulate keyboard actions in Python?
4. How do you run automations using screenshots?

CHAPTER 9

Image Based Automations

Introduction

In this chapter, we will look at computer image fundamentals and the **Pillow** Python library for manipulating images. We would also look at the OCR libraries to extract text within images and scanned documents.

Structure

In this chapter, we will cover the following topics:

- Computer image fundamentals
- Pillow for image manipulation
- Extracting text from images using OCR

Objectives

After studying this chapter, you will be able to manipulate and modify computer images, and extract text from scanned documents and images. You will also learn about **Optical Character Recognition** (**OCR**) which is a technique used to extract text from saved images.

Computer image fundamentals

A computer image consists of a **picture element** (**pixel**) which is the smallest component of a computer image. When an image is manipulated by the computer, the pixel is a dot of a single color and the image is made up of pixels on a rectangular grid. The resolution of the image is the number of points in the grid; for example, *1920x1080* means that the image is *1920* pixels wide by *1080* pixels high.

There are a large number of formats for storing digital images. Most of the image formats were developed to be used by particular programs but few of

them have become image format standards and can be used across a variety of applications. These image formats are also called **Bitmap** formats where Bitmap is the memory organization to map pixels for storing images. The following are the most commonly used image formats by different applications:

- **CompuServe Graphics Interchange Format** (**GIF**): This image format is used for file interchange and it has a good compression algorithm built into the format.
- **Tagged Image File Format** (**TIF/TIFF**): This is a flexible image format with a number of compression algorithms.
- **Joint Photographic Experts Group** (**JPG/JPEG**): This is an image format which is developed as a standard by ISO and CCITT. It has a very good compression algorithm for continuous-tone images. For most images, this format can compress and reduce the size of the images to be *20 times* smaller. This format does not support transparency or transparent backgrounds.
- **Portable Network Graphics** (**PNG**): It is one of the most used image formats on the Internet. It can display transparent backgrounds and was created to replace the GIF format. It is an open format with no copyright limitations and it compresses images without any loss of image data (also known as **lossless compression** which involves reducing the size of the file without any loss of quality). This format supports transparency and transparent backgrounds.

In the next section, we will look at the **Pillow** image library that can be used to manipulate images and modify image properties.

Pillow for image manipulation

`Pillow` is a Python library used to manipulate images and it is based on the **Python Imaging Library** (**PIL**). The `Pillow` library adds image processing capabilities and provides extensive support for converting image files from one format to another.

To install the `Pillow` library, use the `mu` package manager, type `pillow`, and click on `OK` as shown in *Figure 9.1*:

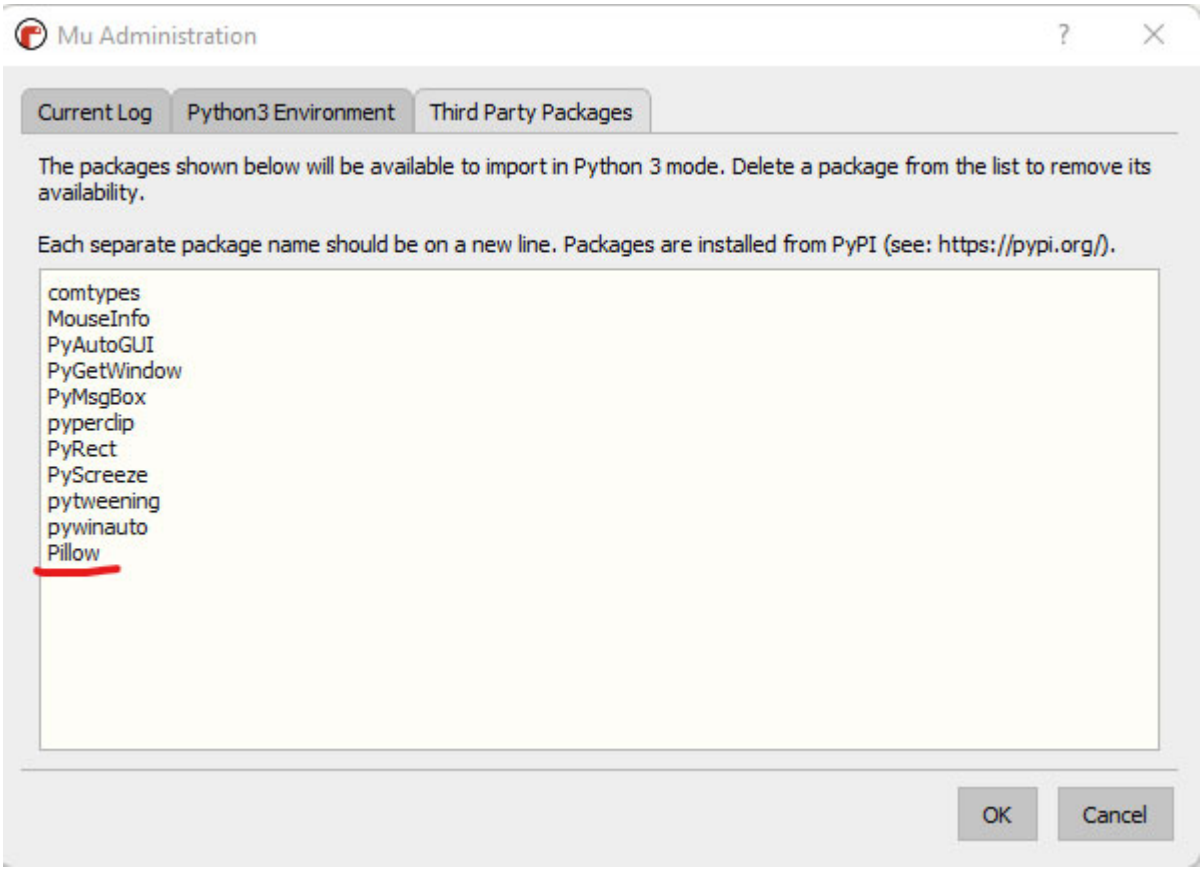


Figure 9.1: *Mu package manager*

To import the **`pillow`** image library, use the statement **`from PIL import Image`**. **`Image.open("loadImage.png")`**. Once the image is loaded with the **`pillow`** module, you can get the image details such as format, size, and mode as shown in *Figure 9.2*:

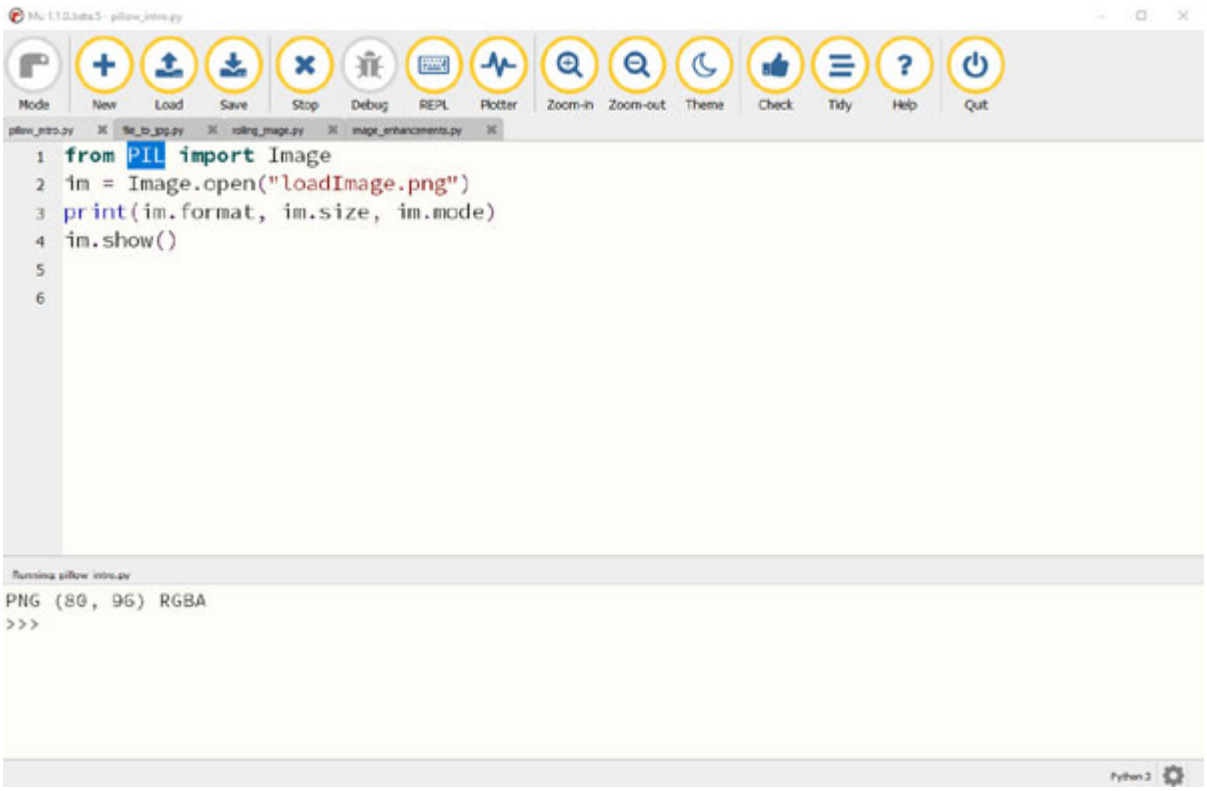


Figure 9.2: *Image properties using pillow*

The **`pillow`** library can be used to convert images between different image formats. For example, to convert an image from PNG to JPG, you will first need to change the color band to **Red, Blue, and Green** (**RGB**) from **Red, Green, Blue, and Alpha** (**RGBA**)- **Alpha** is transparency), and then save the image with extension as **`.jpg`**. Pillow will convert the image as per the given file extension and save the image in the new format as shown in *Figure 9.3*:

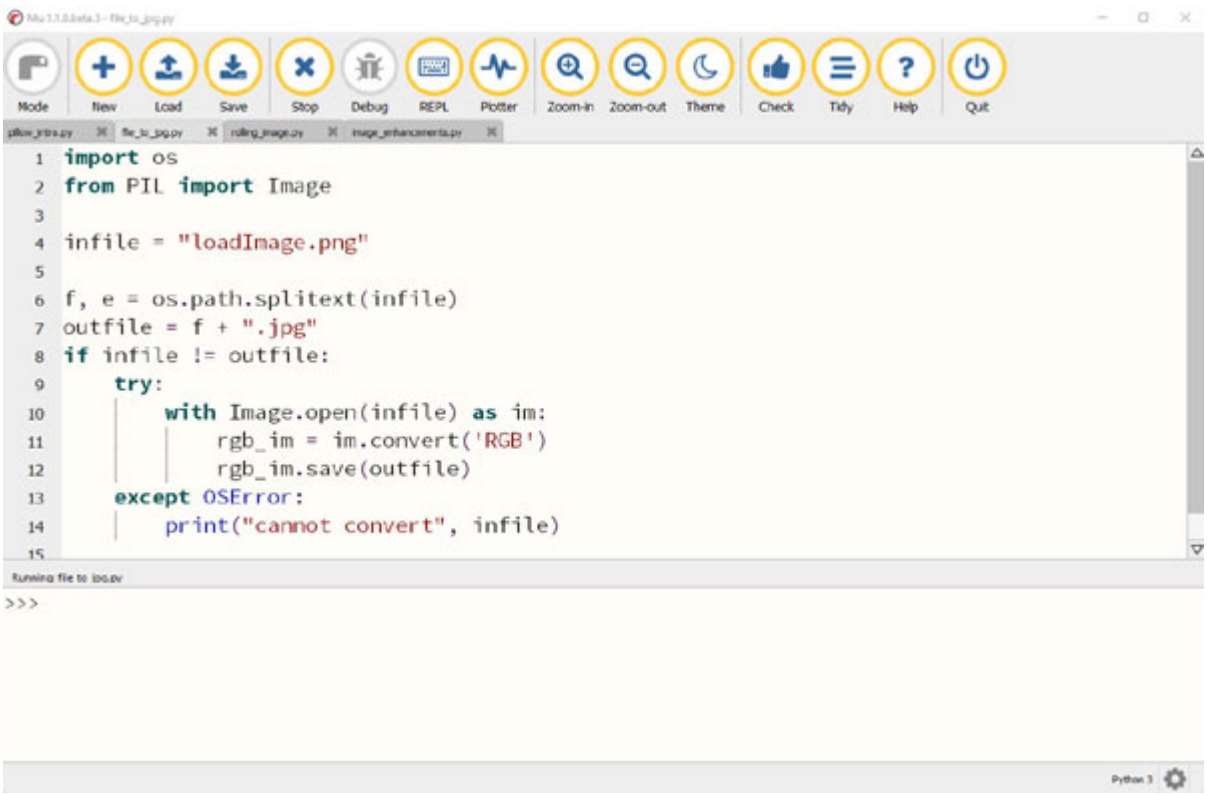


Figure 9.3: *Converting an image to JPG format*

The **`Pillow`** library has an **`ImageEnhance`** module that has a number of classes that can be used for image enhancement. The main image enhancement classes are as follows:

- **`ImageEnhance.Color`**: This class adjusts the color balance of the image. You can pass a color enhance factor to the **`enhance`** function of this class where a factor of **`1.0`** is the *original* image color and a factor of **`0.0`** is the *black and white* image.
- **`ImageEnhance.Contrast`**: This class adjusts the contrast of the image. You can pass a contrast enhance factor to the **`enhance`** function of this class where a factor of **`1.0`** is the *original* image color and a factor of **`0.0`** is a *solid gray* image.
- **`ImageEnhance.Brightness`**: This class adjusts the brightness of the image. You can pass a brightness enhance factor to the **`enhance`** function of this class where a factor of **`1.0`** is the *original* image color and a factor of **`0.0`** is a *black* image.
- **`ImageEnhance.Sharpness`**: This class adjusts the sharpness of the image. You can pass a sharpness enhance factor to the **`enhance`**

function of this class where a factor of `1.0` is the *original* image color, a factor of `0.0` is a *blurred* image, and a factor higher than `1.0` gives a *sharpened* image.

Figure 9.4 shows an example of using the **`ImageEnhance`** module to increase the sharpness of the image:

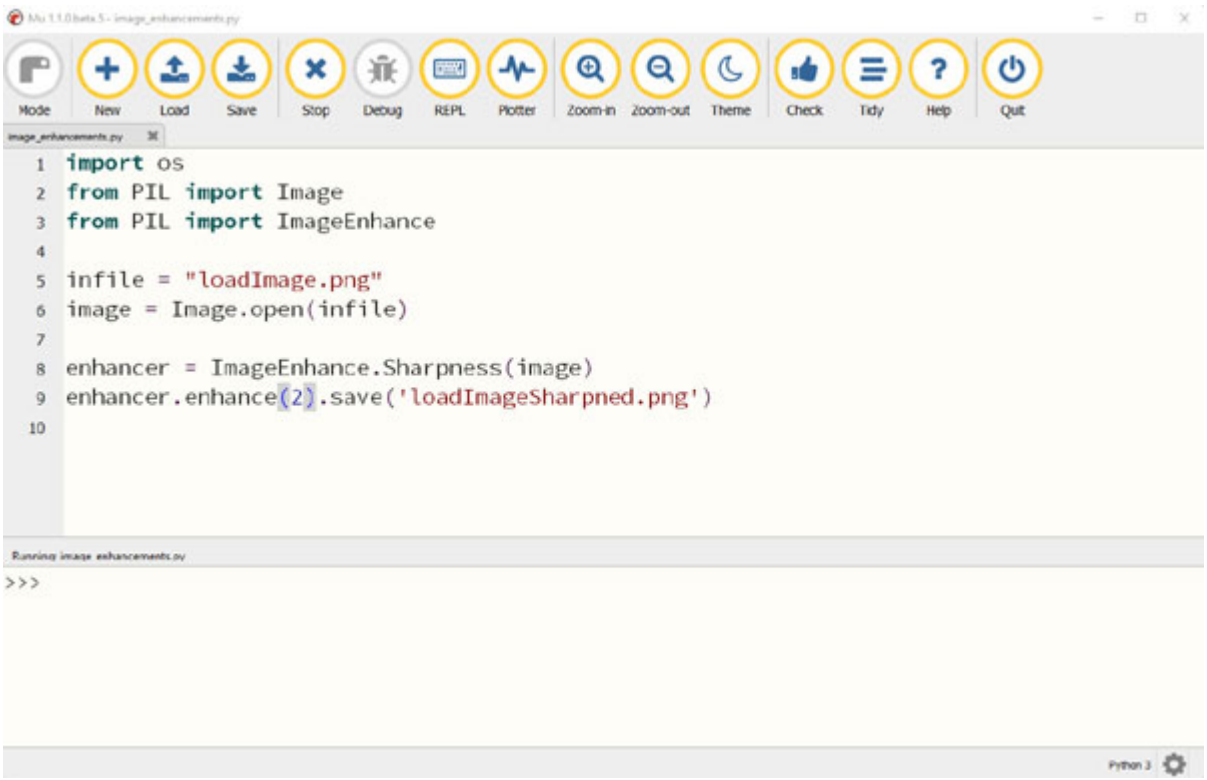


Figure 9.4: *Increasing the sharpness of the image*

After calling the **`ImageEnhance.Sharpness`** enhance function, a new sharpened image is generated. *Figure 9.5* shows the original image on the left-hand side and a sharpened image on the right-hand side:

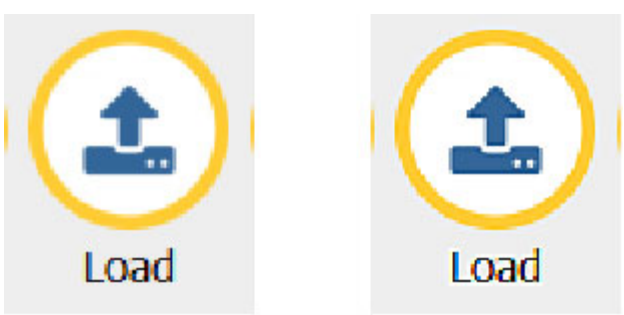


Figure 9.5: *Original image (left) and sharpened image (right)*

In the next section, we will look at the **Optical Character Recognition** (**OCR**) library in Python to extract text from images. This technique is particularly useful when you are working with scanned documents and images.

Extracting text from images using OCR

Optical Character Recognition (**OCR**) is a technique used to extract machine-encoded text from images or handwritten documents. We will look at an open-source OCR library called `tesseract` in this chapter, but there are a variety of different OCR libraries and APIs which can be used for extracting text from images and handwritten documents.

Tesseract can be used as a command line program or can be used with `pytesseract` library which is a Python wrapper for the `tesseract` engine. `Pytesseract` requires the `tesseract` library to be installed on your computer.

To install `tesseract` on your Windows machine, download the `tesseract` installer for Windows (**https://github.com/UB-Mannheim/tesseract/wiki**) and follow the installation process as shown in *Figure 9.6*. For other operating systems, `tesseract` can be downloaded from the `tesseract` binaries page (**https://tesseract-ocr.github.io/tessdoc/Home.html#binaries**):

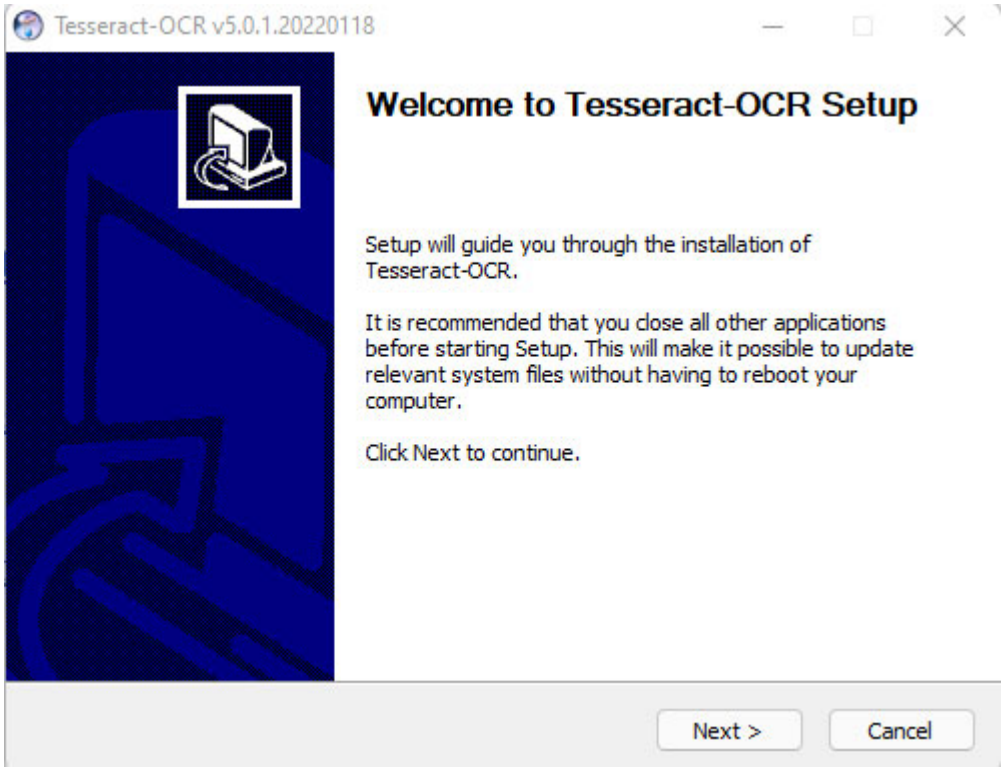


Figure 9.6: *Installing tesseract ocr*

After installing the **`tesseract`** library, install the **`pytesseract`** library by using the **`mu`** package as shown in the following *Figure 9.7*:

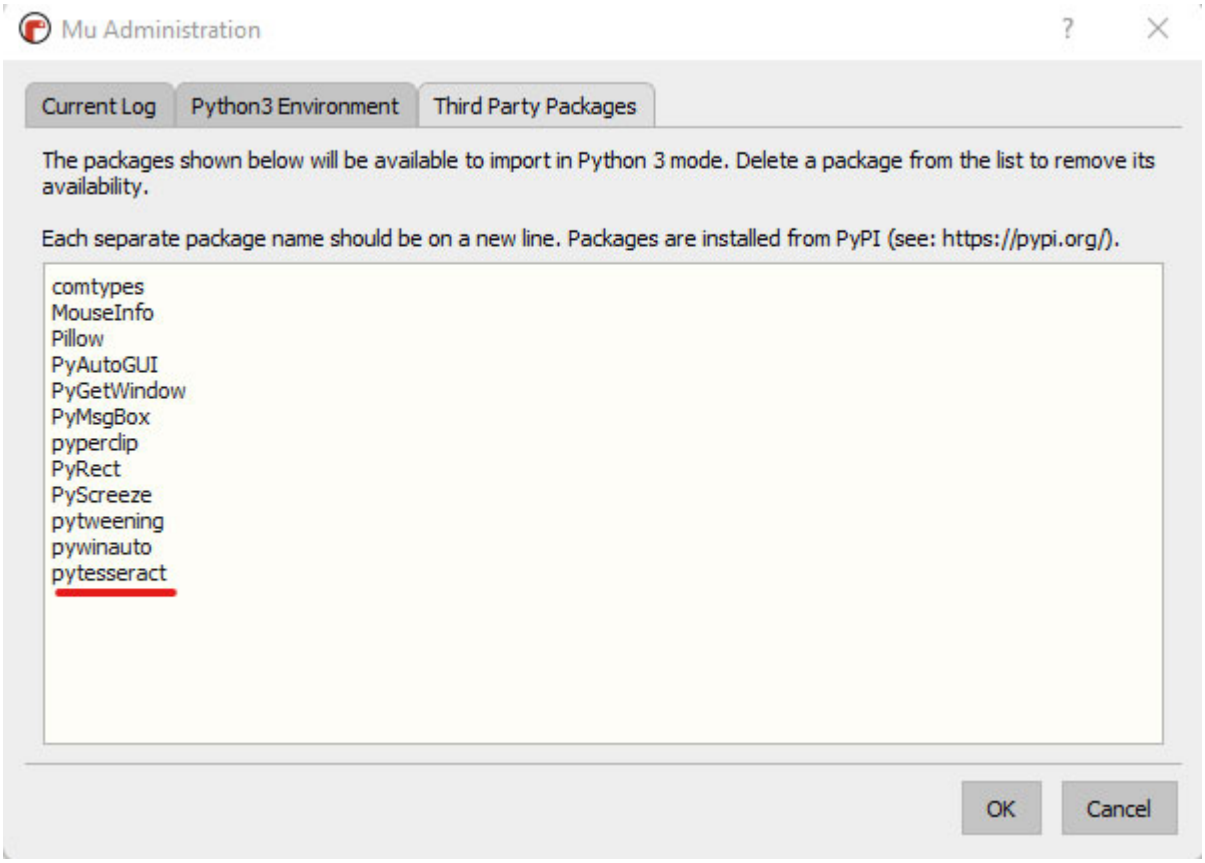


Figure 9.7: *Mu package manager*

To extract text from an image using the **`pytesseract`** library, you can use the **`image_to_string()`** function as shown in *Figure 9.8*. You will need to supply the **`tesseract`** path to the **`pytesseract.tesseract_cmd`** variable (on Windows, this path is generally, **`C:\Program Files\Tesseract-OCR`**):

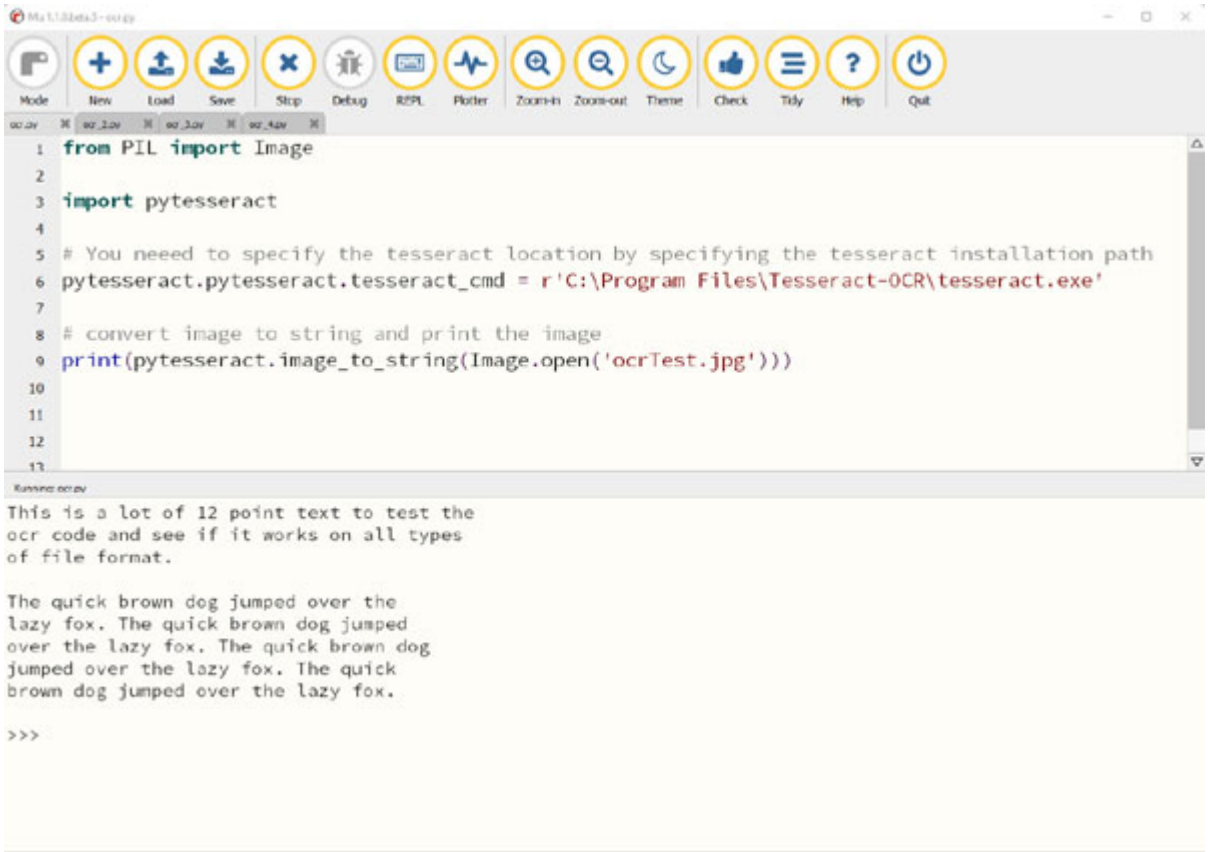


Figure 9.8: *Image to text using tesseract*

The **`Pytesseract`** library accepts different arguments for the **`image_to_data`** or **`image_to_string`** function. The following is the function signature:

```
image_to_data(image, lang=None, config='', timeout=0)
```

The different parameters accepted by this function are described below:

- **`image`**: This can be an object (**PIL** image/**NumPy** array) or file path of the image to be processed by **Tesseract**.
- **`lang`**: This is a language string with the default value of **`eng`** if no language string is specified. You can also pass in multiple language strings as a parameter; for example, **`eng+fra`** for English and French. Before passing the **`lang`** string, make sure you have downloaded the correct **`tessdata`** for the desired language (**https://github.com/tesseract-ocr/tessdata**).
- **`config`**: This passes in custom configuration flags such as page segmentation modes and OCR engine modes

(**https://manpages.ubuntu.com/manpages/bionic/man1/tesseract.1.html**).

- **`timeout`**: This passes in a duration in seconds to timeout the OCR processing engine.

When you pass in a timeout argument, **`pytesseract`** raises **`RuntimeError`** if the duration of OCR processing is taking longer. This can be handled within the **`try`** except statement as shown in *Figure 9.9*:

```
from PIL import Image
import pytesseract

pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-OCR\tesseract.exe'

# Timeout/terminate the OCR after a period of time
try:
    # Timeout after 0.1 second
    print(pytesseract.image_to_string('ocrTest.jpg', timeout=0.1))
except RuntimeError as timeout_error:
    # Tesseract processing is terminated
    print("TIMED OUT")
    pass

TIMED OUT
>>>
```

Figure 9.9: *Adding timeout for longer image conversion processes*

The **`Pytesseract`** library provides a variety of functions for the **`tesseract`** library, which are as follows:

- **`get_languages`**: This gets all supported languages by the **`Tesseract`** library.
- **`get_tesseract_version`**: This gets the **`Tesseract`** version installed.
- **`image_to_boxes`**: This gets the box boundaries and returns the containing characters within these box boundaries.

- **`image_to_osd`**: This gets the information about the script detection and orientation.
- **`image_to_alto_xml`**: This gets the result in the form of the **`Tesseract`**'s **ALTO XML** format.
- **`run_and_get_output`**: This gets the raw output from the **`Tesseract`** OCR.
- **`image_to_string`**: This gets the output as a string from the **`Tesseract`** OCR.
- **`image_to_data`**: This gets the result containing the confidence, box boundaries, line, and page numbers.

Figure 9.10 shows an example of using the **`image_to_boxes`** and **`image_to_data`** functions result containing the confidence, box boundaries, line, and page numbers:

```
from PIL import Image

import pytesseract

pytesseract.pytesseract.tesseract_cmd = r'C:\Program Files\Tesseract-OCR\tesseract.exe'

# Get bounding box for the text in images
print(pytesseract.image_to_boxes(Image.open('ocrTest.jpg')))

# Get verbose data about the image including boxes, confidences, line and page numbers
print(pytesseract.image_to_data(Image.open('ocrTest.jpg')))
```

```
level page_num    block_num   par_num     line_num    word_num    left  top   width heightconf  text
1     1     0     0     0     0     0     0     640   480   -1
2     1     1     0     0     0     36    92    582   269   -1
3     1     1     1     0     0     36    92    582   92    -1
4     1     1     1     1     0     36    92    544   30    -1
5     1     1     1     1     1     36    92    60    24    96.566017   This
5     1     1     1     1     2     109   92    20    24    96.913322   is
5     1     1     1     1     3     141   98    15    18    95.939819   a
5     1     1     1     1     4     169   92    32    24    95.939819   lot
5     1     1     1     1     5     212   92    28    24    96.492249   of
5     1     1     1     1     6     251   92    31    24    96.492249   12
5     1     1     1     1     7     296   92    68    30    96.398697   point
5     1     1     1     1     8     374   93    53    23    96.270226   text
```

Figure 9.10: *Getting image text and confidence level from tesseract*

Other popular OCR libraries are **Filestack OCR**, **ABBYY OCR**, **Anyline OCR**, and so on. There are also Cloud ORC libraries provided by Amazon

Web Services, Microsoft Azure, and Google Cloud Platform which can provide better accuracy for certain image to text tasks.

Conclusion

In this chapter, we learned about the image fundamentals and the Pillow Python library for manipulating images. We also looked at the Tesseract library which can be used to extract text within images and scanned documents.

In the next chapter, we will look at scheduling automations using dates and timer's functions. We would also look at Python hooks which can allow us to run automations based on certain events such as receiving a new email or during the start of a new applications.

Further reading

There are a lot of online resources to help you learn more about Pillow for image manipulation and Tesseract OCR. The following *Table 9.1* lists some of the best resources to further improve your learning on Pillow and OCR libraries:

Resource name	Link
Pillow documentation	**https://pillow.readthedocs.io/en/stable/index.html**
Python-tesseract is a Python wrapper for Google's Tesseract-OCR	**https://pypi.org/project/pytesseract/**
Detect text in images	**https://cloud.google.com/vision/docs/ocr**
Tesseract OCR	**https://github.com/tesseract-ocr/tesseract**
Amazon Textract	**https://aws.amazon.com/textract/**

Table 9.1: *Resources on image automation in Python*

Questions

1. What are the functions available in Pillow module for image manipulation?
2. How can you extract text from images?

3. What is `tesseract` library?
4. How can you convert images in multiple languages to text in Python?

CHAPTER 10

Creating Time and Event - Based Automations

Introduction

In this chapter, we will look at scheduling automations using **dates** and **timers**. We will also look at external applications that can allow us to run automations based on certain events such as receiving a new email or during the start of an application.

Structure

In this chapter, we will cover the following topics:

- Scheduling automation
- Writing timer programs
- Launching programs from Python
- Using external tools for triggers

Objectives

After studying this chapter, you will be able to schedule automations at a particular time of the day. You will also be able to create workflows based on triggers and use external tools to help you to run automations with triggers and interact with web applications.

Scheduling automation

You can schedule automation in Python to run during certain times of the day or run them based on certain events. **Advanced Python Scheduler** (**APScheduler**) is a Python library that allows you to schedule your automations in Python. You can add or remove jobs on the fly, and you can

even store these jobs on the database. **APScheduler** works across operating systems and offers three main scheduling functionalities which are as follows:

- **Cron job like syntax**: Cron jobs uses Linux like cron command-line utility syntax.
- **Interval-based syntax**: This allows jobs to run on specified intervals with an optional start and end time.
- **One-off delayed execution**: This allows you execute jobs once based on your set date and time.

To install the `APScheduler` library, use the `mu` package manager, type `APScheduler`, and click on `OK` as shown in the following figure:

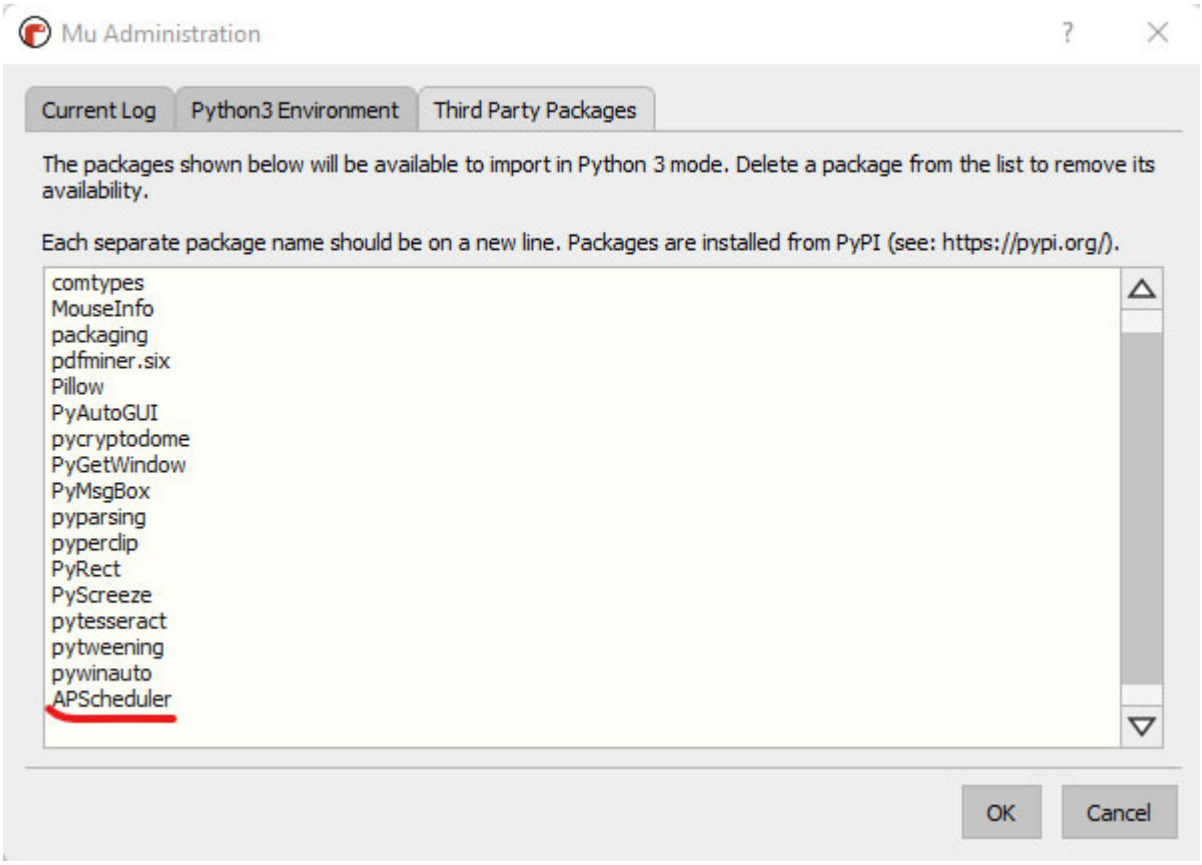


Figure 10.1: *Mu package manager*

For Windows operating systems, you can also use **Windows Task Scheduler** to schedule tasks at a certain date and time. With the task schedule, you can schedule tasks such as running the required Python

automation, sending an email message, or starting a new application. Windows tasks scheduler supports running tasks based on the following events:

- On a specific system event
- At a particular time or schedule
- When the computer is idle
- During the start of the computer
- During the user logging action

To start the task scheduler, on the *Start* menu, search or press *Windows* + *R* keys on your keyboard to launch **`Run`** and type **`taskschd.msc`**. On the task scheduler, select the **`Create Basic Task...`** option in the **`Actions`** section to create a basic task as shown in *Figure 10.2*:

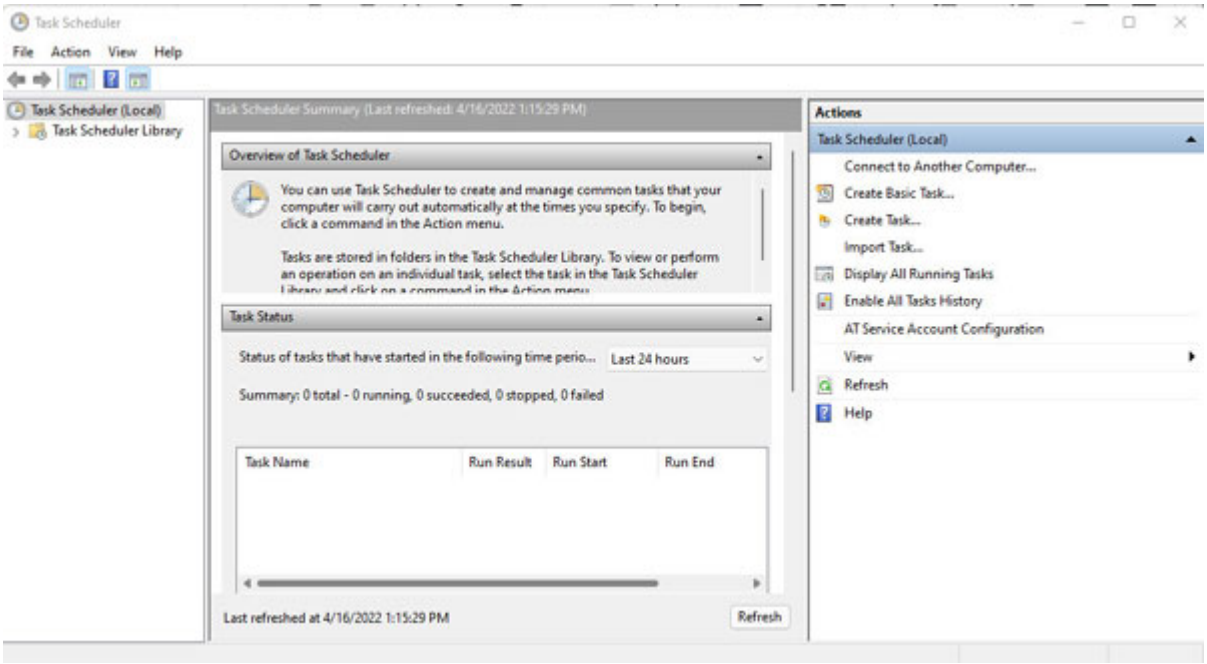


Figure 10.2: *Task scheduler home page*

Once you click on the option to create a basic task, you will see the **`Create a Basic Task`** wizard where you can add a name to your scheduled tasks and add a description for the task as shown in *Figure 10.3*:

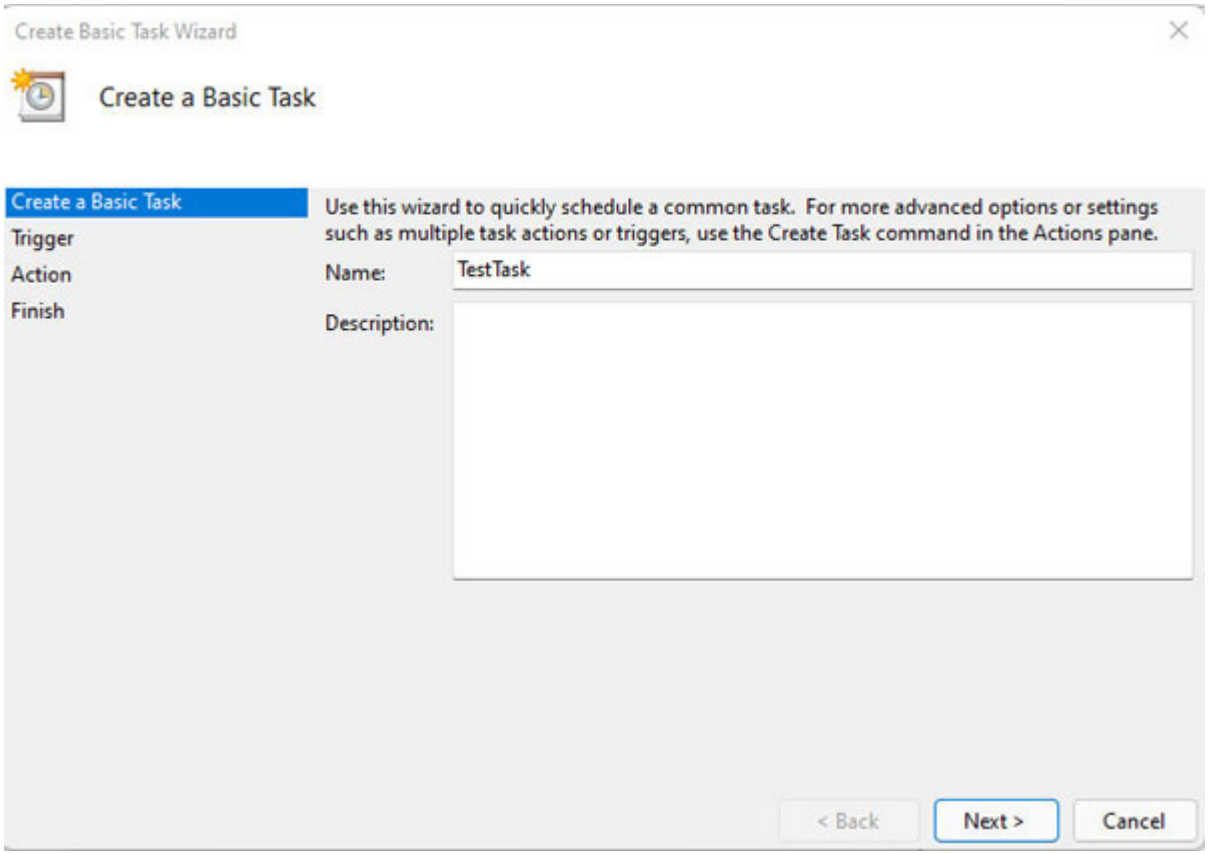


Figure 10.3: *Create a basic task wizard*

Once you click on **`Next >`**, you can select the trigger you want for your job such as running the job daily, weekly, monthly, and so on as shown in *Figure 10.4*:

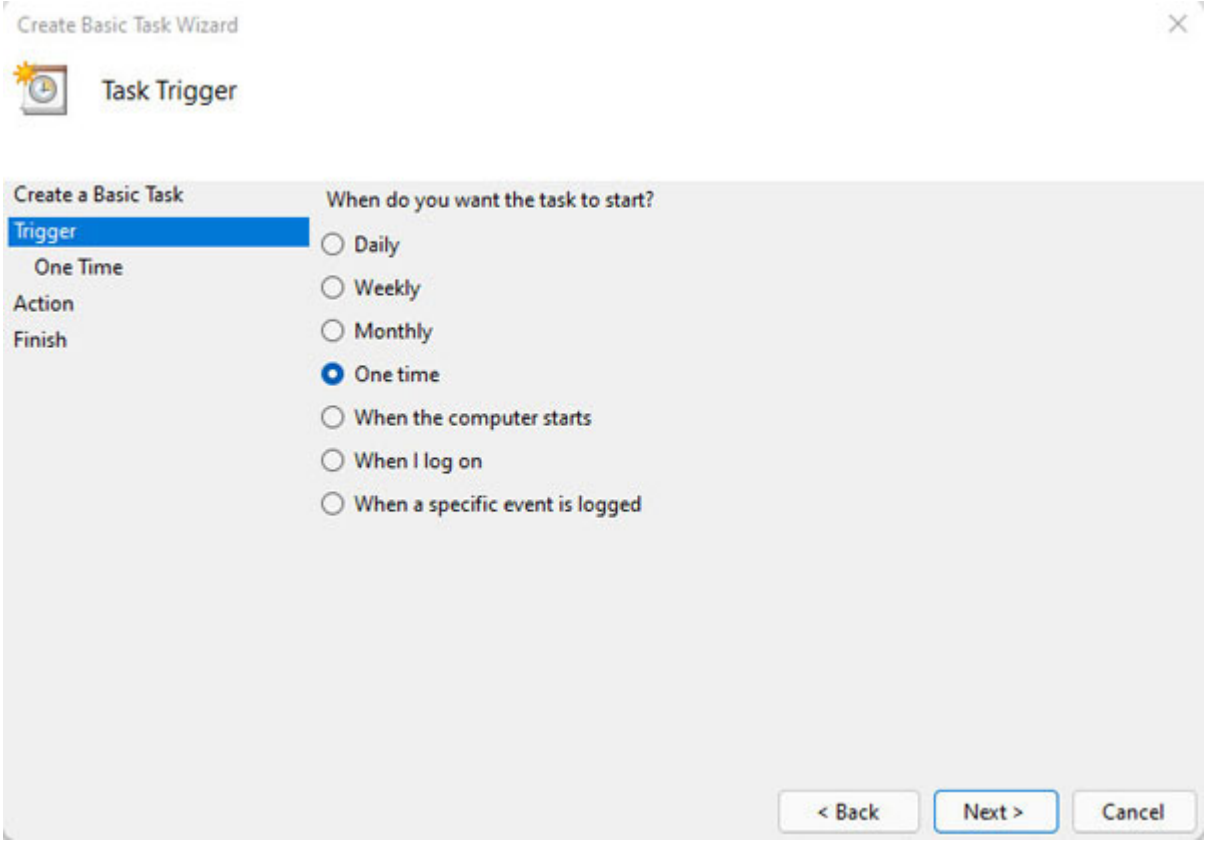


Figure 10.4: *Selecting the task trigger*

Once you click on **`Next >`**, you need to specify the trigger parameters such as when you want to schedule your task as shown in *Figure 10.5*:

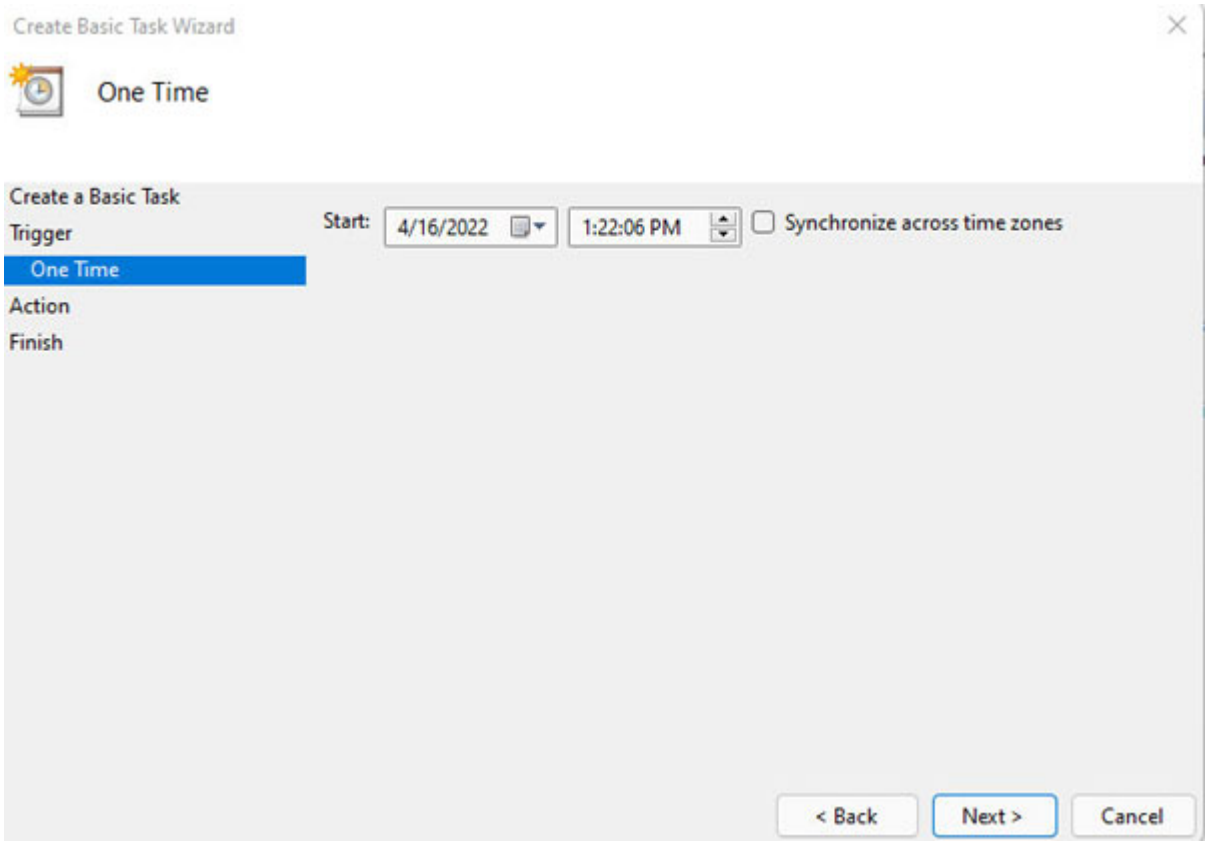


Figure 10.5: *One time schedule*

Once you click on **`Next >`**, you need to specify if you want to **`Start a program`**, **`Send an email`**, or **`Display a message`** as shown in *Figure 10.6*:

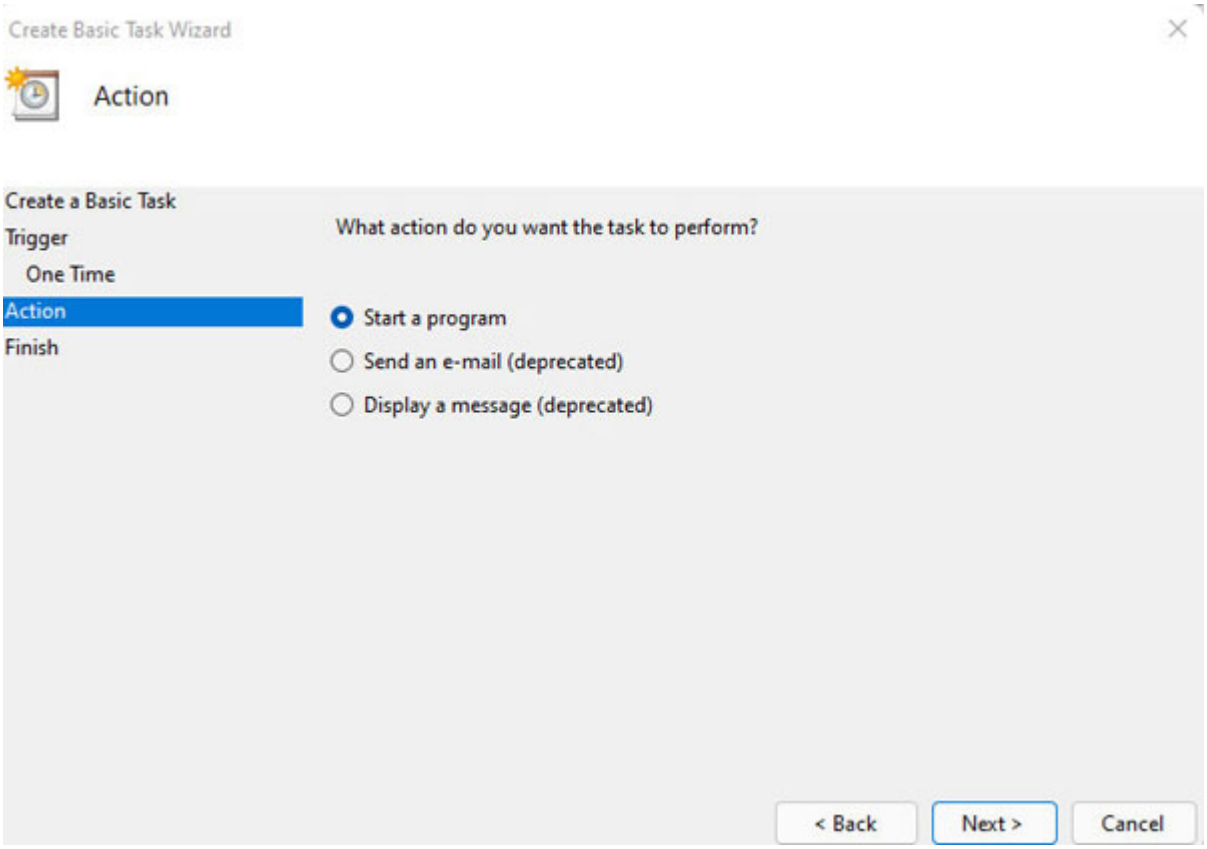


Figure 10.6: *Task manager actions options*

In this case, we will select **`Start a program`** and click on **`Next >`**, we will specify the Python automation script path as shown in *Figure 10.7*:

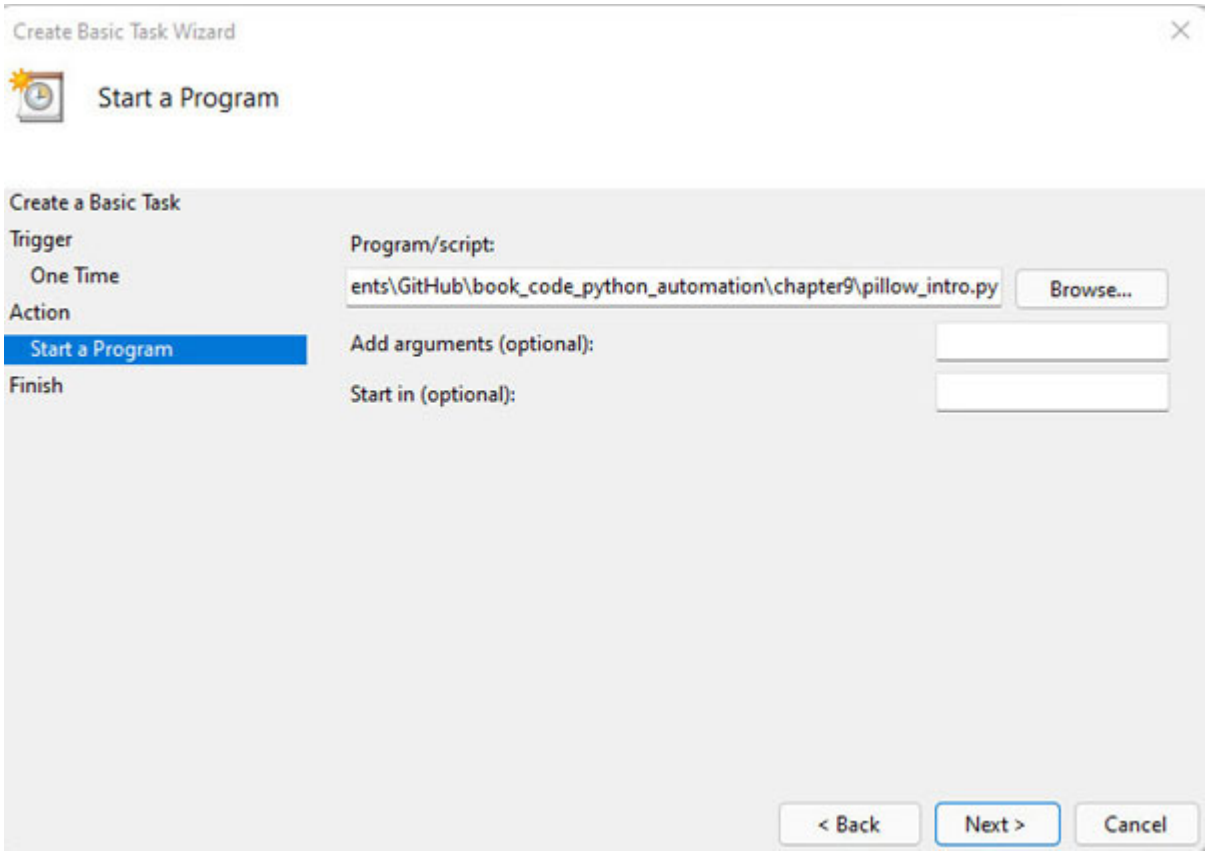


Figure 10.7: *Specifying the Python program path*

Once you specify the path, click on **`Next >`** where you will be presented with the summary of the task, and you need to click on finish to start the trigger of your task based on the selected trigger. You can schedule more complicated tasks using **task scheduler** using the create task option where you can have multiple triggers to run the same task. There are task scheduler applications available for Linux and Mac operating systems as well such as **crontab** which uses the Cron style syntax to schedule tasks.

In the next section, we will look at writing timer programs and scheduler triggers with the Python **`APScheduler`** library.

Writing timer programs

`APScheduler` allows you to write the timer programs to schedule and run Python programs as discussed in the previous section. **`APScheduler`** has a **`BlockingScheduler`** which is a simple scheduler that runs in the foreground.

You can start the scheduler by calling the **`start()`** function. With **`BlockingScheduler`**, you will start the scheduler once you are done with the initializing steps such as adding the jobs and passing the correct automation scripts. To add jobs to the scheduler, use the **`add_job()`** function that returns an **`apscheduler.job.Job`** instance that can be used to modify the job or remove it later. A scheduled job can be removed using the **`remove()`** function.

For example, **`scheduler.add_job(myfunc, 'interval', minutes=2)`** creates a new job and **`job.remove()`** removes the job. You can modify a job by using the **`modify()`** function and reschedule a job using the **`reschedule()`** function.

To shut down the scheduler, there is a **`shutdown()`** function, to pause the job use the **`pause()`** function, and to resume it, use the **`resume()`** function.

To start a simple scheduler job, you can create a **`BlockingScheduler`**, add a job to this scheduler using the **`add_job`** function, and start the scheduler using the **`start`** function as shown in *Figure 10.8*:

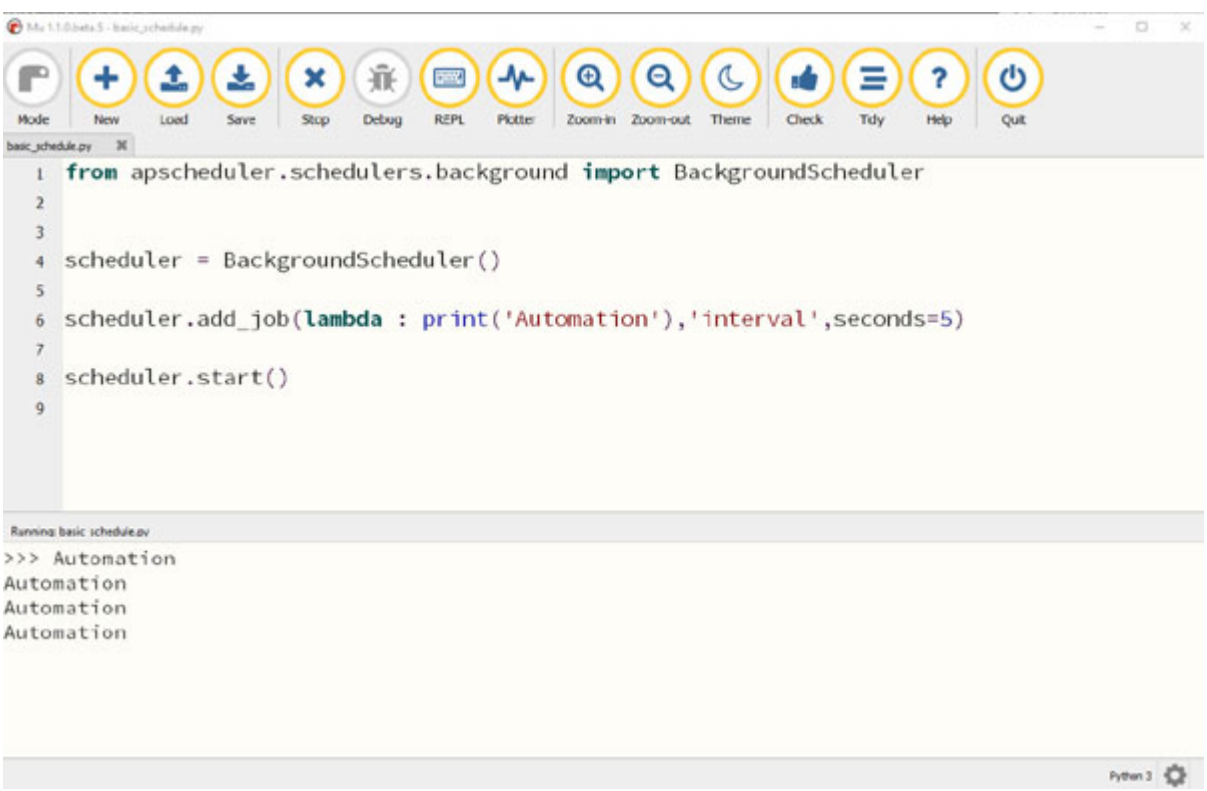


Figure 10.8: *Starting a simple APScheduler program*

Instead of passing a Lambda function, you can pass any other Python function to the scheduler, and the scheduler will call the function at the specified time. **`APScheduler`** supports cron-based triggers which is similar to the UNIX cron scheduler with parameter such as:

- **`year`**: 4-digit year number
- **`month`**: Month number (1-12)
- **`day`**: Day of month (1-31)
- **`week`**: ISO week (1-53)
- **`day_of_week`**: Number or name of weekday (**0-6** or **mon**, **tue**, **wed**, **thu**, **fri**, **sat**, **sun**)
- **`hour`**: Hour (0-23)
- **`minute`**: Minute (0-59)
- **`second`**: Second (0-59)
- **`start_date`**: Start date/time to start the trigger
- **`end_date`**: End date/time to end the trigger
- **`timezone`**: Time zone to use for the date/time calculations (defaults to scheduler timezone)

For example, you can run a cron-based trigger to run a job every second by passing the second parameter as `*` as shown in *Figure 10.9*:

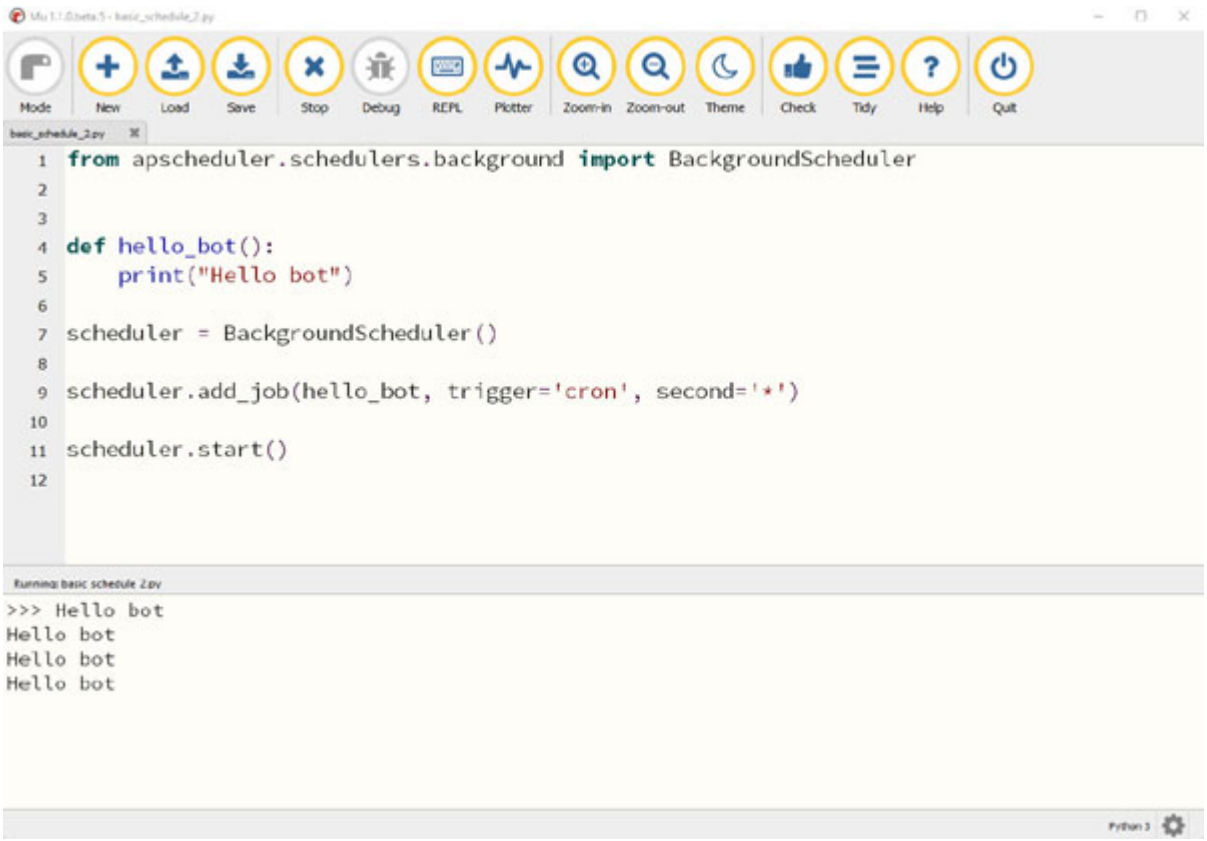


Figure 10.9: *Cron-based scheduler program*

In the next section, we will look at libraries to launch other programs and applications using Python.

Launching programs from Python

You can also launch different programs and applications from Python. This is particularly useful with timer programs; for example, at every login, you want certain applications to start on your desktop and be set up automatically.

A Python script can start other programs on your computer using the **`subprocess.Popen()`** function. The sub process module allows you to create new processes, connect to the input, and output pipes, and obtain the return code from external programs.

The **`subprocess.Popen`** takes arguments as a sequence of program arguments or program string. One example of using the launch program is shown in the following *Figure 10.10*:

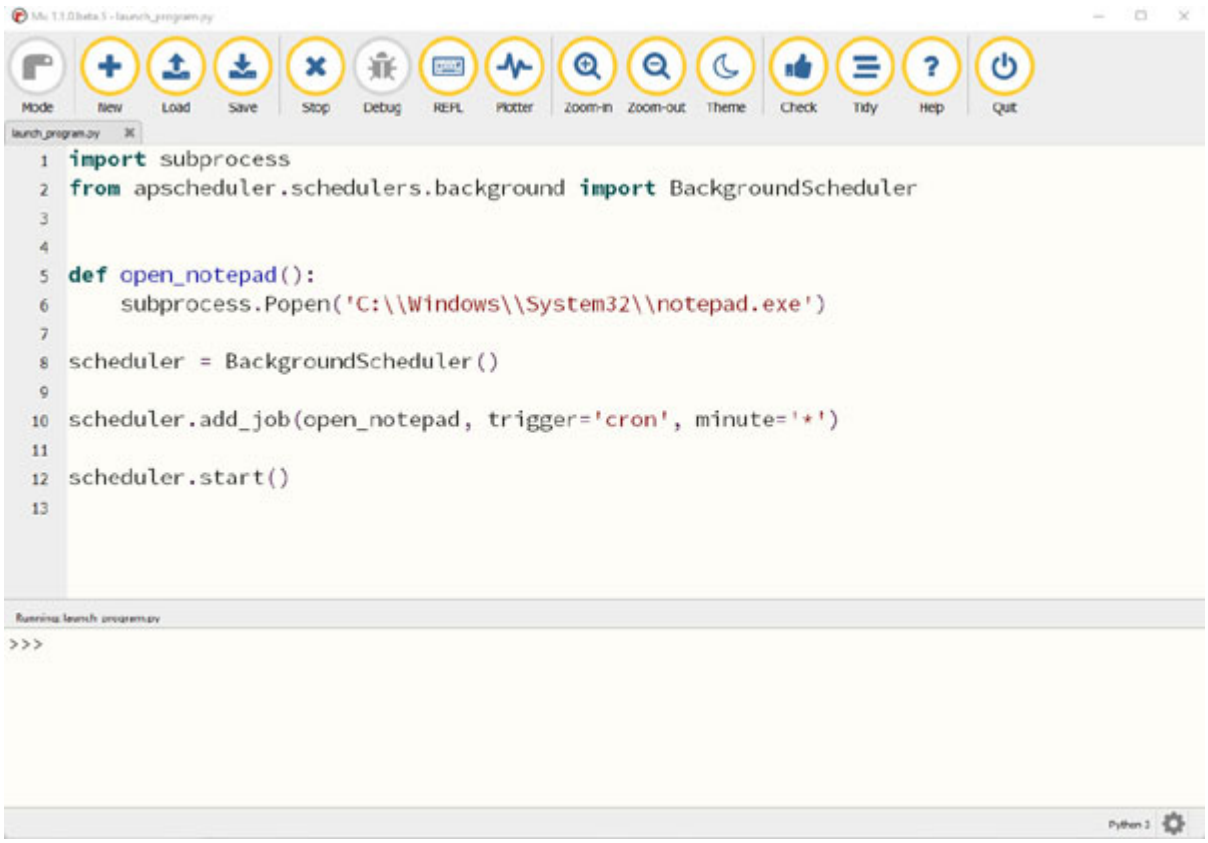


Figure 10.10: *Cron-based scheduler to launch program*

In the next section, we will look at some of the external tools that can help you run trigger-based automations. External tools provide the benefit that they are much easier to use, come with pre-configured workflows, and you do not have to program the triggers yourself.

Using external tools for triggers

One of the most popular ways to automate workflows based on triggers is to use external workflow automations tools such as **n8n** (**https://n8n.io/**) which is open source and has its source code available to modify for customizations (**https://github.com/n8n-io/n8n**). The **n8n** tool has a desktop app and has a lot of workflow templates from which you can choose a desired workflow to run based on your requirements. The home screen is shown in *Figure 10.11*:

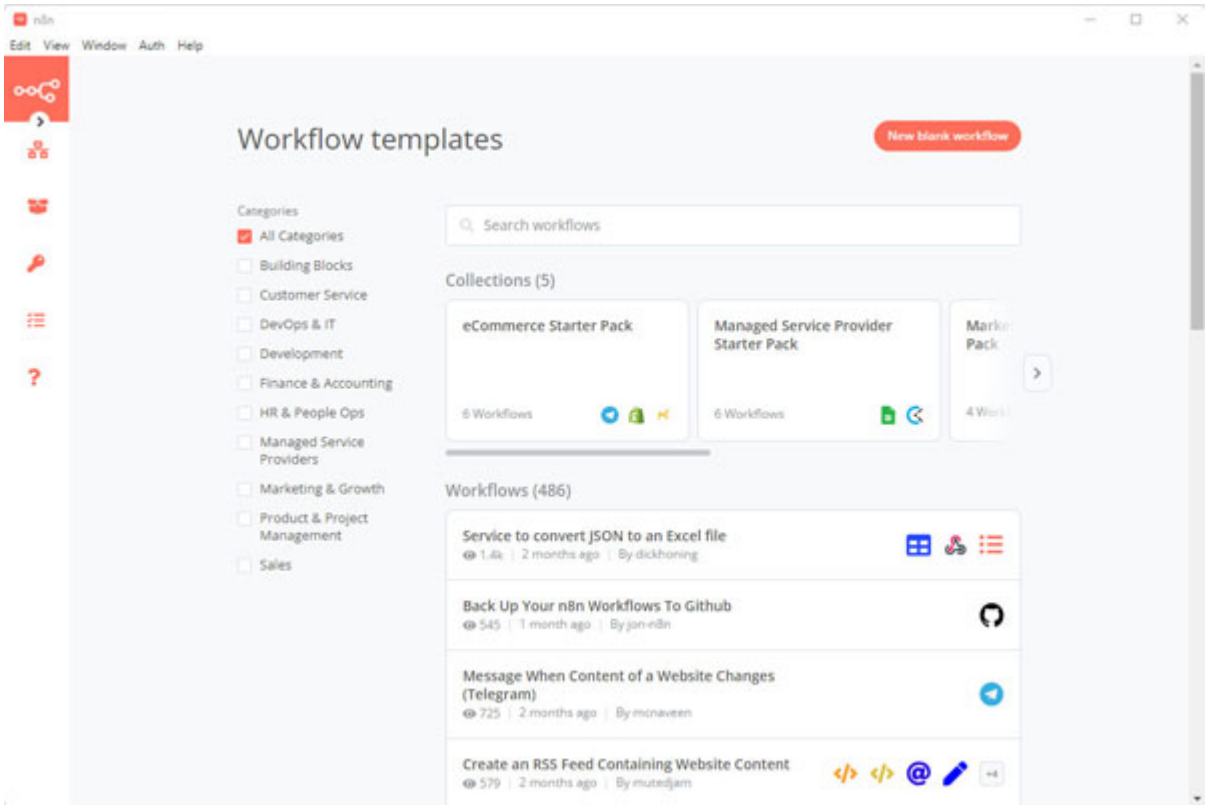


Figure 10.11: *Home screen of n8n*

For example, one of the workflow automation possible with **n8n** is to **`Gmail - Get emails with a certain label, remove the label, and add a new one`** as shown in *Figure 10.12*. *Further documentation on how to configure the workflow is available on the* **n8n** website and the desktop application:

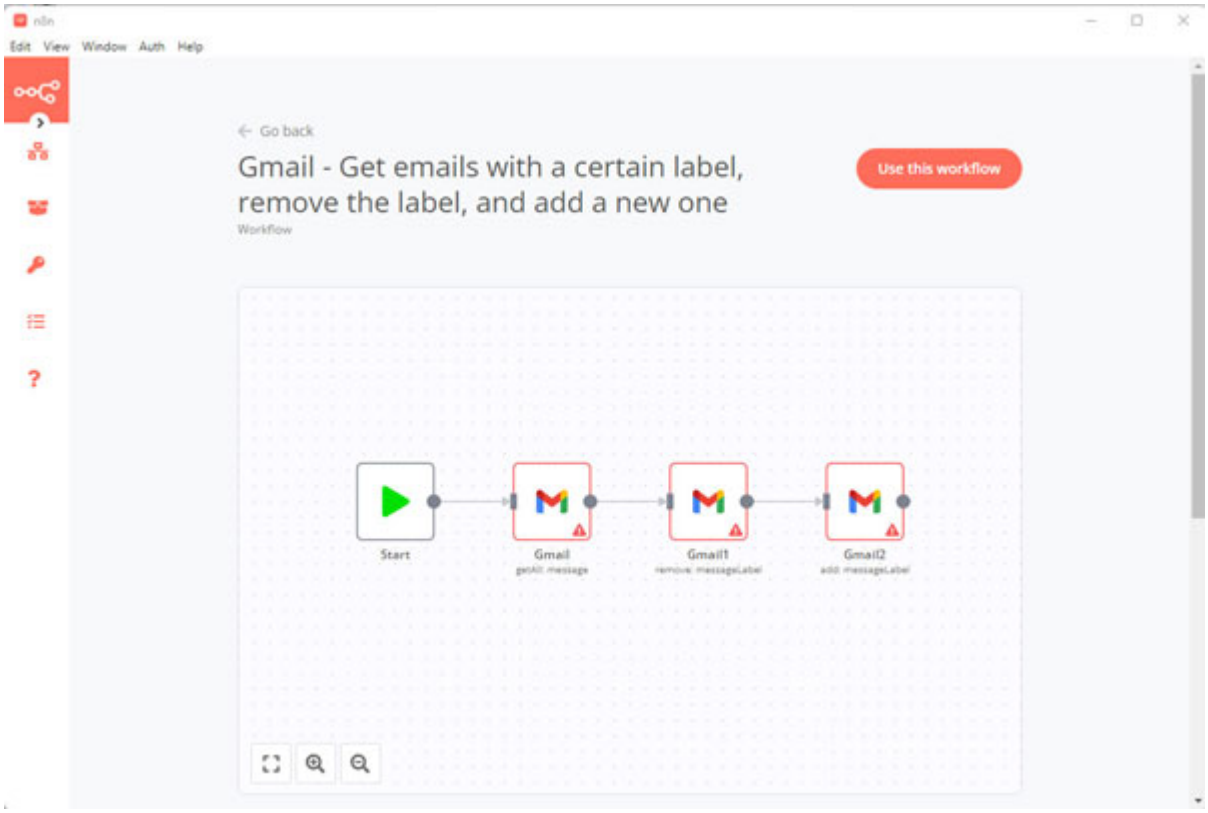


Figure 10.12: *Cron-based scheduler to launch program*

Python also has a `Twisted` library (https://pypi.org/project/Twisted/) which can be used for asynchronous programming and event-based framework for Internet applications to create web automation triggers.

Conclusion

In this chapter, we learned about timer programs, the Python `APScheduler` library, and Windows task scheduler. We also looked at the Python subprocess library to launch new programs and the **n8n** automation tool to create web-based automation based on triggers.

In the next chapter, we will look at writing more complicated automations based on the things we learned in this book. We would also look at creating Python web services using the Flask API which will allow you to create web-based automation endpoints that can be deployed to a server and the APIs can be shared across multiple applications.

Further reading

There are a lot of online resources to help you learn more about creating time and event-based automations. The following *Table 10.1* lists some of the best resources to further improve your learning on time and event-based automations.

Resources on image automation in Python:

Resource name	Link
Advanced Python Scheduler	**https://apscheduler.readthedocs.io/en/3.x/**
Introduction to APScheduler	**https://betterprogramming.pub/introduction-to-apscheduler-86337f3bb4a6**
How to create an automated task using Task Scheduler on Windows 10	**https://www.windowscentral.com/how-create-automated-task-using-task-scheduler-windows-10**
Subprocess management	**https://docs.python.org/3/library/subprocess.html**
Subprocess module	**https://www.bogotobogo.com/python/python_subprocess_module.php**
n8n - Automate without limits	**https://n8n.io/**
n8n - Workflow automation tool	**https://github.com/n8n-io/n8n**
Twisted library	**https://www.twistedmatrix.com/trac/**

Table 10.1: *Resources on timer and event-based automation in Python*

Questions

1. How can you schedule to run an automation at **9:00 am** every day?
2. What is the library used to write timer programs in Python?
3. What is **n8n**?
4. How can you create automations based on triggers?

CHAPTER 11

Writing Complex Automations

Introduction

In this chapter, we will look at methods to extend your Python scripting knowledge and develop complex end to end process automations based on your requirements. We will learn how to work with external libraries and use external code to build these automations. We will also look at creating Python web services and using machine learning for automation.

Structure

In this chapter, we will cover the following topics:

- Creating APIs with Python
- Combining multiple automation scripts
- Finding solutions online
- Using machine learning for automation

Objectives

After studying this chapter, you will be able to build a web server in Python using the Flask APIs and build complex automations integrating multiple automation libraries. You will also learn about machine learning techniques that can be used to build automation programs.

Creating APIs with Python

You can create the **Application Programming Interface** (**API**) using the **Flask** library in Python. APIs allow you to connect different applications and are particularly useful when automating applications based on triggers. For example, you can create an API to check for incoming emails and run the required automation. In this section, we will mainly look at

Representational State Transfer (**REST**) APIs that are flexible, lightweight, and the most common way to connect components and applications.

The REST APIs use four common HTTP methods, `GET` (provides read only access to resources), `POST` (used for creating new resources), `DELETE` (used for removing a resource), and `PUT` (used for updating an existing resource).

We will use the `Flask` Python library to create REST APIs based server in Python. **Flask** is a micro web framework for Python and can be used to create web applications from scratch. You can build web pages, applications like Wikipedia, commercial websites, or even a search engine like *Google* using the `Flask` library. `Flask` also supports template engines to build dynamic websites.

To install the `Flask` library, use the `mu` package manager, type `Flask`, and click on `OK` as shown in the following figure:

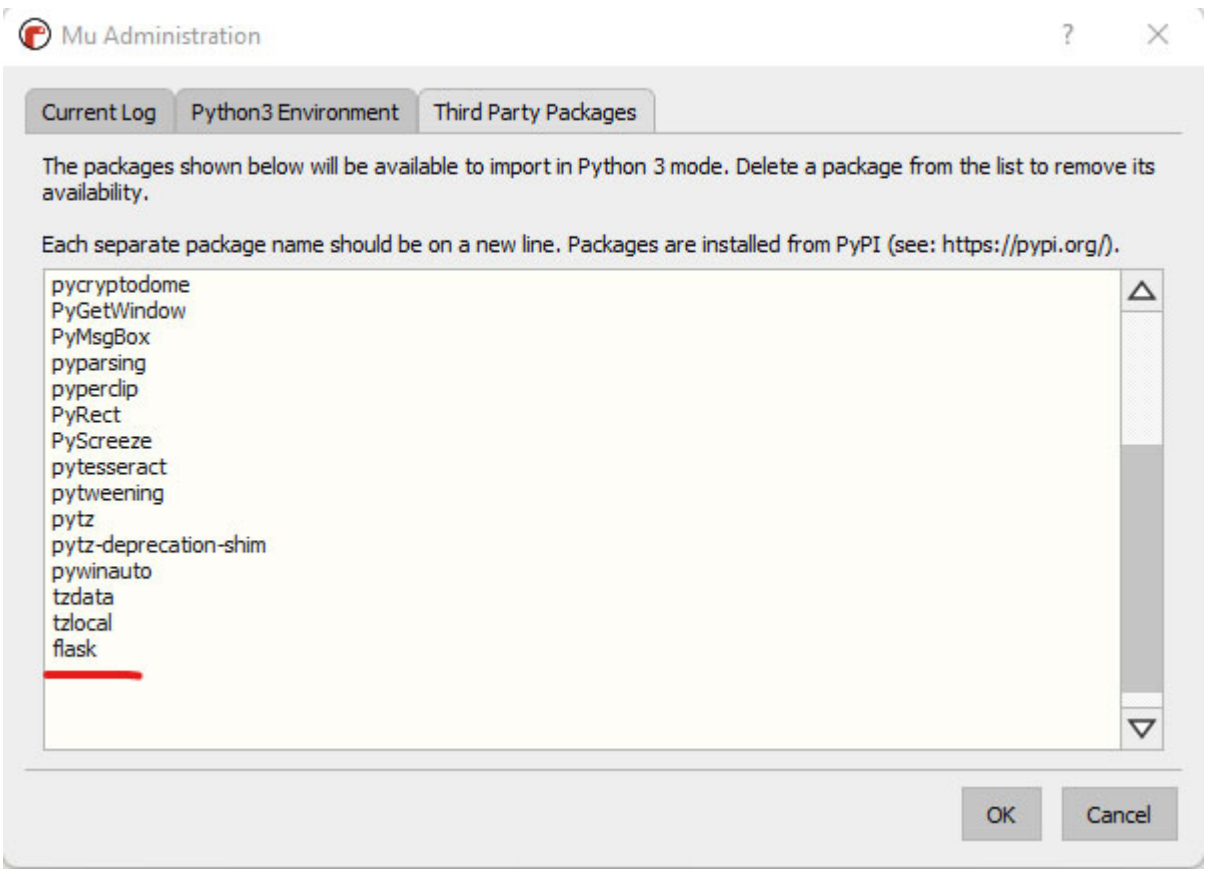


Figure 11.1: *Mu package manager*

To create a simple Flask application, you will need to import the `Flask` class whose instance will allow us to create a **Web Server Gateway Interface** (**WSGI**) application.

To create an instance of the `Flask` class (instance is a specific realization of a class), pass the name of the application's module, or use `__name__` as a convenient shortcut. This is required to let the `Flask` know the location to look for resources such as templates and static files.

We then use the `route()` decorator to specify the URL path to trigger the function. A **decorator** in Python is a function that extends the behavior of another function without explicitly modifying it. We use the `@my_decorator` syntax to easily call the decorator function.

After using the `route()` decorator, we can call any function and return the data we want to show in the document. The default content type is HTML so when you pass an HTML string, the HTML data would be rendered by the browser.

To run a `Flask` application locally, you call the `app.run` function inside the `main` method and pass in the arguments as:

- `host`: The IP address of the Python webserver; by default we use local host which is at IP `127.0.0.1`.
- `port`: The port to host the Python webserver; by default, we use `8080`.
- `debug`: Set `True` if you want to enable the debug mode, `False` otherwise.

Figure 11.2 contains an example of creating a simple flask application that can return the `Automation Bot!` string at the default route of `/`:

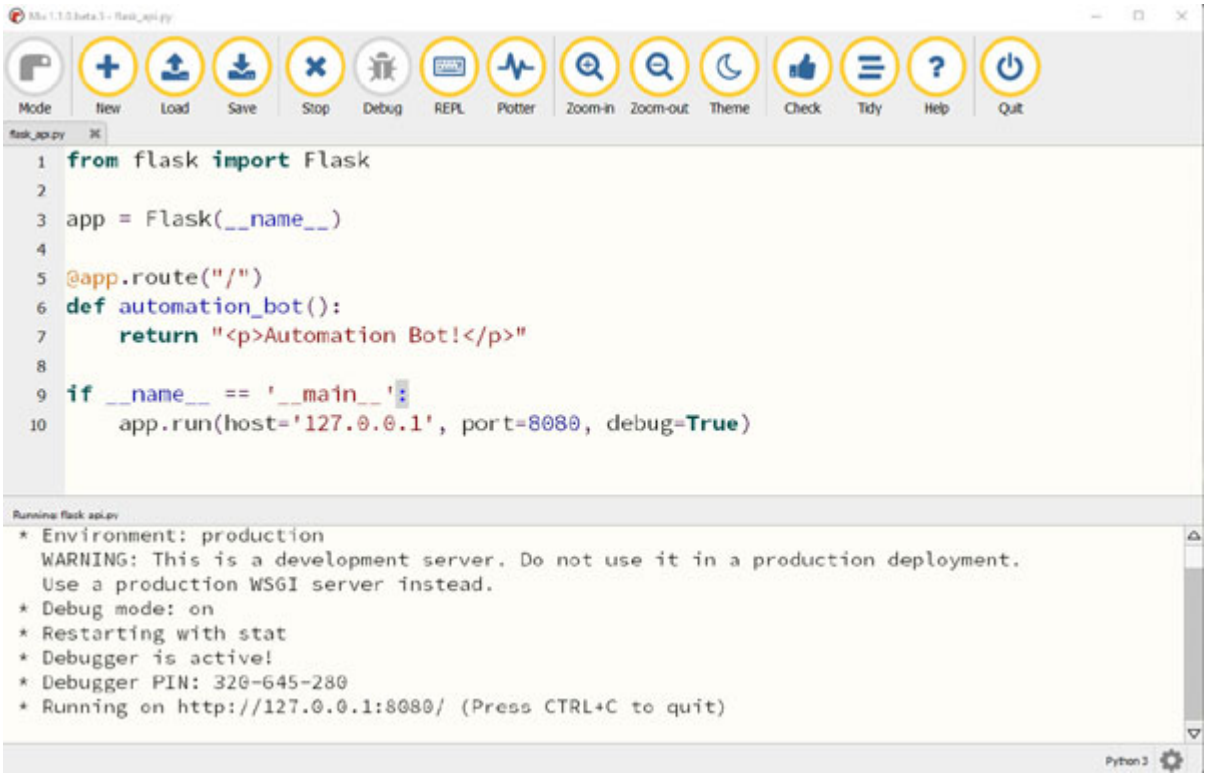


Figure 11.2: *Simple flask application*

Once the webserver is running, you can go to the specified host and port number; in this case, `127.0.0.1:8080` in your browser, and you will see `Automation Bot!` being printed out as shown in *Figure 11.3*:

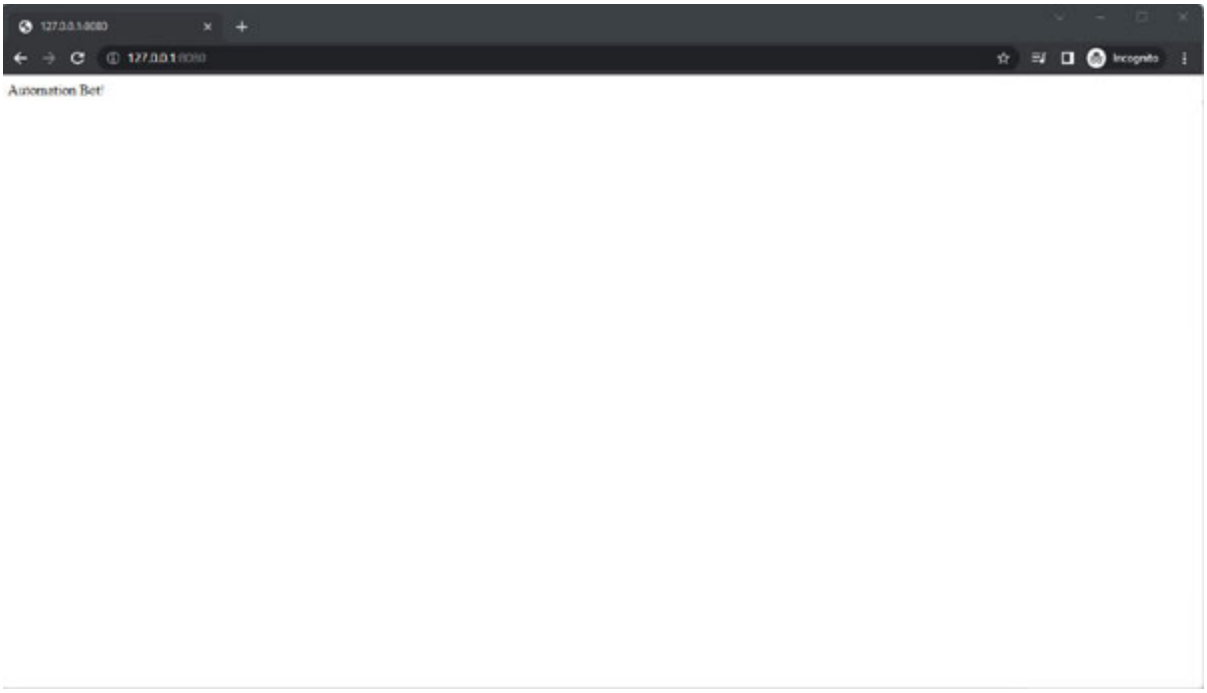


Figure 11.3: Output from the Python webserver

Flask provides the ability to create dynamic routes with variables. You can add the variable sections in the URL route by marking the sections with **`<variable_name>`**. The function then receives the **`variable_name`** as a keyword argument. It is important to note that when returning HTML data in **`Flask`** (which is its default type), if there are any user-provided values, they must be escaped to protect from injection attacks. The **`escape()`** function from the markup safe provides the functionality to escape the user provided data.

With the **`Flask`** variable route; for example, you can create dynamic routes such as passing the bot name in the URL route which gets passed on the function that can use this variable and return it to display it in the browser, as shown in *Figure 11.4*:

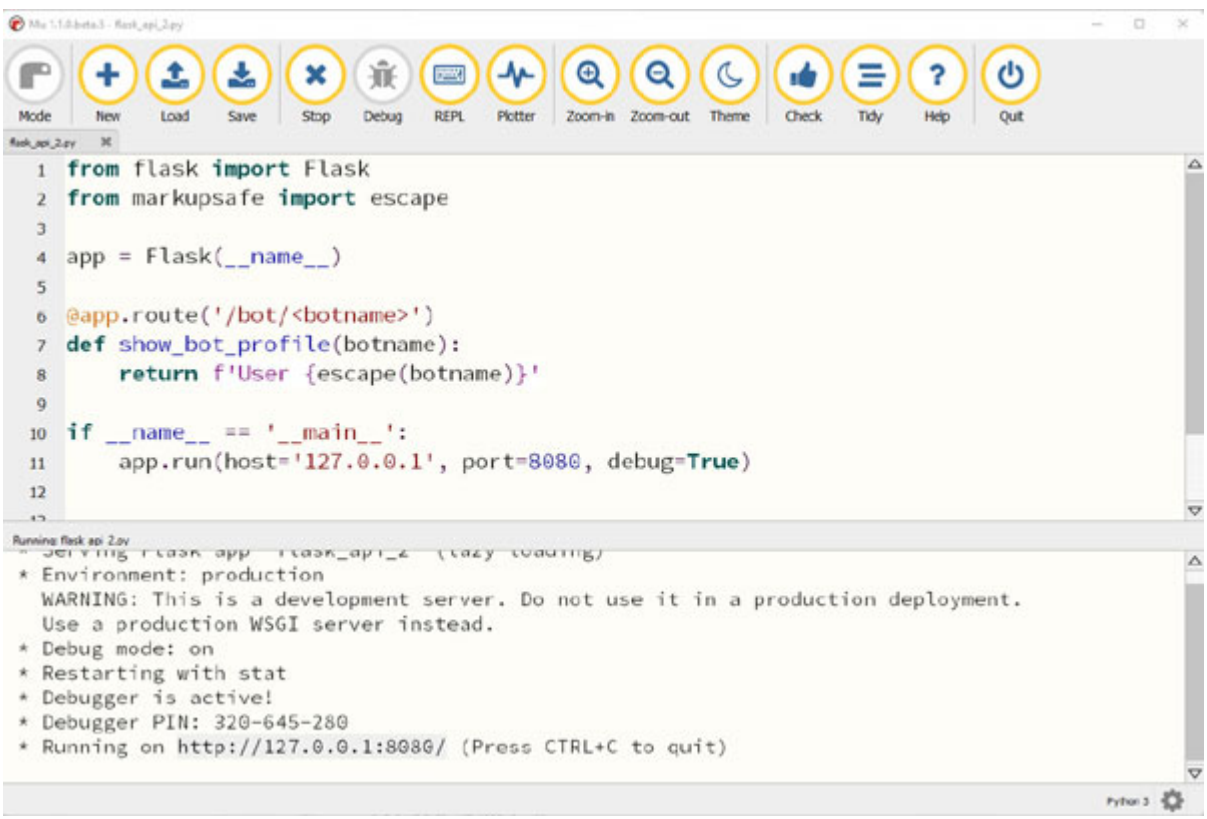


Figure 11.4: Variable route template in Flask

Once you start the webserver, you can go to the specified host and port number; in this case, **`127.0.0.1:8080`** in your browser, and then add

`/bot/<any_value>` in the URL, and you will see the variable value being printed out with the prefix `User` as shown in the browser in *Figure 11.5*:

Figure 11.5: *Output from the Python webserver with the dynamic route*

Flask has a `url_for()` function that is used to build an URL for a specific function. It takes in the name of the function as the first argument and keyword arguments corresponding to the variable part of the URL rule. Unknown variable parts are added to the end of the URL as query parameters. This URL building method also shows the escaping of special characters and the paths generated are always absolute paths.

For example, we can use the `test_request_context()` method to try out the `url_for()` function to get the URL for functions as shown in *Figure 11.6*:

```
from flask import Flask, url_for
from markupsafe import escape

app = Flask(__name__)

@app.route("/")
def automation_bot():
    return "Automation Bot"

@app.route('/bot/<botname>')
def show_bot_profile(botname):
    return f'User {escape(botname)}'

with app.test_request_context():
    print(url_for('automation_bot'))
    print(url_for('automation_bot', next='/'))
    print(url_for('show_bot_profile', botname='TestBot'))
```

```
/
/?next=%2F
/bot/TestBot
>>>
```

Figure 11.6: *Flask url_for function to get URL paths*

Flask allows you to create a webserver supporting different HTTP methods such as **`GET`**, **`POST`**, **`DELETE`**, and **`PUT`** requests. By default, the **`Flask`** route allows only GET requests. You need to use the **`route()`** decorator to handle different HTTP methods.

For example, you can specify **`GET`** and **`POST`** routes for the default URL using the methods argument such as **`@app.route('/, methods=['GET', 'POST'])`**. The **`Flask`** library contains the request object which is created by default whenever a request is made to the URL route. To get the **`request`** method received by the function, the **`method`** attribute is available which can tell you which type of request is made by the user (such as **`GET`**, **`POST`**, **`DELETE`**, or **`PUT`** requests). By specifying the different HTTP methods in the **`route()`** decorator, you can do different tasks based on the request type received such as a **`POST`** request or a **`GET`** request.

You can check whether the **`request.method`** equals **`POST`**, then get the data from the **`POST`** request, and complete the **`POST`** request otherwise perform a **`GET`** request. We can create an automation bot endpoint to perform the

automation on the **POST** request and provide the list of available automation on the **GET** request as shown in the example in *Figure 11.7*:

```
from flask import Flask
from flask import request

app = Flask(__name__)

@app.route('/automation_bot', methods=['GET', 'POST'])
def automation_bot():
    if request.method == 'POST':
        return "Doing the automation"
    else:
        return "I can do automations"

if __name__ == '__main__':
    app.run(host='127.0.0.1', port=8080, debug=True)
```

```
Use a production WSGI server instead.
* Debug mode: on
* Restarting with stat
* Debugger is active!
* Debugger PIN: 320-645-280
* Running on http://127.0.0.1:8080/ (Press CTRL+C to quit)
```

Figure 11.7: *Bot endpoint with GET and POST requests*

To get the data from the **GET** request for the preceding webserver, you can navigate to the automation bot endpoint in the browser as shown in *Figure 11.8*:

Figure 11.8: *Bot endpoint returning data with the GET request*

To get the data from the **POST** request, you will need to use the Python **request** library or tools such as **Postman**. **Postman** is an API platform that allows you to easily test and document APIs, and it can be downloaded from **https://www.postman.com/downloads/**. We can easily send a **POST**

request with **Postman** and check the response of the **POST** request as shown in *Figure 11.9*:

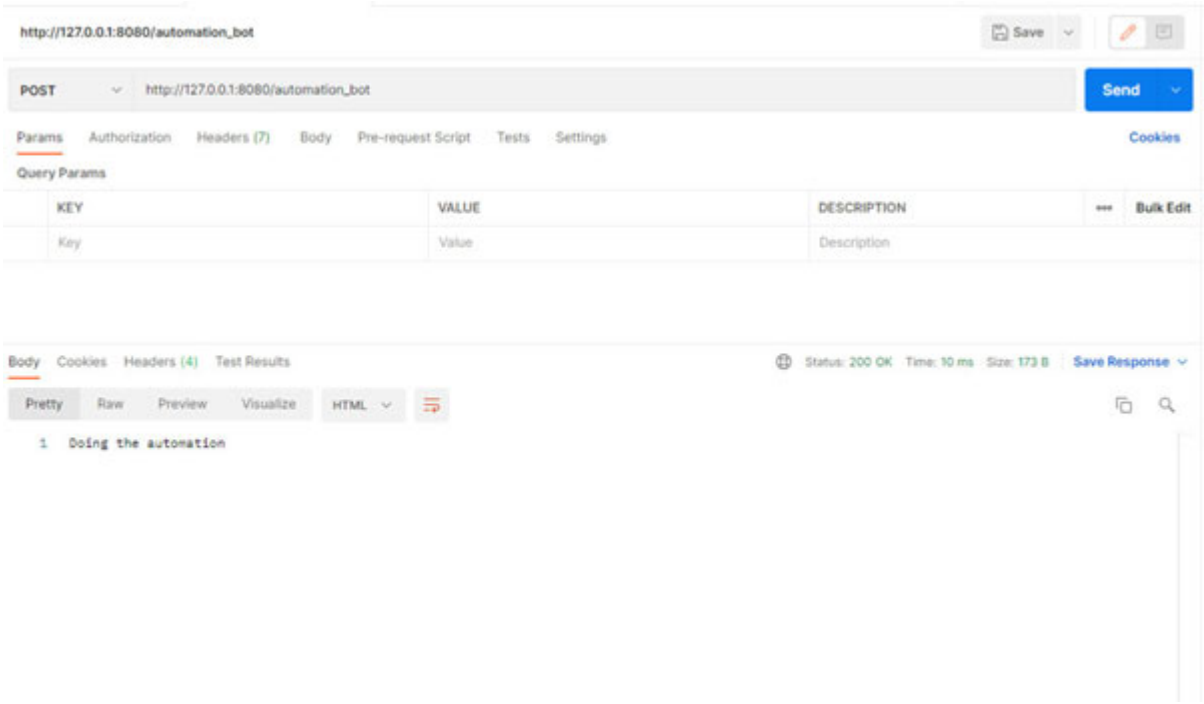


Figure 11.9: *Sending POST requests with Postman*

Flask can also be used to build dynamic web applications with static files. The static files will usually include CSS, JavaScript, and other files that are required for the web application. To generate URLs for these static files, you can use the **`url_for`** function with the special **`static`** endpoint name, such as **`url_for('static', filename='style.css')`**. This file needs to be stored on the file system as **`static/style.css`**. Flask also supports HTML template engines with **Jinja2** as its default template engine. Templating engines allow you to modify and reuse the common HTML code across several pieces of view.

In the next section, we will look at an example of combining the **`Flask`** webserver scripts with other automation scripts we learned throughout this book to build automations for end-to-end processes.

Combining multiple automation scripts

You can build complex automation's scripts by combining different automation scripts we learned throughout the book. For end-to-end process

automation, you may need to convert a PDF document to a text file, extract data from the text file, and add it to the web form, submit the web form, and record the completed processes in a Word document. This automation process would require combining the scripts from *Chapter 6, Automating File-Based Tasks* and *Chapter 5, Automating Web-Based Tasks*. You can also create an API for this automation using Flask web service.

A simple way to create an API for automation is to call the following automation function the **`route()`** decorator and perform the required automation steps. For example, if we want to create a web service which opens a new Notepad application using the sub process library, we will call the **`subprocess.open`** to open the Notepad application in the **`open_notepad`** function and return a success message after the process is complete as shown in *Figure 11.10*:

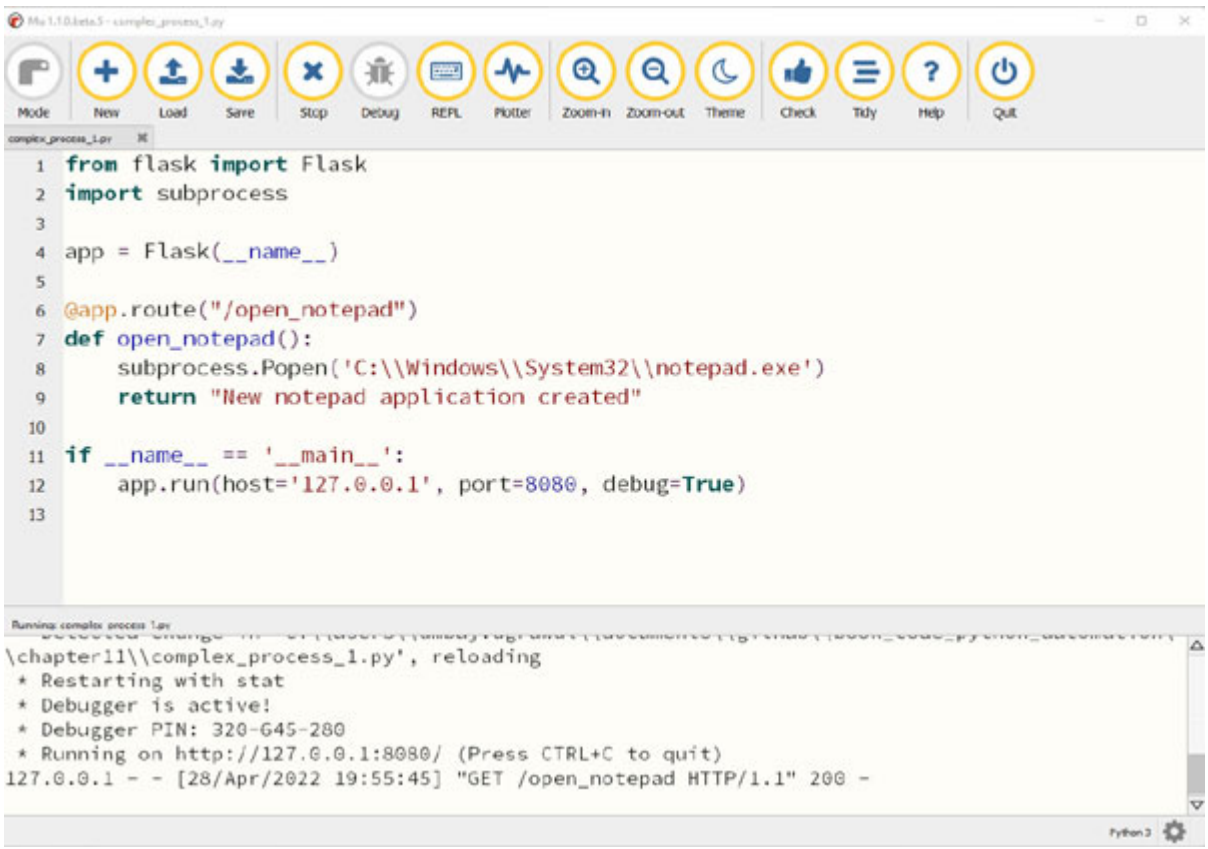


Figure 11.10: *Creating a web API to open new Notepad application*

When you call this web service endpoint, a new Notepad application is created on the server running the web service, and you will see a

confirmation message `New notepad application created` on the browser as shown in *Figure 11.11*:

Figure 11.11: *Response from the web service*

If you want this website to be available on the **World Wide Web** (**WWW**), you would need to deploy it on the public IP address of your computer, or deploy it on a cloud server. One of the best cloud services to deploy APIs is the *Google App Engine* that allows you to deploy your code on the fully managed server less platform. More information on the **App Engine** is available on the documentation page at **https://cloud.google.com/appengine**.

In this section, we looked at a simple example of combining multiple scripts learned in this book to create process automation. In the next section, we will look at some of the online resources that can help with solutions to common technical problems and discovering new libraries to help with your automation tasks.

Finding solutions online

One of the most popular websites to get answers to technical questions is **Stack Overflow**. Stack Overflow is a question and answer website where people post questions and solutions for technical problems and it has around *2 million questions* tagged with `Python` as a language as shown in *Figure 11.12*:

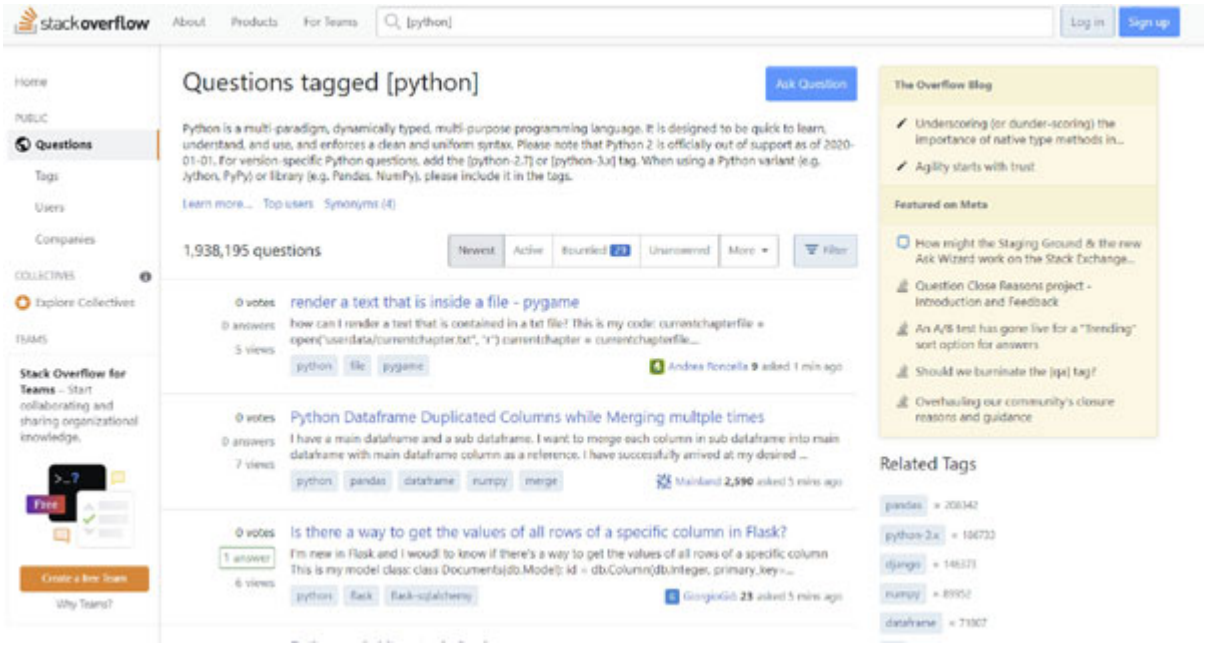


Figure 11.12: *Stackoverflow question on Python*

Even when you do a *Google* search for a particular problem, you would get result links from *Stack Overflow* which is generally one of the first places to start with when you are looking for a solution. Another good place to start looking for interesting automation libraries written in Python is GitHub which has over *2 million Python* programming language repositories as shown in *Figure 11.13*:

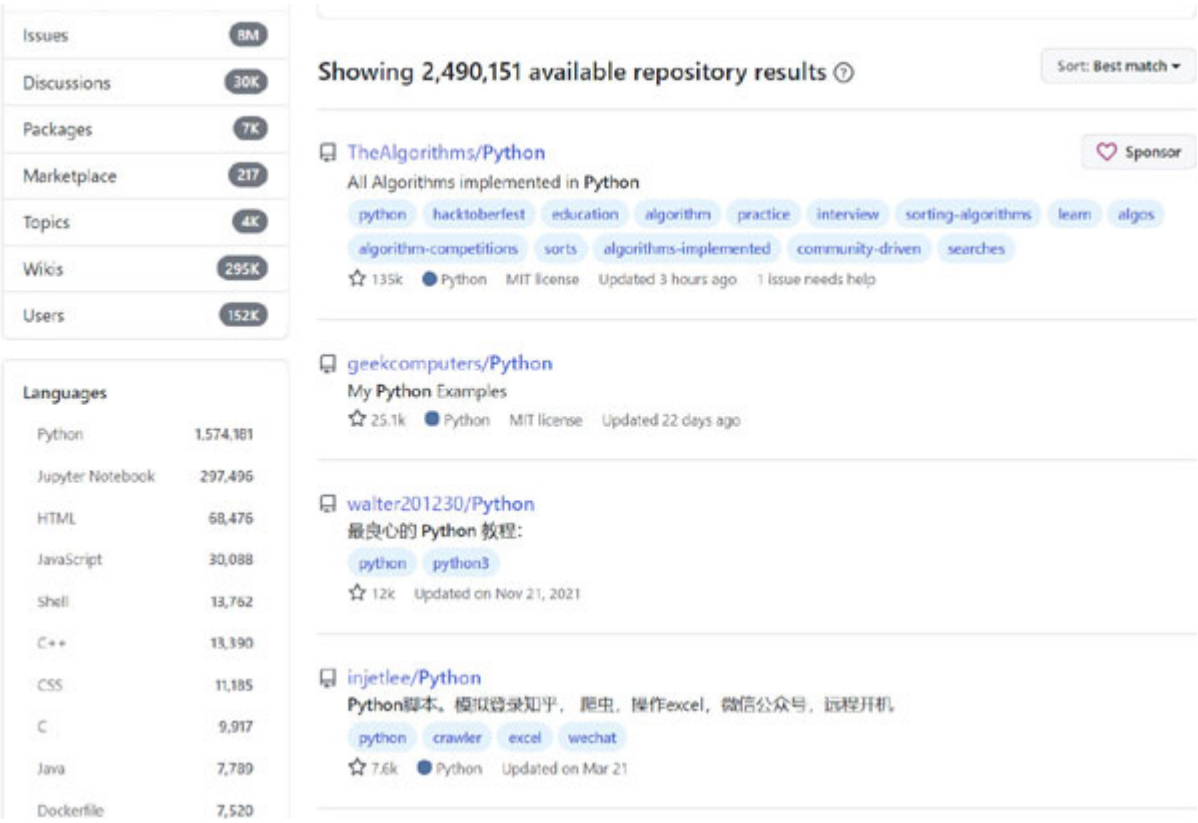


Figure 11.13: *GitHub repositories on Python*

If you are more specific in your search and looking for code repository tagged Python-automation, you will get relevant repositories for this such as repository and code for sending **Python Automated Bulk WhatsApp Messages,** and **Python Automation Scripts** as shown in the following figure:

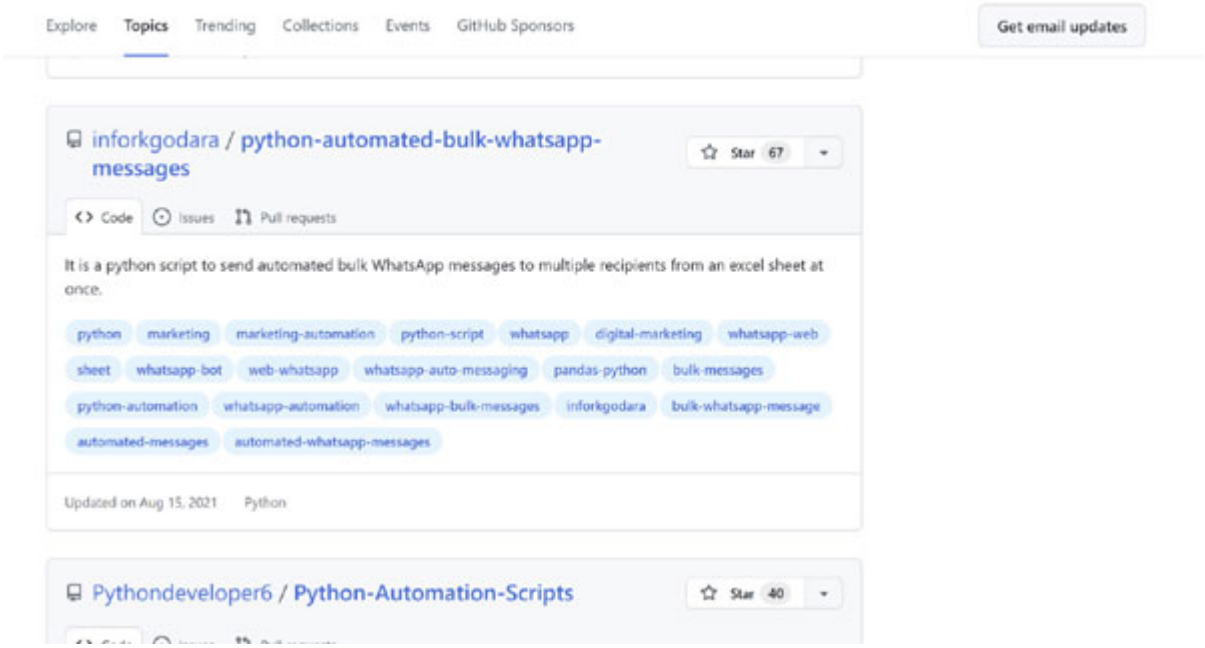


Figure 11.14: *GitHub repositories on Python Automation Scripts*

In the next section, we will look at the basics of machine learning that would be useful for creating automations.

Using machine learning for automation

Artificial Intelligence (**AI**) is a very broad field with many subareas and involves the study of automated recognition and understanding of signals, reasoning, planning, and decision-making learning, and adaptation. **Machine Learning** (**ML**) is a type of AI that provides computers with the ability to learn without being explicitly programmed.

Machine learning can be subdivided into three main categories which are as follows:

- **Supervised learning**: This involves labeled data and can be used for **classification** (grouping similar instances) and **regression** (learning *what normally happens* to draw inferences from datasets). For example, learning to classify emails as spam and not spam-based on training data of spam and not spam email messages.
- **Unsupervised learning**: This involves unlabeled data and can be used to discover patterns in the dataset. For example, learning the patterns in the *English* language from the *Wikipedia* pages.

- **Reinforcement learning**: This involves learning from experimentation based on rewards and feedback loop and can be used to train agents in simulated environments. For example, teaching a bot to play a computer game, and maximize the score, and chances of winning will use a **Reinforcement Learning** (**RL**) algorithm.

Machine learning allows you to learn from data and create automations without explicitly programming the automations. It can also help you identify repeatable processes performed in the organizations.

Python has a lot of libraries and scripts to perform ML on datasets with libraries such `PyTorch`, `TensorFlow`, `Keras`, `scikit-learn`, and so on. Training ML models requires a lot of data and computational power. For our automation requirements, generally a pre-trained ML model would work most of the time.

We will look at the `PyTorch` library to perform text summarization with a pre-built model. **Hugging Face** (**https://huggingface.co/**) provides pre-trained ML models for a variety of tasks audio classification, image classification, object detection, question answering, summarization, text classification, translation, and so on.

To install the **Hugging Face** with the `PyTorch` library, use the `mu` package manager, type `transformers[torch]`, and click on `OK` as shown in the following figure. The installation process will take a bit of time as it will install various dependencies and other ML libraries:

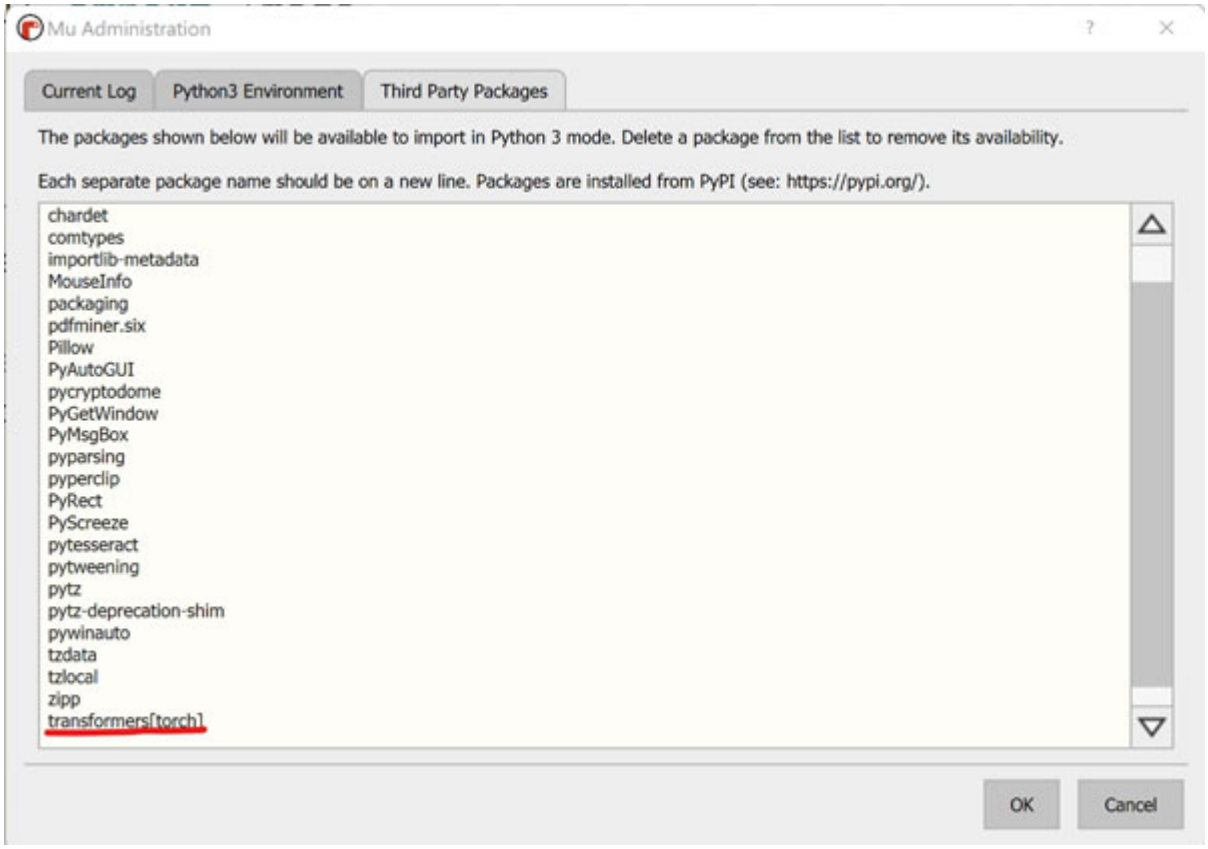


Figure 11.15: *Mu package manager*

Once the library is installed, you can use the transformers library to perform summarization by using the summarization pipeline object with the syntax as **`pipeline("summarization")`**. On the first run, the pipeline will download the default model to perform text summarization, and it will take a bit of time to complete the downloaded process. Once the model is downloaded, you can call the model on any text data, call the model on the text data, and you will get the summarized text as shown in *Figure 11.16*:

Figure 11.16: *Mu package manager*

You can also perform sentiment analysis on a piece of text using a pre-trained model. This is particularly useful when you want to analyze sentiments of customer reviews and feedback. To use the pre-trained model to perform *sentiment-analysis*, use **`pipeline("sentiment-analysis")`** as shown in *Figure 11.17*. You can also use a particular trained model by specifying a model argument to the **`pipeline`** function:

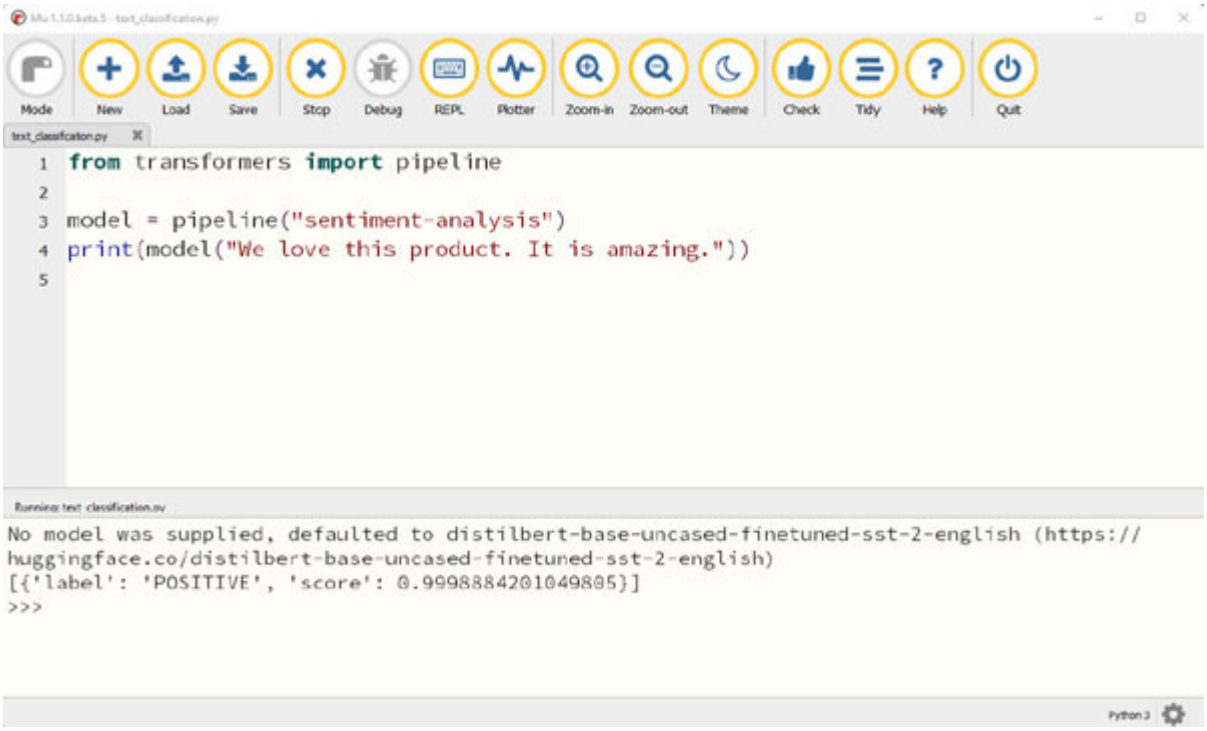


Figure 11.17: *Mu package manager*

There are a lot of other trained machine learning models available in Python that can be used for your day-to-day automation requirements. These ML models can help you build automations for converting images to text, automating message replies, language translations, and a variety of other tasks.

Conclusion

In this chapter, we learned about the Flask fundamentals and the machine learning libraries in the Python language. We looked at simple ways to use pre-trained ML models with few lines of code in Python with the transformers library. We also looked at some of the ways to build end to end process automations by combining different automations throughout this book and online resources to help you find solutions to technical challenges.

Further reading

There are a lot of online resources to help you improve your learning on building complex automation machine learning. The following *Table 11.1*

lists some of the best resources to further improve your learning on Flask and machine learning:

Resource name	Link
Pillow documentation	**https://flask.palletsprojects.com/en/2.1.x/**
An introduction to the Flask Python web app framework	**https://opensource.com/article/18/4/flask**
Developing RESTful APIs with Python and Flask	**https://auth0.com/blog/developing-restful-apis-with-python-and-flask/**
Python machine learning	**https://www.w3schools.com/python/python_ml_getting_started.asp**
Your first machine learning project in Python step-by-step	**https://machinelearningmastery.com/machine-learning-in-python-step-by-step/**
The AI community building the future	**https://huggingface.co/**
Hugging Face transformers	**https://www.kdnuggets.com/2021/02/hugging-face-transformer-basics.html**
PyTorch - open source machine learning framework	**https://pytorch.org/**
An end-to-end open source machine learning platform	**https://www.tensorflow.org/**
Keras: the Python deep learning API	**https://keras.io/**
Introduction to machine learning	**https://developers.google.com/machine-learning/crash-course/ml-intro**

Table 11.1: *Resources on Flask and machine learning in Python*

Questions

1. What is Flask application?
2. How can you create APIs with Python?
3. How do you combine multiple automation scripts?
4. What are the most popular machine learning libraries in Python?

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

R

S

T

V

W