

# **Introduction to Python for Data Analysis**

---



**AI SCIENCES**

AI Publishing

# **INTRODUCTION TO PYTHON FOR DATA ANALYSIS**

**Guide for Beginners**

**AI Sciences Publishing**



**AI SCIENCES**

## How to contact us

Please address comments and questions concerning this book  
to our customer service by email at:

[contact@aisciences.net](mailto:contact@aisciences.net)

*Our goal is to provide high-quality books for your technical learning in  
Data Science and Artificial Intelligence subjects.*

*Thank you so much for buying this book.*

If you noticed any problem, please let us know by  
sending us an email at [review@aisciences.net](mailto:review@aisciences.net) before  
writing any review online. It will be very helpful for us  
to improve the quality of our books.



**AI SCIENCES**

# Table of Contents

<b>Table of Contents .....</b>	<b>iv</b>
About Us .....	1
About our Books .....	2
To Contact Us: .....	2
<b>From AI Sciences Publishing .....</b>	<b>3</b>
<b>Preface .....</b>	<b>6</b>
Who Should Read This? .....	6
How to Use This Material? .....	7
Why learn Python & programming? .....	9
<b>A Quick Example .....</b>	<b>14</b>
The Big Picture: Using Python for Data Analysis .....	18
Mastering the Basics .....	18
<b>Python Setup.....</b>	<b>20</b>
Overview.....	20
Download & Install Anaconda .....	20
Open Jupyter Notebook .....	21
Install Atom & Download Packages .....	23
Opening & Running the Terminal.....	26
Common Mistakes .....	28
How to Fix Errors .....	28
Summary & Review .....	29
<b>What is Python.....</b>	<b>31</b>

What is Programming? .....	31
How Useful It Is Really? .....	31
Why Many Universities are Teaching Python .....	32
Clarity & Intuitiveness .....	32
<b>Python Level One .....</b>	<b>34</b>
Using the Interactive Shell .....	34
Simple Operations Including Maths .....	36
Python Data Types (Integers, Floats, and Strings) .....	38
What Can You Do with Strings .....	40
Summary & Review .....	41
More Examples & Exercises .....	41
<b>Python Level Two.....</b>	<b>43</b>
Print Statements .....	43
Assigning Variables.....	44
Creating Your First Program.....	46
Comments for Your Own Notes.....	52
Asking the User for an Input.....	53
Summary & Review .....	54
More Examples & Exercises .....	54
<b>Python Level Three .....</b>	<b>56</b>
Comparison Operators .....	56
Boolean Operators (and, or, not).....	57
If, Elif, and Else Statements.....	59
While Loops .....	61
For Loops .....	65
Summary & Review .....	71

More Examples & Exercises .....	72
<b>Python Level Four .....</b>	<b>74</b>
Creating Mini-Programs within Your Program (Functions) .....	74
Scope (Global & Local Variables) .....	77
Handling Errors & Exceptions .....	79
Summary & Review .....	83
More Examples & Exercises .....	83
<b>Python Level Five .....</b>	<b>86</b>
Creating a List .....	86
Indexing the List Values (starts at 0, not 1) .....	87
Slicing the List .....	87
Getting the Number of List Values .....	89
Changing the Values .....	89
List Operations (e.g. Concatenation, Append) .....	90
Summary & Review .....	92
More Examples & Exercises .....	93
<b>Recap and Some Advice .....</b>	<b>95</b>
<b>Resources .....</b>	<b>96</b>
<b>The Zen of Python .....</b>	<b>96</b>
<b>Thank you ! .....</b>	<b>98</b>



- Do you want to discover, learn and understand the methods and techniques of artificial intelligence, data science, computer science, machine learning, deep learning or statistics?
- Would you like to have books that you can read very fast and understand very easily?
- Would you like to practice AI techniques?

If the answers are yes, you are in the right place. The AI Sciences book series is perfectly suited to your expectations!

Our books are the best on the market for beginners, newcomers, students and anyone who wants to learn more about these subjects without going into too much theoretical and mathematical detail. Our books are among the best sellers on Amazon in the field.

### *About Us*

We are a group of experts, PhD students and young practitioners of Artificial Intelligence, Computer Science, Machine Learning and Statistics. Some of us work in big companies like Google, Facebook, Microsoft, KPMG, BCG and Mazars.

We decided to produce a series of books mainly dedicated to beginners and newcomers on the techniques and methods of Machine Learning, Statistics, Artificial Intelligence and Data Science. Initially, our objective was to help only those who wish to understand these techniques more easily and to be able to start without too much theory and without a long reading.

Today we also publish more complete books on some topics for a wider audience.

### *About our Books*

Our books have had phenomenal success and they are today among the best sellers on Amazon. Our books have helped many people to progress and especially to understand these techniques, which are sometimes considered to be complicated rightly or wrongly.

The books we produce are short, very pleasant to read. These books focus on the essentials so that beginners can quickly understand and practice effectively. You will never regret having chosen one of our books.

We also offer you completely free books on our website: Visit our site and subscribe in our Email-List: [www.aisciences.net](http://www.aisciences.net)

By subscribing to our mailing list, we also offer you all our new books for free and continuously.

### *To Contact Us:*

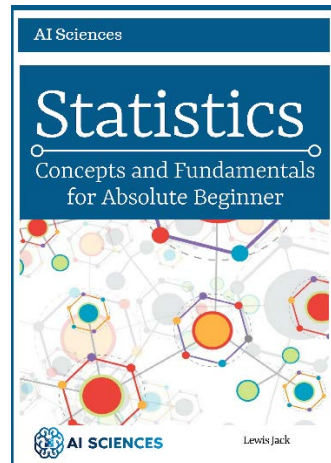
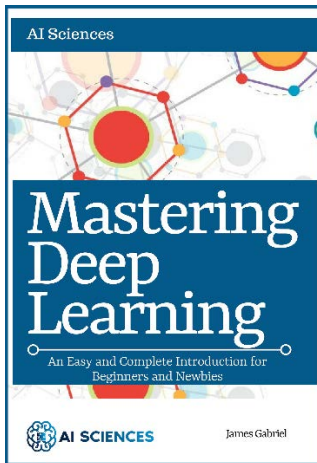
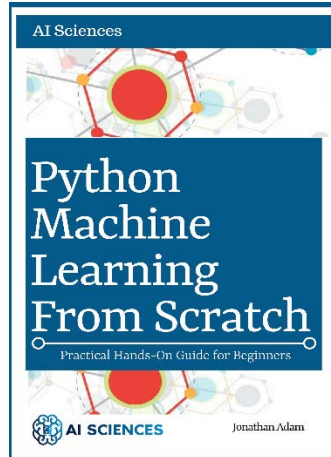
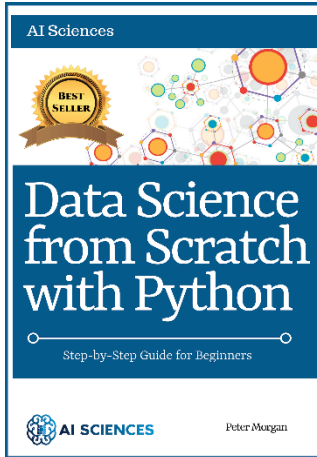
- Website: [www.aisciences.net](http://www.aisciences.net)
- Email: [contact@aisciences.net](mailto:contact@aisciences.net)

Follow us on social media and share our publications

- Facebook: [@aisciencesllc](https://www.facebook.com/aisciencesllc)
- LinkedIn: **AI Sciences**



## From AI Sciences Publishing



**[WWW.AISCIENCES.NET](http://WWW.AISCIENCES.NET)**

EBooks, free offers of eBooks and online learning courses.

Did you know that AI Sciences offers free eBooks versions of every books published? Please subscribe to our email list to be aware about our free eBook promotion. Get in touch with us at [contact@aisciences.net](mailto:contact@aisciences.net) for more details.



At [www.aisciences.net](http://www.aisciences.net) , you can also read a collection of free books and receive exclusive free ebooks.

**[WWW.AISCIENCES.NET](http://WWW.AISCIENCES.NET)**

Did you know that AI Sciences offers also online courses?

We want to help you in your career and take control of your future with powerful and easy to follow courses in Data Science, Machine Learning, Deep learning, Statistics and all Artificial Intelligence subjects.

Most courses in Data science and Artificial Intelligence simply bombard you with dense theory. Our course don't throw complex maths at you, but focus on building up your intuition for infinitely better results down the line.



Please visit our website and subscribe to our email list to be aware about our free courses and promotions. Get in touch with us at [academy@aisciences.net](mailto:academy@aisciences.net) for more details.

# Preface

*“For me, data science is a mix of three things: quantitative analysis (for the rigor necessary to understand your data), programming (so that you can process your data and act on your insights), and storytelling (to help others understand what the data means).”*

**—Edwin Chen, Data Scientist and Blogger**

The overall aim of this book is to give you a beginner overview in Python.

## Who Should Read This?

Python and programming can be intimidating especially those with zero background. Good news is you can still learn it with the proper approach and constant practice (this is really important).

No matter where you’re coming from (major in math, physics or just a curious being), you can start to learning programming if you continue reading this book. You don’t actually need a strong mathematical background or a very powerful computer. What you really need is just a dedicated time to work on this material and practice on the exercises.

Programming is just like any other skill wherein you can learn it through constant practice (ideally every day). Most concepts might not make sense at first. But I promise you, if you continuously practice you’ll eventually get the “feel” of programming. Somewhere along the way something will suddenly click that would make everything clear.

## How to Use This Material?

The goal is to make you good enough in Python and programming so you can create simple programs on your own. Another goal is for you to take advantage of Python for data analysis (and other related exciting opportunities such as machine learning and artificial intelligence).

To accomplish this, it's recommended to follow through the following tutorials and dedicate an hour or two each day to complete each chapter (may or may not include the exercises). This approach is far more effective than consuming the whole material in one sitting.

There will be exercises at the end of almost every chapter. This is to solidify your understanding of the concepts of programming in Python. You can always look up the solutions if you're really stuck. You can also skip the exercises and come back to them later.

The order of this material is increasing complexity. We build up from simple operations and then combining them to create simple programs that require several lines of code. Many of the programming concepts require some maths. That's because in many cases, solving a math problem is one of the best ways to illustrate the power of programming.

For example, you can add the numbers from 1 to 100 using a few lines of code:

```
sum = 0
for num in range(1, 101):
    sum += num
print(sum)
```

The result will be 5050. Notice that we didn't perform long calculations by hand. Just a few lines of code and we're able to do a possibly long task. This is just one illustration on the power of programming. You can see more examples that better show how to use Python to automate tasks and possibly handle multiple operations fast.

At first the code above seems intimidating. But Python is actually intuitive. Just by reading through the code you can already get an idea of what it does. In the above example, we initiate the sum to zero and add the numbers from 1 to 100 to the sum (101 is not included because it's how Python works). We then "print" the sum so we can see the output.

In the terminal (the place where we execute code) it might look something like this:

```
>>> sum = 0
>>> for num in range(1, 101):
...     sum += num
...
>>> print(sum)
5050
```

That white font and black background add a nice touch to what we're doing. It makes us feel that we're finally doing some programming and software engineering.

Well, it's a simple example. Later on we'll be dealing with dozens of lines of codes. These could be overwhelming at first because the blocks of code work together to create a final output (or a series of outputs).

To get the most out of this learning material, always try to understand what the blocks of code really do. In many cases

Python is already intuitive and you can understand even what others wrote because of the simple and clear syntax (in contrast to Java, C, Lisp, and other programming languages).

Repetition is also the key. As we build up the concepts, it could be difficult to track which goes where and how each concept fits into the whole. That's why as we make the programs more complex, we'll review or briefly mention what the previous were. This way, we can reinforce your learning and make it easier for you to catch up.

But first, let's try to motivate you in learning Python and programming. This way you'll know the possible incentives and possibly get a big picture of how Python will improve your work.

### **Why learn Python & programming?**

Many people actually studied programming out of curiosity or they just want to execute their own ideas (e.g. the dream of having a revolutionary startup, building the next Google or Amazon). In fact, many programmers even in huge corporations are only self-taught. They don't have a computer science degree. They just bought or borrowed books, followed online tutorials or got a mentor to set their path.

However, it doesn't mean it's easy. You might need hundreds of hours of practice before you can call yourself a decent programmer. Also, you'll make tons of mistakes along the way. You might be then using Google far more often because of the weird errors your terminal or program makes. You'll search for answers (StackOverflow is a good resource) and it's likely you'll find good answers. Other learners and programmers were

likely to have been encountered and solved the problems you're facing now.

For some this commitment to time and effort will be very discouraging. Good news is this could also challenge you to take on the enormous task. This book will be the starting point (and it's enough to challenge you already). And when you strive for greatness, programming becomes more fun and stressful. You'll finally understand how some programmers wake up in the middle of the night with a knot in their gut.



© Copyright 2017 by AI Sciences  
All rights reserved.  
First Printing, 2016

Edited by Davies Company  
Ebook Converted and Cover by Pixel Studio  
Published by AI Sciences LLC

ISBN-13: 978-1719247221  
ISBN-10: 1719247226

The contents of this book may not be reproduced, duplicated or transmitted without the direct written permission of the author.

Under no circumstances will any legal responsibility or blame be held against the publisher for any reparation, damages, or monetary loss due to the information herein, either directly or indirectly.

**Legal Notice:**

You cannot amend, distribute, sell, use, quote or paraphrase any part or the content within this book without the consent of the author.

**Disclaimer Notice:**

Please note the information contained within this document is for educational and entertainment purposes only. No warranties of any kind are expressed or implied. Readers acknowledge that the author is not engaging in the rendering of legal, financial, medical or professional advice. Please consult a licensed professional before attempting any techniques outlined in this book.

By reading this document, the reader agrees that under no circumstances is the author responsible for any losses, direct or indirect, which are incurred as a result of the use of information contained within this document, including, but not limited to, errors, omissions, or inaccuracies.

*To my wife Jannet Danboard.*

## A Quick Example

```
>>> for num in range(1, 101):  
...     if num%2 == 0:  
...         print("Even")  
...     else:  
...         print("Odd")  
...  
...
```

```
for num in range(1, 101):  
    if num%2 == 0:  
        print("Even")  
    else:  
        print("Odd")
```

Can you guess what this code does? As with the earlier example, it will “work on” the numbers from 1 to 100. The difference is that instead of adding the numbers, it will check if a certain number is “Odd” or “Even” (divisible by two).

The line `if num%2 == 0` means if the number is divided by 2 and the remainder is zero, print “Even.” Otherwise, print “Odd” (what else is there?). This simple example is actually one of the building blocks of many complex programs for today. For example, think of how email login works. If what you typed in as your password matches the one saved in their database, you get access to your account. Else, you’ll be told to try again or even get blocked from accessing your account.

This if-else statement will be very useful whenever there’s a “fork” in the process. Will the program take this particular path or the other one? What happens if we choose that path?

By the way, the result of the above program will be this:

```
Odd
Even
Odd
Even
Odd
Even
```

And so on until it prints out the result for 100 (which is “Even”). We can take this a step further by adding the Even numbers in a list and finding out how many of them are. We can do this through the following:

```
even_list = []
for num in range(1, 101):
    if num%2 == 0:
        print("Even")
        even_list.append(num)
    else:
        print("Odd")
print(even_list)
print("Counting even numbers: ")
print(len(even_list))
```

The result will be like this:

```
Odd
Even
Odd
Even
Odd
```

```

Even
Odd
Even
...
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28,
30, 32, 34, 36, 38, 40, 42, 44,
46, 48, 50, 52, 54, 56, 58, 60, 62, 64, 66, 68, 70,
72, 74, 76, 78, 80, 82, 84, 86, 8
8, 90, 92, 94, 96, 98, 100]
Counting even numbers:
50

```

Let's review the code:

```

even_list = []
for num in range(1, 101):
    if num%2 == 0:
        print("Even")
        even_list.append(num)
    else:
        print("Odd")

print(even_list)
print("Counting even numbers: ")
print(len(even_list))

```

First, we initiate an empty list (`even_list = []`) where we'll put the list of Even numbers. Next is we run through those numbers and check if each one is Odd or Even. If the number is Even (if `num%2 == 0`), we print "Even" and add that number into our list (`even_list.append(num)`).

We can also count the even numbers or the number of items in our `even_list` (`len(even_list)`). We do this by using `len` which means length.

Read again the code and see if it finally makes sense. The Python syntax there is intuitive. Just by reading the code you can already get an idea of what it does and what are the possible outputs.

It's good practice to try to make sense of the code first before reading the explanation. This way you'll build your intuition on programming.

Aside from developing your intuition, learning Python will also make you think in more structured ways. For instance, let's reverse the above example and start with the problem.

The problem is how to create a list of Even numbers from 1 to 100 and then count how many are there. The steps that might be required to solve this problem are:

1. Create an empty list where we can put the even numbers.
2. Check each number from 1 to 100 if it's Odd or Even.
3. If it's Even, add the number to the list.
4. Once we've checked the each number, print the list.
5. Then, check the number of items (count even numbers).

Once we've outlined the steps, it's now time to translate that into code. This is where knowledge in Python and programming comes in. You translate the problem and each step into statements the computer will understand (and be able to execute).

It's a way of thinking just like translating word problems into algebraic expressions we can solve. It's similar to translating a language so the other person can understand what we're saying. In our case though, we're translating our problem into something the computer can understand and execute.

## **The Big Picture: Using Python for Data Analysis**

There are dozens of programming languages out there aside from Python. But it's good to start with Python because it has clear and intuitive syntax. Also, many universities actually use Python in teaching introductory computer science courses.

Python is also used in web development, software development and even in various numeric and scientific applications. You can use Python to perform advanced data analysis. In fact, one reason Python became more popular recently is because of data science and machine learning.

You can find tons of resources to applying Python in data science (the intersection of data analysis, programming, and domain knowledge). In machine learning (ability of the machine to learn from data and experience, and then improve performance), Python is also widely used.

When you get distracted and sidetracked from reading this book (we're sure you will be), you will read conflicting ideas and recommendations that relate to Python and programming ("use R because statisticians use it", "Java is better", "data science and machine learning will be democratized so no need to learn programming").

You can always explore other paths later. But first, try to finish this book. You don't have to understand everything. Through the weeks and months, bit by bit the concepts here will make sense. Also, you'll eventually understand how useful Python is in many applications especially in numeric and scientific fields.

## **Mastering the Basics**



I bet you're excited to perform cool and advanced stuff using Python. Unfortunately, we won't start there. It will take a while before you can get to build cool stuff. But you'll eventually get there if you finish this book and learn more as you go along.

First, we should set up our tools so we can write code and execute it. You will download and install modern tools that modern programmers use every day. This is important because using quality tools will make learning much easier.

Second, we'll study the basics so we can form a solid foundation in Python. We'll explore how Python works and what can you do about it. Aside from developing your skill, mastering the basics will also give your ideas on what's possible when using Python and in programming in general.

Now let's start setting your computer so you can begin programming.

# Python Setup

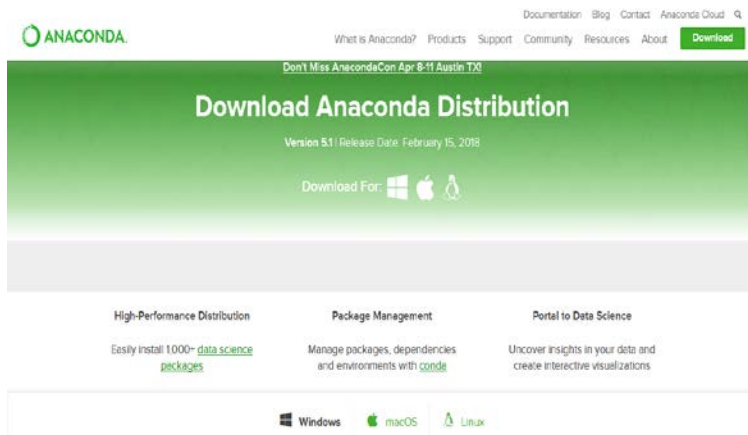
## Overview

The goal here is to install tools that will allow you to write code and enable your computer to read and execute Python.

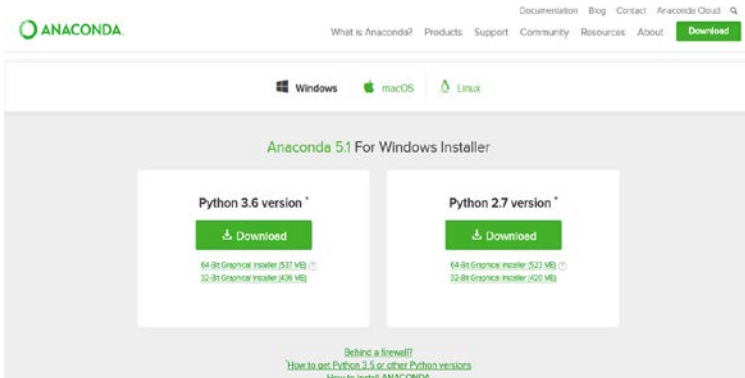
First is we'll download and install Anaconda. This way, your computer will be able to read and execute Python code when you write it. Second is we'll install Atom (by Github). We'll use this as our text editor so you can write code. We can also use its Terminal so we can run our program.

## Download & Install Anaconda

Let's start with setting up Anaconda in our computer (it's free). Go to Anaconda download page (<https://www.anaconda.com/download/>). You'll see something like this:



You can choose an OS (Windows, macOS, or Linux) and download the appropriate package.

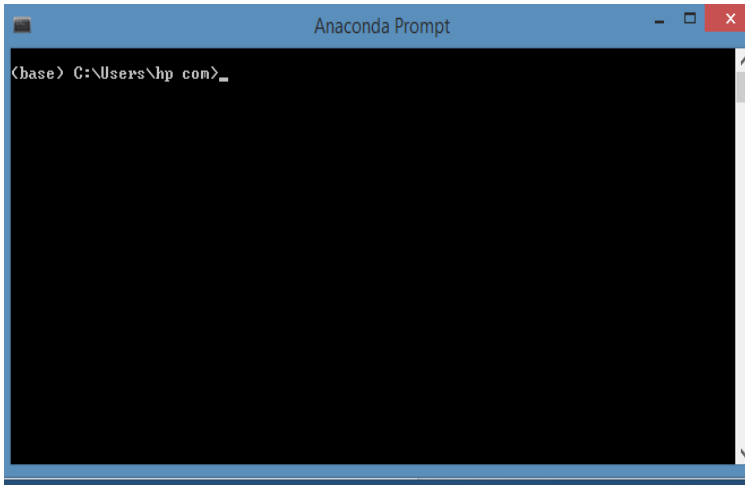


Choose the Python 3 version and click the green Download button. Downloading will take a while (400+ or 500+ MB file).

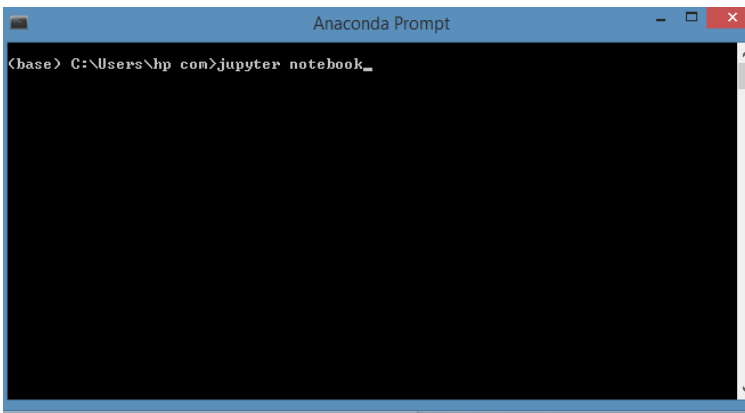
When the download's finished, you can just do a standard installation. Double click the file and click next until everything's done. For now it's important that you install it for one User only (there are issues in multi-user installation). If there's an error, you can Google search and find out the solution (many other users have likely encountered your problem before).

## Open Jupyter Notebook

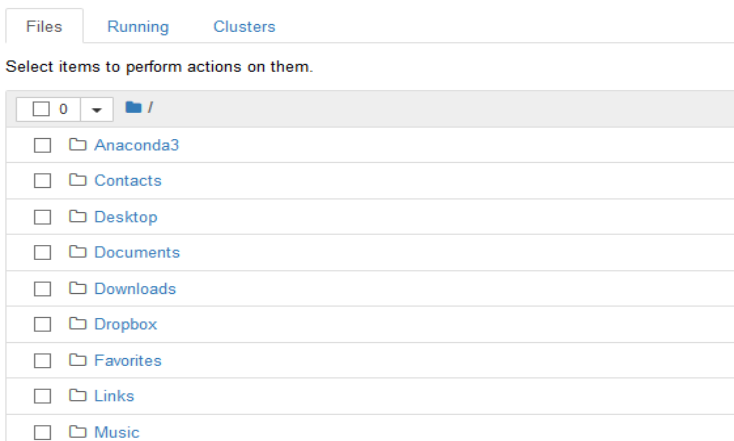
Let's then try if our installation really worked. In your Start menu search for Anaconda Prompt. When you click and run it, it should look something like this:



Next is type jupyter notebook next at the blinking underscore:



Press Enter and then wait for your default browser (or a new browser tab) to open. The result will be like this:

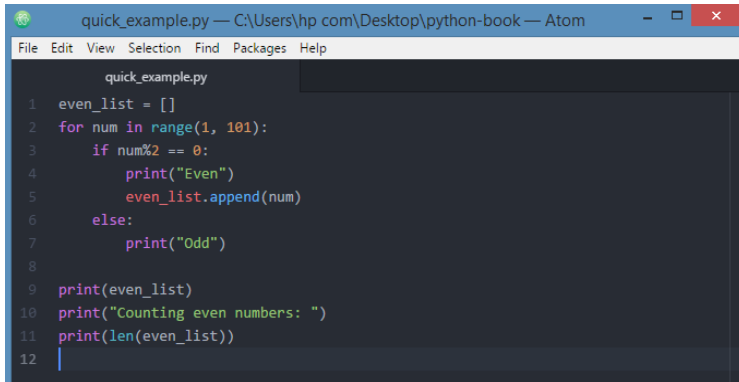


Now we're certain that our installation worked. Later on we'll explore more about Jupyter Notebook. It's very useful when performing data analysis and keeping track of what we're doing. Many instructors use Jupyter Notebook because it's truly a notebook wherein you can execute code, analyze data, include text and notes, graph data, and more (all in one place).

## Install Atom & Download Packages

Let's exit Jupyter Notebook by pressing CTRL + C in the Anaconda Prompt (this is the proper way to stop Jupyter Notebook). You can then close the browser tab where the Jupyter Notebook opened.

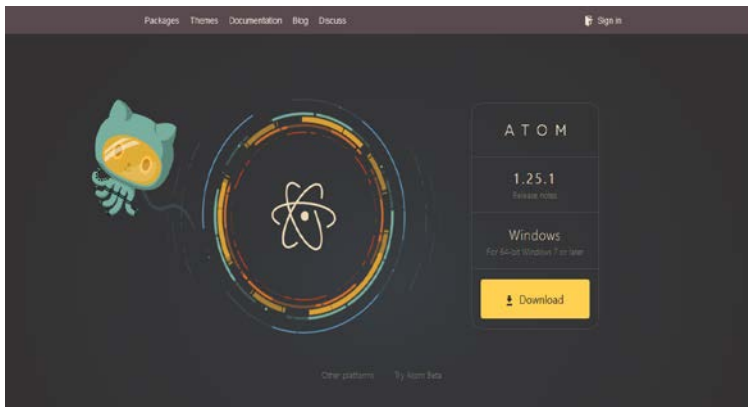
In the first part of this book we'll heavily use Atom instead of Jupyter Notebook. Atom is a text editor where you can type the code (with nice formatting that makes you feel cool). For instance, if we type our earlier example code in Atom, it will look something like this:



```
quick_example.py
1 even_list = []
2 for num in range(1, 101):
3     if num%2 == 0:
4         print("Even")
5         even_list.append(num)
6     else:
7         print("Odd")
8
9 print(even_list)
10 print("Counting even numbers: ")
11 print(len(even_list))
12
```

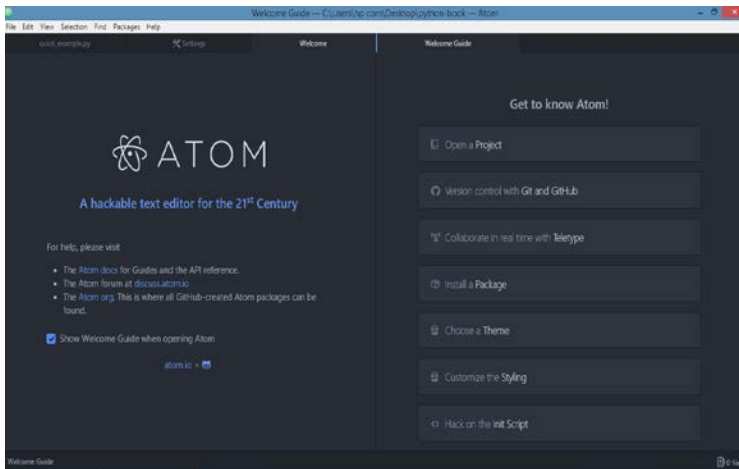
Notice the different colors and black background. Atom is one of the modern tools for programming and more and more users are switching to it for their web and software development projects.

To download Atom, visit their page (<https://atom.io/>) and click the Download button (I think the page auto-detects your computer OS).

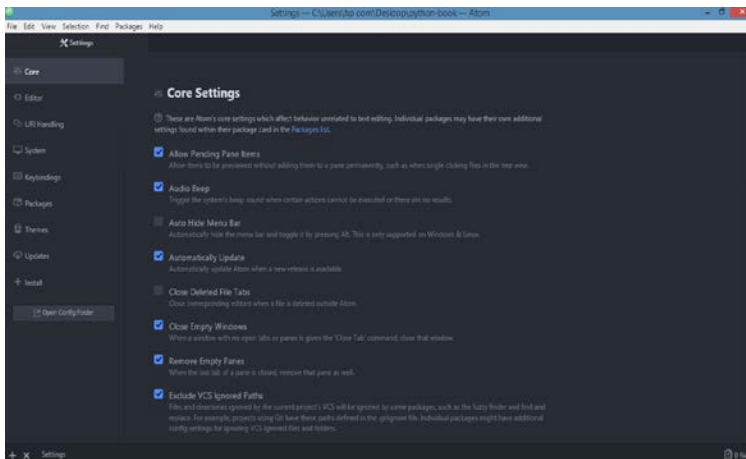


It's a 100+ MB file and once the download's done, do the standard installation (click Next until finished).

We're not done yet. Open Atom and you'll see this:



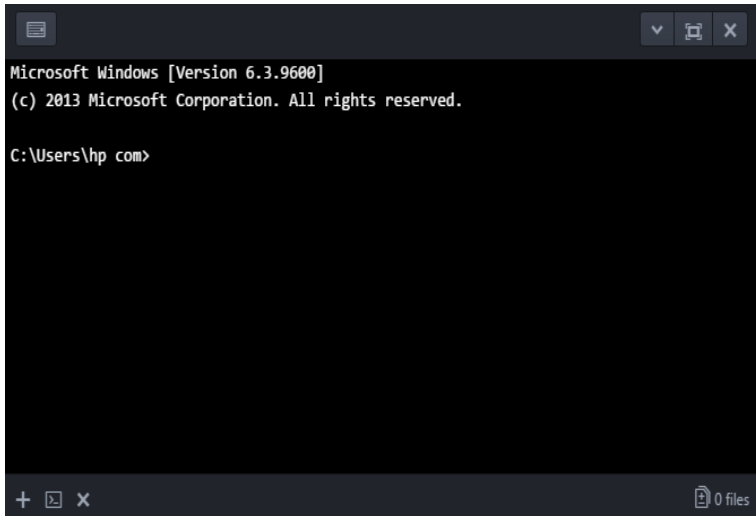
Click File and then go to Settings.



Click + Install (in the left side) and at the top you'll see Install Packages. In the search box type "platformio ide" and then click Install at the top search result (1M+ downloads). This is for adding a "Terminal capability" into our Atom text editor (where we can run and execute code without opening another application).

After installing a package inside the Atom. Close Atom and open it again (to make sure changes have taken effect).

The Welcome Page may appear again. At the bottom right corner, notice there's a little plus (“+”) sign there. Hover it and it will say “New Terminal” hinting you that when you click it, it will open a Terminal for us.



You can open it anytime by clicking the plus sign (or exit by clicking the X sign at the top left).

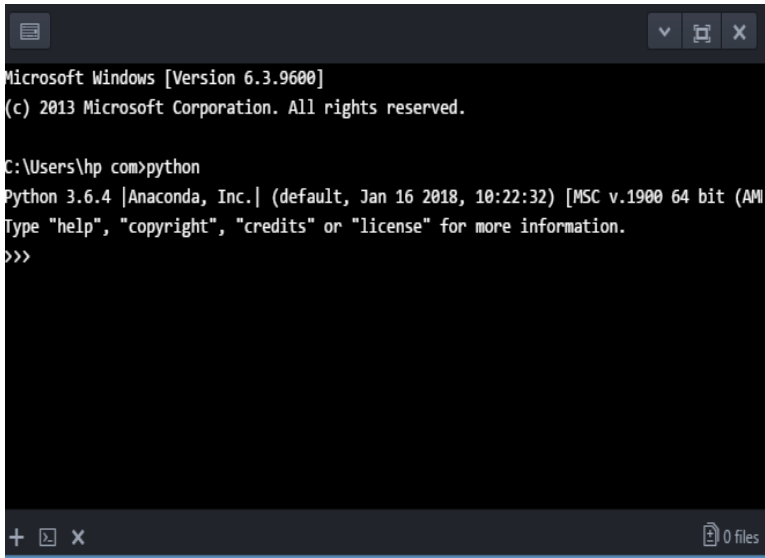
Aside from Platformio IDE, it's also good to install autocomplete-python. Go again to File, Settings, +Install, and search for “autocomplete python.”

Those two packages will make your life a bit easier in learning how to program. Also, you'll have more focus on the programming itself than tweaking the tools.

## Opening & Running the Terminal



Let's try working on the terminal and get your feet wet on programming. Open the terminal inside Atom (click plus sign at the bottom right corner). Once the terminal is ready, type "python" on the blink and then hit Enter. You'll see something like this:

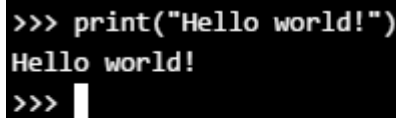
A screenshot of a Windows command prompt window. The title bar says "Microsoft Windows [Version 6.3.9600]". The text inside shows the command prompt path "C:\Users\hp com>python" and the output "Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)] Type "help", "copyright", "credits" or "license" for more information." followed by a prompt ">>>". The window has standard Windows controls (minimize, maximize, close) in the top right and a taskbar at the bottom showing a plus sign, a file icon, and "0 files".

```
Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\hp com>python
Python 3.6.4 |Anaconda, Inc.| (default, Jan 16 2018, 10:22:32) [MSC v.1900 64 bit (AMD64)]
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

This is now your interactive shell where you can write and run Python instructions. Let's try printing "Hello world!" Type this into the shell:

```
print("Hello world!")
```

A screenshot of the Python interactive shell. It shows the prompt ">>>" followed by the command "print('Hello world!')", the output "Hello world!", and another prompt ">>>" with a cursor. The background is black and the text is white.

```
>>> print("Hello world!")
Hello world!
>>> 
```

There you go you just got a taste of Python. To exit the Python interactive shell, just press CTRL + Z and then hit Enter. To open it again type the “python” on the blink.

## Common Mistakes

In programming it's very common to experience errors even during the installation. In fact, even experienced programmers encounter errors from time to time.

Prepare yourself because you're likely to encounter countless errors while learning how to program. One such error is actually a typographical one. Perhaps you just forgot to add double quotes or a closing parenthesis. Or you forgot to type “python” before the filename when running the program (we'll discuss this later when running Python programs).

Thankfully, you can still quickly solve the problem by doing some research or double checking your code. Most of the time though it's a typo error or the structure of the code is wrong. Eventually, you'll be able to avoid those common errors altogether as you gain more experience.

## How to Fix Errors

In some cases though, the errors seem incomprehensible and cryptic (somewhat mysterious, but programmers prefer using cryptic). Let's open the Python interactive shell in Atom and type the following:

```
print("Hello " + 123)
```

Press enter and you'll get this:

```
Traceback (most recent call last):  
  File "<stdin>", line 1, in <module>
```

```
TypeError: must be str, not int
```

What just happened? You must be expecting “Hello 123” to come out but instead it displayed something you’ve never seen before. To solve this, you can copy-paste the error message and paste it into Google search bar. Most likely one of the top results is from StackOverflow. This is where lots of programmers answer questions and find answers to their programming problem (as of this writing, you don’t have to sign up to view the answers).

The problem in the code above is that we’re trying to combine a string with an integer (we’ll discuss these later). They should be both strings so we can combine them. We can make the code work by typing this into the Python interactive shell:

```
print("Hello " + "123")
```

Or better (by converting 123 into a string):

```
print("Hello " + str(123))
```

Hello 123 will be the output.

Remember to use Google and StackOverflow whenever you’re stuck. But first check for any typos before searching answers online. This is good practice just like proofreading what you wrote before hitting Publish to ensure there are no grammatical errors.

## Summary & Review

In this chapter we made you ready to write and run Python programs by installing Anaconda and Atom (including the packages Platformio IDE and autocomplete-python). You’ve

also got a taste of running the Python interactive shell in the Terminal.

In the next chapter let's discuss what is Python so you'll be better enlightened on what can you do with it. We'll also provide you an overview of what you're about to learn in the succeeding chapters. Let's start.

# What is Python

## What is Programming?

Programming is just creating and sending instructions to the computer. Then the computer will act on those instructions.

The basic principle is a lot like using a calculator. You enter  $3 + 3$  and the calculator (or computer) will give you back 6. You sent instructions and the calculator performed a task.

However, your instructions should be in the form that the computer will understand. This is why Python is a programming **language**. It's like communicating to a computer through a language that you both can understand.

It's a simplistic explanation but it's enough to get us going. The explanation doesn't have to be complicated. We'll be reserving our minds to more important topics such as doing data science on huge datasets or building complex web apps that will change the world.

## How Useful It Is Really?

For now, forget all about the other programming languages you've heard of. Just focus on Python because you can always learn others later.

Anyway, how useful is Python? Google, YouTube, Dropbox, Reddit, Instagram, and other popular sites have Python in them. In one form or another, Python is being used behind the scenes in many other websites. It works on the background to act on the user's actions and then output something relevant.

Of course you'll still learn other languages as you dive deeper into programming. Also, many programming jobs actually require a knowledge of a dozen programming languages and technologies. The good thing here is what you learn from Python will be easily transferable once you start learning other programming languages.

Yes there will be differences and other quirks. But the most important thing is you learn the concepts of programming including how to translate problems and solutions into lines of code that your computer can execute.

### **Why Many Universities are Teaching Python**

Even the Massachusetts Institute of Technology (MIT) has a course Introduction to Computer Science and Programming being taught in Python. Other universities followed (or are already teaching Python for introductory computer science courses).

One reason is the Python's syntax which makes it easier for beginners to focus on the programming concepts instead of the quirks. Another reason is Python just got there first because there are other programming languages that offer the same simplicity and clarity.

Whichever is the case, there's a wealth of other resources you can use as reference. Most likely someone else has already asked the question you have in mind that relates to Python. Also, the thriving Python community is always ready to help whenever you get stuck.

### **Clarity & Intuitiveness**

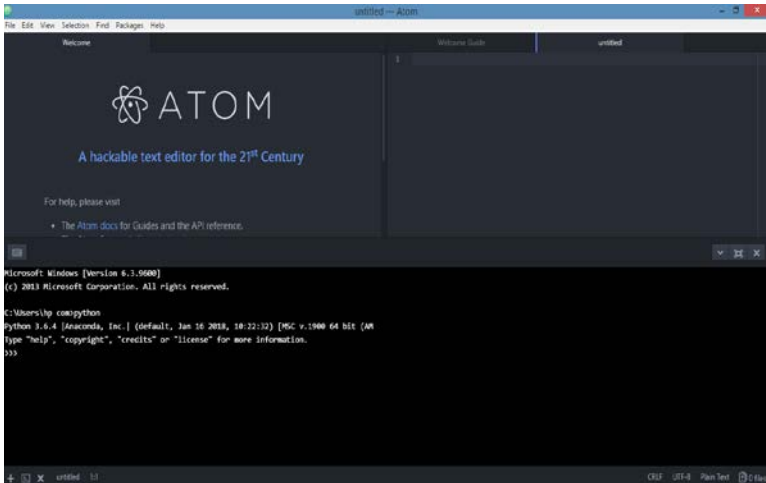
Again and again we're always mentioning these advantages of Python. That's because Python's clarity can make you focus on understanding programming instead of the nuances of computer science.

Later you can always learn other programming languages and discover for yourself the true strengths of Python (and perhaps how other languages are superior). But for now, let's focus on Python itself and finally take a deeper dive into programming (especially if you've dabbled on programming earlier but no continuity).

# Python Level One

## Using the Interactive Shell

Let's open Atom, run the Terminal (click the + sign bottom left corner), and open the interactive shell (type Python). What appears should be similar to this:



Now we can start typing beside the `>>>` and hit Enter to run simple commands. Let's start again with using the print statement.

```
>>>print("I am awesome.")  
I am awesome.  
>>>print("I can do this.")  
I can do this.
```

Notice that after hitting Enter the output immediately appears. Let's play some more but this time it's a bit different.

```
>>>print I am awesome.
```



```
File "<stdin>", line 1
    print I am awesome
      ^
SyntaxError: Missing parentheses in call to 'print'.
Did you mean print(I am awesome)?
```

It's a syntax error which means execution of the line failed because there's something wrong with how you wrote it (e.g. no parenthesis, no quotation marks, etc.). It's like the computer didn't fully understand your instruction because there's a "grammatical error."

Good thing here is that Atom provided a suggestion on how to solve the problem. We can then try this instead:

```
>>>print(I am awesome)
File "<stdin>", line 1
    print(I am awesome.)
      ^
SyntaxError: invalid syntax
```

Another SyntaxError again. This just shows that programming tools are not yet perfect. We should still rely on our knowledge and skills when it comes to spotting errors and creating programs.

Anyway, how do we get rid of that error and make the code run correctly? Here's the solution:

```
>>>print("I am awesome.")
```

Enclose the statement to be printed in parenthesis. Also don't forget the quotation marks so your computer can interpret it correctly.

As before, you can exit the shell by pressing CTRL + Z then hitting Enter.

## Simple Operations Including Maths

Aside from printing statements, we can also do simple math operations (just like using a calculator) using the Python interactive shell. Let's open again the shell by typing "python" then try adding 3 and 4

```
>>>3 + 4
7
```

We can also do it with no spaces between them:

```
>>>3+4
7
```

Let's try more math operations:

```
>>>3*4
12
>>>12/3
4
>>>12%3
0
>>>12%7
5
```

What happened and what did the percent sign (%) do? It's actually called the modulo operator which simply returns the remainder. In the above example, what's the remainder of 12 divided by 7?

Let's play some more. Guess what happens if we do this?

```
>>>2**3
```

The answer is 8. How? It's like putting an exponent of 3 to 2. It becomes 2x2x2.

What about the order of operations? Python follows what we've learned from school. For example, the answer will be 10 for the following:

```
>>>2*3+4
```

Or

```
>>>4+2*3
```

It follows MDAS (multiplication/division then addition/subtraction). But if we put a parenthesis to make addition a priority, we do this by:

```
>>>(4+2)*3
```

```
18
```

Next is let's try this:

```
>>>4 + "I am awesome."
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
TypeError: unsupported operand type(s) for +: 'int'  
and 'str'
```

It returned an error because we can't add an integer and a string (we'll discuss more about this later). In Python it simply doesn't make sense. We can make it work though by doing this:

```
>>>str(4) + "I am awesome."
```

```
'4I am awesome.'
```

We made it work by converting 4 into a string. This way we can add them together and combine into a single statement.

## Python Data Types (Integers, Floats, and Strings)

We learned that we can't combine integers and strings unless we make them the same data type. Aside from strings and integers, another data type is the float. We can easily know their differences by viewing some examples:

String ("I am awesome.", "You are great.", "Python")

Integer (-3, 0, 5, 301)

Float (3.14, 10.99, 77.77)

They are of different data types which means they can't be combined into a single value. Unless we apply a change (e.g. `str(4)` to make 4 a string), the code won't run correctly. It's also the case when combining a float and an integer (although they're both numbers).

```
>>>3 + 3.0
```

```
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
```

```
KeyboardInterrupt
```

But

```
>>>3.0 + 3.0
```

```
6.0
```

The latter worked because they're both floats. We can transform 3 into a float by doing this:

```
>>>float(3)
```

3.0

Hence,

```
>>>float(3) + 3.0  
6.0
```

Aside from numbers, we can also work on letters and words. Let's type the following:

```
>>>"I think."  
'I think.'  
>>>'I think.'  
'I think.'
```

Notice it's the same output whether we use a double or single quote. We can use them interchangeably but be careful if one is enclosed over the other. Here's an example:

```
>>>"She said, 'I am the best.'"  
"She said, 'I am the best.'"
```

It worked perfectly because we enclosed the whole thing with double quotes. We can also do this:

```
>>>'She said, "I am the best."'  
'She said, "I am the best."'
```

But if we're missing a single or double quote, we might see an error:

```
>>>'She said, "I am the best."  
SyntaxError: EOL while scanning string literal
```

So if you're using multiple quotes or a combination of single and double quotes, just make sure you're properly enclosing the whole thing.

## What Can You Do with Strings

Aside from printing words and statements, you can also do other things with strings such as joining them. The right term for that is concatenating strings. Let's do that.

```
>>>'I am' + 'awesome.'  
'I amawesome.'  
Let's add a space to make it neat:  
>>>'I am' + ' awesome.'  
'I am awesome.'
```

We can concatenate strings by adding a plus sign between them. But what about saying something 4 times? Like Python Python Python Python?

```
>>>4 * 'Python '  
'Python Python Python Python '
```

Wait. We discussed earlier that we can't combine two data types (integer + string). So how it's possible that when we multiply a string by n, the string appears n times?

Here's the explanation. Concatenating a string with an integer doesn't work. On the other hand, the \* sign is called replicating the string. It's being used as a string replication operator. That's it for now because it's just how Python works.

In the future a few concepts might seem unclear and inconsistent. There will be those nuances that may shatter your

learning foundation. Don't worry though. Even experienced programmers make those mistakes and confusion. The key is just moving forward and it will all make sense later (or not at all but it still works).

## Summary & Review

In this chapter we've got some practice using the Python interactive shell. We've played around by working on integers, floats, and strings. We did some basic math operations including the use of the modulo operator (`%`, which returns the remainder).

We've also discovered a few limitations when it comes to working with different data types. We can't concatenate a string and integer (but we can replicate a string *n* times). We can't add an integer and a float (but we can convert an integer into a float and then add them together).

In the next few chapters we'll further discuss how to work on strings, floats, and integers. Review this chapter as you see fit or whenever you get stuck on the new concepts

## More Examples & Exercises

Try to figure out what happens on the following:

```
>>>5 * 'math '  
>>>3**2  
>>>3 % 2  
>>>3/2  
>>>'I am number ' + '9'  
>>>'I am number ' + str(9)  
>>>'I am number ' + 9
```

Try to guess the answer first before typing them into your Python interactive shell. You can also play around further and test things yourself.



## Python Level Two

In the previous chapter we've covered some of the basics including working on strings, integers, and floats. We did some simple math operations and played some in the Python interactive shell.

In this chapter let's take it a step further and create a real program. Yes, we'll write some code, save it, and then run it. We'll start with something simple (a few lines of code) and then as the chapter progresses, we'll make it a bit longer and more complex. Let's start by reviewing the print statement because this is essential in the succeeding lessons.

### Print Statements

```
>>>print('This is Python Level Two.')  
This is Python Level Two.
```

```
>>>print(3*3)  
9
```

```
>>>print(3)  
3
```

```
>>>print('I am number ' + str(9))  
I am number 9
```

Why use the print statement? It's one way to know that our program runs correctly (gives the correct output) when we save our code and run it. Some programmers also use it a sanity

check to see that each step works fine before extending their code or adding more functionalities to it.

Later we'll see how useful this is when we save a program and run it.

## Assigning Variables

We can store a value in a variable. Aside from that, we can make that variable interact with data types and fellow variables. We can see it more clearly by looking at examples:

```
>>>x = 9
>>>x
9
```

We assigned a value of 9 to x so when we call x, it has the value of 9. Let's make some more variables:

```
>>>myname = 'Thon'
>>>myname
'Thon'
>>>mylanguage = 'Python'
>>>print(mylanguage)
Python
```

We can do more with variables by letting them interact with one another.

```
>>>x = 3
>>>y = 4
>>>x + y
7
>>>x * y
```

```
12
>>>y % x
1
```

We can also perform math operations on variables:

```
>>>x * 3
9
>>>x + 9
12
>>>y * 3
12
>>>x + y + 3
10
```

What happens if we apply a string replication operator?

```
>>>x * 'Python '
```

You guessed it right. The result will be 'Python Python Python '

Variables are very useful when you'll be using their values repeatedly throughout the program. It's a convenient way to store values so you can access them whenever you need them.

Also, variables may vary (the root word of variable) as the program progresses. You can change the assignment anytime. For example,

```
>>>x = 3
>>>x
3
>>>x = 9
>>>x
9
```

You can also do this:

```
>>>x = x + 1
>>>x
10
```

What just happened? We know that 9 is the assigned value to x. But as said earlier, we can easily change that value. What we did is that we're assigning a new value to x by adding 1 to it.

We can also accomplish that in a somewhat shorter version. First, let's again assign 9 to x:

```
>>>x = 9
>>>x += 1
>>>x
10
```

The latter is shorter and looks cooler. You can choose either as long as it's clear to you what it does. Many programmers choose the latter though because it's different and makes them feel more like a programmer.

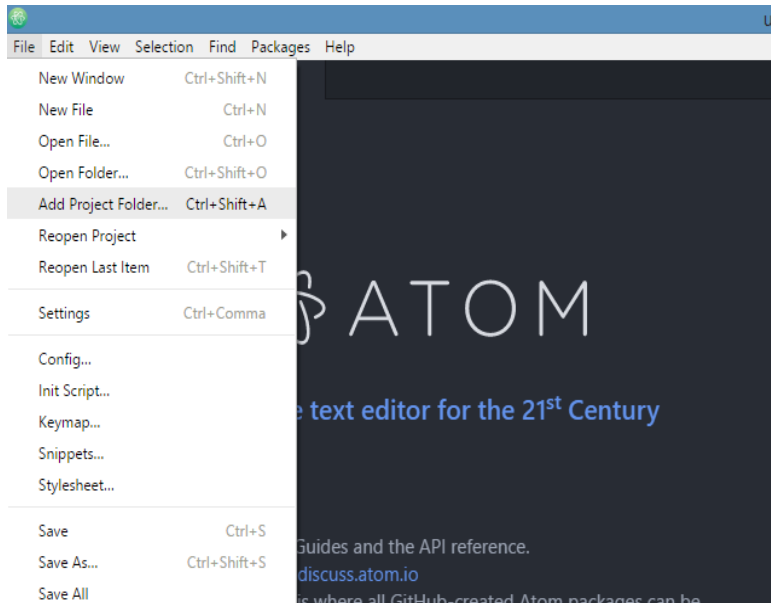
We can also concatenate strings assigned to variables. Let's do it.

```
>>>firstfood = 'eggs'
>>>secondfood = 'ham'
>>>thirdfood = 'bread'
>>>firstfood + secondfood + thirdfood
'eggshambread'
```

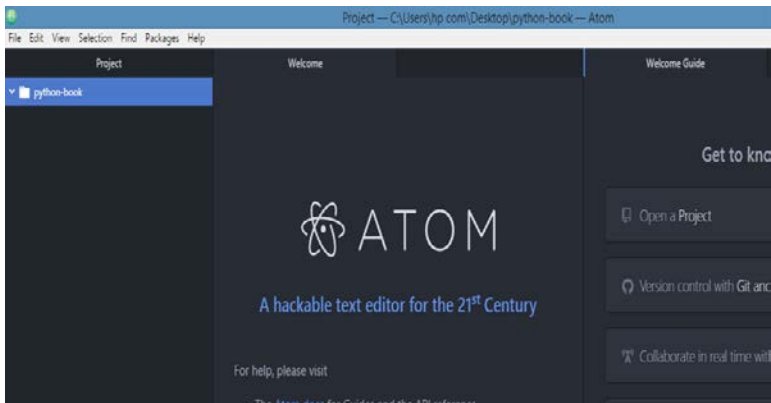
## Creating Your First Program

Seriously you actually know almost everything there is to create a very basic program. You already know what strings, integers, and floats are. You also know how to do math operations and print statements and values. Recently you've also learned how to assign variables and work with them (adding them together or reassigning them to new values).

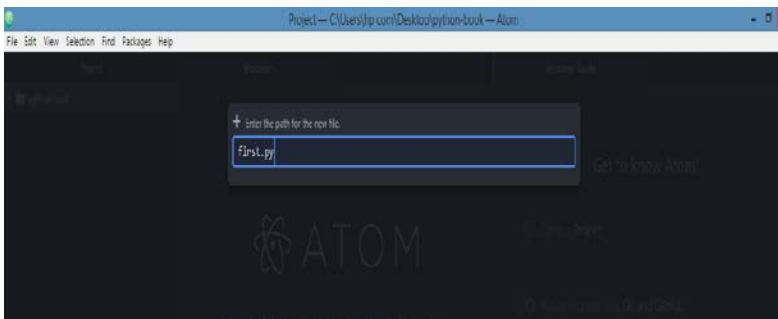
Let's apply them all together by saving our first simple program. Create a new folder in your Desktop and name it something like 'python-book' (or you can pick any simple name for it). Next is open Atom, go to File, and click Add to Project Folder:



Look for your newly made folder and click Select Folder.



The folder is empty because we haven't saved anything in it yet. To add a file, right click the 'python-book' in the Atom text editor and select New File. Enter the filename you want and add .py at the end.

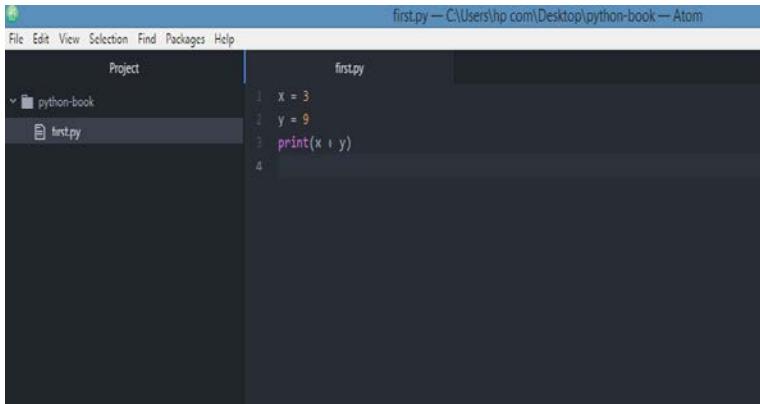


The .py extension is necessary so your computer will recognize that it should be run in Python. Almost all the files we'll be making here in this Part One of the book will require the .py extension.

Once you've typed that hit Enter and presto, you're now ready to type something into the first.py. Let's warm up first. Type this inside the first.py file and CTRL + S afterwards (to save your work):

```
x = 3
y = 9
print(x + y)
```

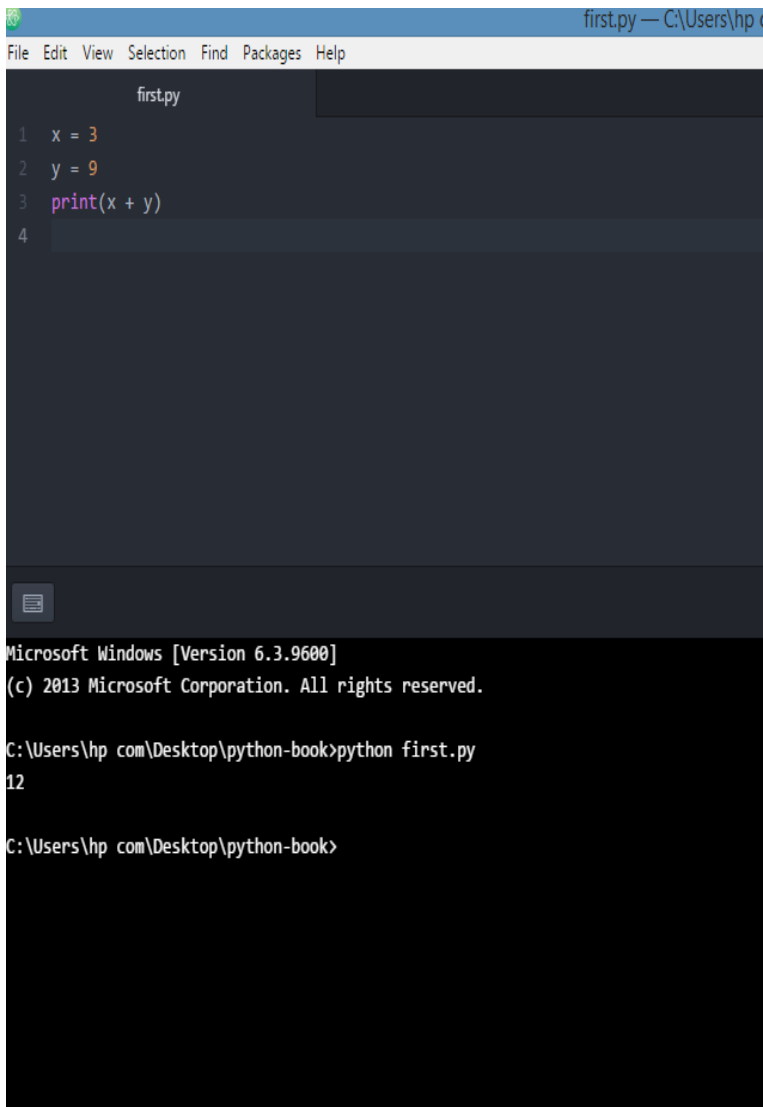
It will look like this:



Let's run this by opening our Terminal (click + sign at the bottom left corner). But don't type 'python' on the blink. That's because we'll be doing things differently this time.

If the Terminal is open now, type this and then hit Enter:

```
python first.py
```



```
first.py
1 x = 3
2 y = 9
3 print(x + y)
4

Microsoft Windows [Version 6.3.9600]
(c) 2013 Microsoft Corporation. All rights reserved.

C:\Users\hp com\Desktop\python-book>python first.py
12

C:\Users\hp com\Desktop\python-book>
```

What happened was that we ran your Python program and got a result (printed 12). Our computer understood that 3 and 9 are assigned to x and y respectively. We then requested for an output by saying `print(x + y)`.



That's it. We wrote some code and executed it by doing "python filename.py" and we got something. Well, what happens if we remove the print statement and instead just said  $x + y$ ?

```
x = 3
y = 9
x + y
```

Run again the program and nothing will appear. That's because we haven't said anything about displaying the result. Print statements are necessary in running Python programs this way so you can see something.

Let's delete what we wrote and start all over again. Then this time, let's create something longer and a bit more complex.

```
firstname = 'Py'
lastname = 'Thon '
multiplier = 2 * 3

print(multiplier * (firstname + lastname))
```

Type 'python first.py' again into the Terminal (with no quotes) and the result will be:

```
PyThon PyThon PyThon PyThon PyThon PyThon
```

Let's discuss how that happened. First, we assign values to the variables `firstname` and `lastname`. We also assigned a value to `multiplier` ( $2 * 3$ ). Finally, we want to display the result of applying the multiplier to the concatenation of `firstname` and `lastname`.

Let's try another one (delete all contents of `first.py` and start all over again). Type this into the file and then save your work:

```
myname = 'Thon'
age = 25
hobby = 'programming'

print('Hi, my name is ' + myname + ' and my age is ' +
      str(age) + '. Anyway, my hobby is ' + hobby + '.')
```

A screenshot of a Python IDE window titled 'first.py — C:\Users\hp.com\Desktop\python-book — Atk'. The window has a menu bar with 'File', 'Edit', 'View', 'Selection', 'Find', 'Packages', and 'Help'. The code editor shows the same code as the previous block, with line numbers 1 through 6 on the left. The code is: 1 myname = 'Thon', 2 age = 25, 3 hobby = 'programming', 4, 5 print('Hi, my name is ' + myname + ' and my age is ' + str(age) + '. Anyway, my hobby is ' + hobby + '.'), 6. The text is color-coded: strings are in quotes, numbers are in blue, and the print function is in purple.

The result will be:

```
Hi, my name is Thon and my age is 25. Anyway, my hobby
is programming.
```

Here's what happened. First, we again assigned values into the variables `myname`, `age`, and `hobby`. Then we do string concatenation by forming a statement. We combine the strings with the variables and then display the result.

Later on, you'll realize how useful this is when creating a basic program. You can assign values to variables and then combine them with something else. Later you'll also discover how to ask input from the user and use that in your own program.

## Comments for Your Own Notes

Comments make code more readable. That's because you can use human language to explain your code or describe what it does. Your computer will ignore (not run) the comments you made. It's just for you and possibly other people who will read your code. Here's an example:

```
# This is a comment.  
# Comments are ignored by the computer. They won't run.  
# A comment is a single line.  
# The goal here is to print numbers from 1 to 10 and  
tell if each one is even or odd.
```

Comments start with a hashtag (or a pound sign if you prefer it). Then anything proceeding that hashtag sign will be ignored by the computer.

This is very useful in keeping track of what you're doing. After all, codes seem cryptic. And believe it or not, even expert programmers forget what a block of code does after a few weeks or months of creating it.

Comments are useful for sanity checks. It's just refreshing to read human language once in a while.

## Asking the User for an Input

Create a new .py file and type:

```
print("What is your name?")  
myName = input()  
print("Your name is " + myName)
```

Run it in shell and you'll see this:

```
What is your name?
```

```
Mario
```

```
Your name is Mario
```

What happened? First, we printed the question “What is your name?” and then asked the user (you) for an input. That input is then saved to the variable `myName`. Finally, we printed “Your name is “ plus the value stored in `myName`.

Although we won’t use this much in this book (especially in actual data analysis), it introduces us to some of the possibilities in programming. We realized that we can ask the user for an input and store or use that value later.

## Summary & Review

In this chapter, we learned about printing statements, assigning variables, saving your program as a `.py` file, using comments, and asking the user for an input (and then storing or using that).

With just that knowledge you can create simple programs (and games) that do something useful. And once you’ve learned more programming concepts and possibilities, your programs will become more powerful. You’ll also be able to use those concepts to perform advanced data analysis.

## More Examples & Exercises

Create a `leveltwo.py` file and write code on it that does something like:

1. Asks the user his/her name.
2. Store that value into the variable `myName`.

3. Asks the user his/her age and then store that value in `myAge`.
4. Finally, print something like “My name is Mario and I am 25 years old.”

If you’re stuck you can review this chapter and other related material. Try solving it first before looking at the answer.

One possible solution:

```
print("What is your name?")
print("What is your name?")
myName = input()
print("How old are you?")
myAge = input()
print("My name is " + myName + " and I am " + myAge +
      " years old.")
```

This outputs when you run `python leveltwo.py`:

```
What is your name?
Mario
How old are you?
25
My name is Mario and I am 25 years old.
```

# Python Level Three

## Comparison Operators

You must be already extremely familiar with equal to, less than, and greater than. These indicate comparison between two values. In Python, we make comparisons by using the following:

Operator	Meaning
==	Equal to
!=	Not equal to
<	Less than
>	Greater than
<=	Less than or equal to
>=	Greater than or equal to

They evaluate to True or False (depends on the values you're comparing). Let's look at some examples by opening our Terminal so we can quickly test the operators. Open the Terminal in Atom by clicking the plus sign at the bottom left corner.

Then we open the interactive shell by typing python. When the >>> symbol appears, it's time to test what comparison operators do:

```
>>>8 == 8
True
>>>8 > 4
True
>>>8 < 4
```

```
False
>>>8 != 4
True
>>>8 != 8
False
>>>8 >= 2
True
>>>8 <= 2
False
```

If the comparison is correct, it returns `True` (otherwise it's `False`). We can also use comparison operators on strings:

```
>>>'hello' == 'hello'
True
>>>'cat' != 'dog'
True
```

Only the `==` (equal to) and `!=` (Not equal to) work on strings. `>`, `<`, `>=`, `<=` work on integer and floats.

Many beginners get confused with the `=` and `==`. The difference is that the first, `=`, is used for assigning values into a variable. On the other hand, the `==` is confirming if two values are equal.

## Boolean Operators (and, or, not)

You can do multiple comparisons in one line of code. Here are examples:

```
>>>8 > 3 and 8 > 4
True
```

What happened there? Both  $8 > 3$  and  $8 > 4$  evaluate to `True`, so the end result is `True`. It was made possible by the Boolean operator *and*. Let's use it again:

```
>>>8 > 3 and 8 > 9
False
>>>8 > 9 and 8 > 10
```

If you want the result to be `True`, make both the comparisons result to `True`. Later on we'll realize how useful it is especially when we've learned about `if`, `elif`, and `else` statements.

For now, let's further explore the Boolean operators. Earlier we discussed the *and* operator. Let's move on to *or*. In contrast to *and*, only one of the comparisons should be `True` for the evaluation to result to `True`. For example:

```
>>>8 > 3 or 8 > 800
True
>>>'hello' == 'hello' or 'cat' == 'dog'
True
```

It says if either A or B is `True`, the evaluation is `True`.

Another Boolean operator is *not*. It just returns the opposite of the supposed evaluation. We can see that here:

```
>>>not True
False
>>>not False
True
```

Let's explore it further:



```
>>>not 8 > 1
False
>>>not 8 > 800
True
```

Yes it's confusing and you might not find immediate use to it. Some programmers still use it though for special cases. It's good to get familiar with it because sooner or later you'll encounter it again if you get serious with programming.

## **If, Elif, and Else Statements**

In the past we've covered some of the basics. In this section we'll add more flavor and power to your programs through Flow Control (the if, elif, and else statements).

In the previous sections, it's all about a single method of executing an instruction. Is 8 greater than 4? What should we do then?

On the other hand, using Flow Control adds more possibilities. The program can skip an instruction, execute something, or do something else depending on the instructions and conditions you've set. Let's quickly look at a general example so you know what we're talking about:

```
myNumber = input()
if myNumber > 100:
    print("The number is greater than 100.")
else:
    print("The number is just less than 100.")
```

We also had an example way earlier about Flow Control:

```

for num in range(1, 101):
    if num%2 == 0:
        print("Even")
    else:
        print("Odd")

```

What happened there was we run the numbers from 1 to 100 and then check if each one meets *a certain condition*. If the number is divisible by 2 (zero remainder), print “Even.” Else, we print “Odd.”

Basically, Flow Control is also used in verifying if the login email and password typed by the user is correct. In a simplified way, it might look something like this:

```

print("What's your email?")
myEmail = input()
print("Type in your password.")
typedPassword = input()
if typedPassword == savedPassword:
    print("Congratulations! You're now logged in.")
else:
    print("Your password is incorrect. Please try again.")

```

What happened there was we’re verifying if the password typed by the user matches the one that is saved on our database. If it’s a match, we do or print something. If it’s not, we do or print something else.

We can also apply this in confirming if a certain number (perhaps entered by a user) is within a range. Type this into a new room.py file:

```

print("What's the temperature in your room at night
(in Celsius)?")

roomTemp = int(input())

if roomTemp <= 20 and roomTemp >= 15:
    print("Nice. It's easy to fall asleep if that's
your room's temperature.")
else:
    print("Perhaps it's too cold or too hot. Not
ideal.")

```

When asked for an input, try an integer such as 19 (or anything you like) just to see how it works:

For example:

```

>python room.py

What's the temperature in your room at night (in
Celsius)?
19

Nice. It's easy to fall asleep if that's your room's
temperature.

```

Try other values so you can understand how the program works. Basically, we use the value of a variable and verify or compare it. We then set conditions so the computer will know which specific instructions to execute based on if a certain condition is met.

## While Loops

Well, the previous lessons are a bit easy. This is where it really starts. Many people actually give up when they start encountering this part.

Good thing here is this what separates the curious from the dedicated ones. It may take some time before it all makes sense. But along the way something will click and realise it's really easy. That's why you just have to go on and repeatedly read and apply the lesson until you comprehend the concept.

Anyway, let's just get started so we can get some progress already.

Many times in programming, a block of code requires to be executed over and over as long as the condition holds True. It can run indefinitely until it meets or exceeds a certain condition. Let's illustrate this by creating a `whileLoop.py` file and typing in the following:

```
inbox = 0
while inbox < 10:
    print("You have a message.")
    inbox = inbox + 1
```

Three interesting things happen here. First, we set an initial value of zero to `inbox`. Then we created a while loop that executes the indented code over and over again as long as `inbox < 10` is True. Then we increment the `inbox` by adding 1 over and over again.

When you run `whileLoop.py` (python `whileLoop.py`), the result will be:

```
You have a message.
You have a message.
You have a message.
You have a message.
You have a message.
```

```
You have a message.  
You have a message.  
You have a message.  
You have a message.  
You have a message.
```

Notice that the code ran for 10 times. That's because the count started at zero (`inbox = 0`) which is the first run of the while loop. For each run, 1 is added to the value of the inbox (second run the inbox is now equal to 1). This happened again and again until `inbox = 9`. Once the value of inbox reached 10, the code stops because `inbox < 10` is now False.

We can illustrate this more easily by typing the following code into the `whileLoop.py` file (overwrite existing code):

```
inbox = 0  
while inbox < 10:  
    print(str(inbox) + " You have a message.")  
    inbox = inbox + 1
```

When you run `whileLoop.py` this will appear:

```
0 You have a message.  
1 You have a message.  
2 You have a message.  
3 You have a message.  
4 You have a message.  
5 You have a message.  
6 You have a message.  
7 You have a message.
```

```
8 You have a message.
```

```
9 You have a message.
```

The count started at zero and ended at 9. This may sound a bit confusing at first but please endure. You don't have to remember everything. You can just first test your code anytime and see what happens.

In the previous example we made a while loop that stops when a certain value was met or exceeded (the block of code terminates when while statement evaluates to False). We can also get a bit more creative with it by making a seemingly infinite loop.

```
name = ''
while name != 'Casanova':
    print('Please type your name.')
    name = input()
print('Congratulations!')
```

When you run this you'll see something like:

```
Please type your name.
Yeah
Please type your name.
joh
Please type your name.
Nope
Please type your name.
Rickon
Please type your name.
Sansa
```

This won't stop until you enter the name that matches the code. If you run the code again and typed 'Casanova', you'll exit the while loop and run the code outside that.

```
Please type your name.
```

```
Casanova
```

```
Congratulations!
```

While loops are good if you want an action to be repeated over and over again until the statement evaluates to False. It's also good for applications wherein you don't know how many times a particular block of code should run.

## For Loops

In contrast, a **for loop** has more obvious limitations. You often specify (or you already know) how many times a particular block of code will run. Let's look at an example:

```
for i in range(10):  
    print(i ** 2)
```

The result when you run the code:

```
0  
1  
4  
9  
16  
25  
36  
49
```

64

81

What happened there? The code inside the **for loop** was executed from numbers 0 to 9 (the range). Each number was squared (exponent of 2) when printing the result.

Here's another example:

```
total = 0
for num in range(101):
    total = total + num
print(total)
```

When you run this, the print out result will be 5050. Here's what happened. First, we initiate a variable equal to zero (total = 0). Then we created a **for loop** which will run the numbers from 0 to 100 (the range feature just excludes 101). Inside the loop we add each “member of the range” into the total. This happens repeatedly from 0 to 100. The total then has changing values because of the consecutive additions.

Once every number in the range was used, the variable total now has a final value. We then print out the result which yields to 5050 (the sum of numbers from 0 to 100).

Instead of manually adding the numbers from 0 to 100 (e.g. 0 + 1 + 2 + 3 + 4 + 5 and so on), we save some time and effort because we're using a for loop instead. It's just a few lines of code and it just looks savvy.

How useful is this? Remember the example we gave earlier:

```
even_list = []
for num in range(1, 101):
```



```
if num%2 == 0:
    print("Even")
    even_list.append(num)
else:
    print("Odd")

print(even_list)
print("Counting even numbers: ")
print(len(even_list))
```

Here we can check if each number is Even or Odd. Then we can “isolate” the even numbers and add them into a list. We can also count how many are the contents of the list (len means length of the list).

If we run this you’ll see something like:

```
Odd
Even
Odd
Even
Odd
Even
Odd
Even
Odd
Even
Odd
Even
Odd
Even
Odd
```

```

Even
Odd
Even
Odd
Even
Odd
Even
Odd
... (it's a long result)
[2, 4, 6, 8, 10, 12, 14, 16, 18, 20, 22, 24, 26, 28,
30, 32, 34, 36, 38, 40, 42, 44, 46, 48, 50, 52, 54,
56, 58, 60, 62, 64, 66, 68, 70,
    72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 94, 96,
98, 100]
Counting even numbers:
50

```

First, we check if each number is Even or Odd. Then, we include the Even numbers into a list (`even_list`). Next is we count how many are those numbers from the range of 1 to 100.

Instead of manually (and using your fast analysis skills of determining if a number is Odd or Even), we just use a **for loop** to run through the numbers and verify each one. We then include those entries into a list and check how many of them are by using `len` (instead of manually counting the numbers).

This is very useful in many applications (especially numerical, computational, and data analysis). For example, we have a list of data and we want to verify if each one exceeds a certain value (e.g. Pass or Fail). We can run through each entry by using a for loop and then checking if it meets a certain condition. Next is we can then separate the passing values (Pass) from the

failing (Fail). We could also count the number of entries for each list.

In code it will look something like this (we check how many 4 and 5 star reviews for a book):

```
all_reviews = [5, 5, 4, 4, 5, 3, 2, 5, 3, 2, 5, 4, 3,
1, 1, 2, 3, 5, 5]
positive_reviews = []
for i in all_reviews:
    if i > 3:
        print('Pass')
        positive_reviews.append(i)
    else:
        print('Fail')

print(positive_reviews)
print(len(positive_reviews))
ratio_positive = len(positive_reviews) /
len(all_reviews)
print('Percentage of positive reviews: ')
print(ratio_positive * 100)
```

Save this in a new .py file (e.g. reviews.py). You can then run the program in the terminal by typing `python reviews.py`

You should see the results as this:

```
Pass
Pass
Pass
Pass
Pass
```

```
Fail
Fail
Pass
Fail
Fail
Pass
Pass
Fail
Fail
Fail
Fail
Fail
Pass
Pass
[5, 5, 4, 4, 5, 5, 5, 4, 5, 5]
10
Percentage of positive reviews:
52.63157894736842
```

First, we initialize an empty list (`positive_reviews`). We then check whether each entry is Pass (review rating should be higher than 3) or Fail (rating is 3 and below). We accomplish this by using a **for loop** where we can run through the entries (or iterate through the entries of the `all_reviews`).

Next is each review rating that's higher than 3 (4 and 5 are the passing ratings) we append them into the empty list (`positive_reviews`). The loop runs and does this until we get to the end of the `all_reviews` list.

Once the loop is finished, we print out the `positive_reviews` and see that only ratings higher than 3 appears:

```
[5, 5, 4, 4, 5, 5, 5, 4, 5, 5]
```

We then want to know how many the positive ratings are. We can do this manually or just apply a `len` to the list to know how many of them are (this is very useful if we have a massive list):

```
print(len(positive_reviews))
```

The result is 10. But this number is not useful in itself. We also want to know the context like the percentage of positive reviews. This way we can immediately get an idea if the reviews are really good (e.g. chance to become an Amazon bestseller?). We can get the percentage using the following lines of code:

```
ratio_positive = len(positive_reviews) /  
len(all_reviews)  
  
print('Percentage of positive reviews: ')  
print(ratio_positive * 100)
```

The output should be this:

Percentage of positive reviews:

```
52.63157894736842
```

What happened here is that we got the ratio between `len` (lengths) of `positive_reviews` and `all_reviews`. Then we multiplied it by 100 to get the percentage value. The result is ~52% which tells us that almost half of the reviews are good or positive.

## Summary & Review

In this chapter we've discussed Comparison Operators (`==`, `!=`, `<`, `>`, `>=`, `<=`), Booleans (True or False), Flow Control (if,

elif, else), while loops, and for loops. With this set of knowledge we can already create a simple program that may guide the user to a series of actions and outputs.

For example, if the user input is equal to the desired input (saved in the database such as login email and password), the user could successfully log in and access a certain page. Another useful application is in fast and large-scale data analysis. With Python, we can quickly check if a certain number or dataset falls within a range. We can also save the “passing values” into a list and determine its count.

Although the examples mentioned above are simple, we already get an idea about the potential of using Python in data analysis and other applications. It’s all about interacting with data or user input. Then, we set the conditions (comparison operators) and flow (if, else, elif).

## More Examples & Exercises

Evaluate if True or False:

```
5 == 5 and 6 != 5
2 == 2 or 3 > 5
5 >= 5 or 6 > 1
```

Predict the output or result of this block of code:

```
the_list = []
for i in range(0, 11):
    the_list.append(i)
print(the_list)
```

Try this another exercise. Predict the result (or explain the possible results) of this block of code:

```
div_three = []  
for i in range(1, 50):  
    if i%3 == 0:  
        div_three.append(i)  
print(div_three)
```

## Python Level Four

This is where it gets really interesting. We're about to create programs that resemble the real world ones. This becomes possible with the use of Functions. It's like creating mini-programs within a huge program. That's because each mini-program specifically does something and outputs an intermediary or final result. We'll discuss later how to accomplish this.

Aside from Functions, we'll also discuss Scope (Global & Local variables) and Errors and Exceptions. It's important to learn these so we can create a truly working program without much confusion. Take note that most valuable Python programs will have hundreds or thousands of lines of code in it. It's crucial that almost everything's clear both to the original maker of the code and the succeeding team.

Don't worry if the later topics won't make sense or if they'll confuse more. After all, they should be. Good news is with repetition you can get the hang of it. Even experienced programmers get stuck or they need to review the basics of Python from time to time.

Well, enough of the motivational talk. After all, it's just more motivating to get into action and quickly see the results. Let's start with creating mini-programs within a program.

### **Creating Mini-Programs within Your Program (Functions)**

Here's a basic example. Type this and save into functions.py

```
def hello():  
    print('Hello world!')
```



```
hello()
```

Run the program by typing `python functions.py`. You should see something like this:

```
Hello world!
```

We've just created a Function. First, we **define** the function and its name (1st line). Next is we add code to the body of Function to tell what it should do (2nd line). After we defined the function the code to be executed, we call the function. Once we do that, it will execute the code inside the body of the Function.

Almost all Functions work this way. We define a function, tell it what it should do, and then call it later on. Remember the correct syntax. Common errors here are often typographical (lacking a colon or the parenthesis).

It's a simple example. Let's take it one step higher.

```
def hi_name(name):  
    print('Hi ' + name)
```

```
hi_name('Aardvark')
```

In the hello function, there's nothing in the parenthesis. But it's a slightly different case here. We put something in the parenthesis which is called a parameter. It's like we're creating a placeholder for a variable (in this case it's *name*). Then we set a value to that variable when we call the Function later.

Yes, this sounds confusing at first. What we should remember here is that we create a function, define what it does, and call it later. Let's look at more examples to make this clearer:

```
def add_numbers(a,b):  
    print(a + b)  
  
add_numbers(5,10)  
add_numbers(35,55)
```

Can you guess what the above function does? Yes, you're correct. Simply, when we call the function we use the numbers inside and add them. When you run this the results should be:

```
15  
90
```

Here's another example:

```
def square_number(num):  
    print(num**2)  
  
square_number(50)  
square_number(9)
```

You're correct again. Just one look and you know immediately what it should do. The function will square the number passed inside when the function is called.

Those are simple examples. What happens if we incorporate Comparison Operators (`==`, `>`, `<`, etc.) and Flow Control (`if`, `elif`, `else`)?

```
def even_check(num):  
    if num % 2 == 0:  
        print('Number is even.')  
    else:  
        print('Hmm, it is odd.')
```

```
even_check(50)
```

```
even_check(51)
```

Let's explain the code for clarity. We defined a function that checks whether a number is odd or even. If the number is even, it prints 'Number is even.' If it's not, it prints 'Hmm, it is odd.' Finally we call the function and pass in the values 50 and 51 to see if each one is even or odd.

The result should be like this when we run the code:

```
Number is even.
```

```
Hmm, it is odd.
```

Those are basic examples of creating and using Functions. If you get more serious about Python programming, you'll encounter Functions with longer and more complex logic. There might be more if statements and possibly loops.

The point here is to demonstrate the power of Functions and the ability to create mini-programs. It's like creating a mini-program we can use again and again (without retyping the whole code). This way, we save time and make our program more readable and professional.

## **Scope (Global & Local Variables)**

Now that you've seen the power of Functions, let's now discuss Global and Local Variables. Yes, variables can be subdivided into two categories. The reason for this is that variables may have a different Scope, which makes them usable or "unusable" in certain parts of the code.

Here's a quick example so you can get an idea:

```
x = 25
```

```
def printer():  
    x = 50  
    return x  
print(x)
```

Save this into `scope.py` and try to guess the output before running the code. Notice `x` is mentioned twice (one before the function, `x = 25` and another inside the function `x = 50`). What would Python show?

If you run the code, it will return 25. But what if you call the function?

```
print(printer())
```

You guessed it right. The output will be 50. That's because we called the Function. Remember that whenever we call a function, it will execute the code inside it. In this case, inside the function we have set `x` equals to 50.

This might seem weird at first. How will we know (and how the computer will know) which 'x' are we referring to? This is where the concept of **Scope** comes in. In general, Python has rules so we'll know which variables we are calling or referencing.

The most important idea here is that Global variables are top-level (they're not inside of any function). When you call or use the variable later on, Python will look at the top-level ones. Another important idea here is Local variables are declared inside the functions. They're not in any way related to variables with the same name outside of the function. In other words, these variables are always being used locally (hence the name).

It will take some time before it all sinks in. Just remember that Global variables could be used or modified widely. On the other hand, Local variables are only ‘local’, which means they won’t be changed if we use a variable of the same name in other functions.

There are other ‘rules’ to allow Python which variables we are referring to. For instance, there are *Enclosing function locals* and *Built-in names*. But for now, it’s enough to know about Local and Global scope (and immediately tell the difference). This way, you’ll know for sure why our code behaves ‘unexpectedly.’ One of the common reasons is about the Scope and how Python determines which variable you’re referring to.

## **Handling Errors & Exceptions**

It’s amazing whenever our code works smoothly. No weird outputs and it’s just fast to move forward. But yes, when there are weird outputs it can become really frustrating. Even experienced programmers and data scientists get stuck for days finding out what really went wrong.

Welcome to the world of errors. It’s the most frustrating part for anyone who learns Python. This is where many beginners give up because it’s also the most time-consuming aspect of programming.

In movies about ‘hacking’ you might have seen hackers typing really fast and non-stop and then they get quick access to private logs and files. However, in real life it’s always much slower. Even experts make mistakes and there are a dozen of things that could go wrong every time we hit the keyboard.

That's because errors exist and will appear. Perhaps it's just a typo error or we failed to write the code properly. We're expecting something but another thing happened (e.g. Local and Global variables). Here's an example:

```
print('Hello'
```

Save this in `errors.py` and run it. You should see something like this:

```
File "errors.py", line 1
```

```
    print('Hello
```

```
        ^
```

```
SyntaxError: EOL while scanning string literal
```

Syntax errors are perhaps the most common error you'll encounter. Perhaps you forgot to add an ending quotation mark or a closing parenthesis. This kind of error is fairly easy to correct if you follow the correct syntax.

Thankfully, the error prompt will give you clues on how to deal with the error. Notice there's a little arrow head there that points the possible location and cause of the error. After that Python will show you what type of error it is. It will be easier then to correct the code because of those helpful indications.

Also notice that the code will just stop running and 'focus' on the error. This means the user won't be able to progress or do anything further once the error shows up. What we want is that our program will detect those errors and continue to run. This is where handling **Exceptions** come in. What this does is it will make the program to continue to run even with an error. It will find an *exception* and then proceed with the code. It works by first 'trying' the code and then moving to the 'except' if there's an error (similar to how if-elif-else statements work).

Let's just look at example to better illustrate this:

```
def spam(divideBy):  
    try:  
        return 42 / divideBy  
    except ZeroDivisionError:  
        print('Error: Invalid argument.')  
print(spam(2))  
print(spam(12))  
print(spam(0))  
print(spam(1))
```

Notice that when we call the function, we're passing a number as the denominator (we're dividing 42 with the number inside spam). The flow of the code starts with calling the function and then 'trying' to execute it. For example, when we call `spam(2)` what happens is 42 divided by 2 (answer is 21.0 if you run it). Then `spam(12)` results to 3.5 (answer to 42/12).

But it's totally different when we divide a number by zero. If we do this, we'll encounter what we call a `ZeroDivisionError`. This is where **except** comes in. It will handle the error by showing or doing something else:

```
except ZeroDivisionError:  
    print('Error: Invalid argument.')
```

After 'trying' the code and detecting an error, the flow will proceed with the *except clause* to handle that kind of error. Instead of stopping the program (or making it crash), it can still go on. Run this whole code and see what happens:

```
def spam(divideBy):
    try:
        return 42 / divideBy
    except ZeroDivisionError:
        print('Error: Invalid argument.')

print(spam(2))
print(spam(12))
print(spam(0))
print(spam(1))
```

Save this into `exception_handling.py` and run it (type `python exception_handling.py`). You should see something like this:

```
21.0
3.5
Error: Invalid argument.
None
42.0
```

The program handled the error and went on. It continued to call the function instead of stopping or crashing. Instead of showing the prompt `ZeroDivisionError` and crashing (doesn't go on with next lines of code), the **except** clause handles the error and continues on with the program.

This is very valuable in the user-side because users can have a smoother experience with the program. Errors happen all the time but we don't want them to fully exit our program. Instead, we want them to continue on while still being notified of the errors.



A similar example related to this is when we're setting a password. Perhaps the website or app requires a number for the password to be accepted. If we fail to include a number, the website should just notify the user that it requires a number for the password to be valid (and not crash the whole site). This way, the user can just 'try' again until the password becomes valid.

## **Summary & Review**

In this chapter we've discussed how to create mini-programs we can use again and again (functions). We've also discussed Scope (Local vs Global variables) and handling Errors & Exceptions. This is to familiarize ourselves with the most common errors and how to deal with them effectively (without consuming much of our time).

Take note that it's normal for programs to 'misbehave' and not print the desired output. Most common cause of error is we forgot to add a closing parenthesis or a colon. Or we've just written our code wrong (syntax error) which makes the program crash.

It's recommended to handle errors as they come and manage them in pieces. It's especially the case when creating and running functions. That's because they're mini-programs themselves. Errors will be common and frustrating. The way to handle them is to carefully read the error message and follow the instructions. In addition, you can also search for the cause and solution online (Google is your friend and you'll find tons of answers in StackOverflow).

## **More Examples & Exercises**

To reinforce and solidify your learning, predict what will happen if we run this function:

```
def sum_three(a,b,c):  
    print(a + b + c)  
sum_three(3,4,5)
```

Here's another exercise:

```
def div_three(num):  
    if num % 3 == 0:  
        print('This is divisible by three.')  
    else:  
        print('Hmm, nope.')  
div_three(3)  
div_three(5)  
div_three(9)  
div_three(1013854)
```

Once you've solved that, now it's your turn to create your own function. This time, instead of checking if the number is divisible by 3, we check if it's divisible by 5 (Hint: focus on the if statement).

Let's also have a review about *try* and *except*. Look at the following code and guess what it does (better though is to type the code and run it yourself later).

```
while True:  
    try:  
        x = int(input("Please enter a number: "))  
        break  
    except ValueError:
```

```
print("Oops! That was no valid number. Try  
again...")
```

Try entering an integer and then run again the program. But this time enter a string such as 'error'. Notice how the code handles the error.

## Python Level Five

We're in the last part of learning the basics of Python. Here we'll talk about the list and how to use it. Learning how to create, modify, and use lists is important in handling data and sequences.

Actually, we've already touched on lists (e.g. remember the for loops wherein we initiate an empty list?). Here we'll review those concepts we discussed earlier and talk more about how to manipulate lists.

### Creating a List

Basically, a list is just a sequence of values. Let's dive into an example:

```
my_list = ['eggs', 'ham', 'bacon']
```

It's similar to setting a value to a variable. But this time, we're literally creating a list (here, we're listing what many people have for their breakfast).

Let's create another list:

```
colours = ['red', 'green', 'blue']
```

There's a consistent syntax there where we set the name of the list and then add the values inside the square brackets. We can also create a list with numbers in it:

```
cousin_ages = [33, 35, 42]
```

We can even create a list with both strings and numbers in it:

```
mixed_list = [3.14, 'circle', 'eggs', 500]
```

And as mentioned in the early chapters, we can create an empty list and then add values into it later.

## Indexing the List Values (starts at 0, not 1)

Now we've created some lists, it's time to 'access' the values in it. This is very useful if we're only working on a certain value or we want to change it (more on this later). Let's look at an example and open an interactive shell (type python in the end so >>> appears):

```
>>>colours = ['red', 'blue', 'green']
```

Press Enter to save the list. To access and use the first value, we can do this:

```
>>>colours[0]
```

This should show 'red' because it's the first value (indexing starts at 0, not 1). To access the second value, we can just do this:

```
>>>colours[1]
```

The result should be 'blue'

This is a very convenient way of accessing the list. We use the list name and then affix the 'index number' of the value we're trying to access.

## Slicing the List

Earlier we've learned how to access a single value from the list. Here we'll access multiple values (which is often called 'slicing' the list). We'll just be accessing a portion of the list. This is very useful if we're only concerned about a list's certain portion/s. Here's an example:

```
my_list = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
print(my_list[0:2])
print(my_list[1:])
```

```
print(my_list[3:6])
```

Save this into list.py and run the program. The result should be this:

```
[0, 1]
[1, 2, 3, 4, 5, 6, 7, 8, 9]
[3, 4, 5]
```

What happened there? Notice that we used a number/s and a colon to determine which portion of the list to access. Also take note that indexing starts at zero (0).

For example, `my_list[0:2]` is like saying “Access the numbers from the start of the list (index zero) up to index 1 (index 2 is not included)”. The result is that it will only print the first two values. It will be a ‘sliced’ list `[0, 1]` and it doesn’t include the value of the second index.

What about `my_list[1:]`? Here, the ‘slicing’ starts at index one and then go on until the end of the list. The slice will be this:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9]
```

We can also slice the original list in the middle (or just somewhere not in the beginning or end). For example, `my_list[3:6]` will return:

```
[3, 4, 5]
```

The slicing starts at index three and ends at index five (but its value is not included in the slice). These may all seem counterintuitive at first but it’s just how Python works.

Just remember that indexing starts at zero. Also, the value of the second index is not included when doing the slicing.

Whenever you're in doubt, you can always test your code to see if the output is what you really desired.

## Getting the Number of List Values

Aside from accessing and slicing the list, it's also important to get its 'length' to see how many values are there in the list. In the previous chapters we've already mentioned this to 'count' the contents of a certain or resulting list.

For example, let's identify the number of values in this list:

```
my_list = [0,1,2,3,4,5,6,7,8,9]
```

We can accomplish that by using the len method onto the list:

```
len(my_list) # or print(len(my_list)) so we can see  
the output
```

If we run this the result will be:

```
10
```

Here counting starts at 1 (in contrast, indexing starts at zero). After all, we're literally counting how many values there are inside the list.

## Changing the Values

Lists are mutable, which means we can change the values anytime we want. For example:

```
colours = ['red', 'green', 'blue']  
colours[0] = 'yellow'  
print(colours)
```

Type, save, and run this and you should see:

```
['yellow', 'green', 'blue']
```

Notice that `colours[0]` is originally equal to 'red'. However, we can change that by first accessing the value we want to change, getting the index, and then setting it equal to a new value.

We can also do this to further modify the contents of the list:

```
colours[1] = 'purple'
colours[2] = 'magenta'
print(colours)
```

The result is:

```
['yellow', 'green', 'blue']
```

The entire list has changed because we've set a different value to each content. But what happens if we use an index that goes beyond the original list (e.g. perhaps adding another value to the list)?

```
colours[3] = 'pink'
print(colours)
```

You'll see an error message because our list is only up to index two (there's no index three):

```
Traceback (most recent call last):
  File "list.py", line 8, in <module>
    colours[3] = 'pink'
IndexError: list assignment index out of range
```

Note that the error message shows what we've done wrong here. This gives us a clear clue of what went wrong and what to do to correct it.

## List Operations (e.g. Concatenation, Append)



Earlier we're trying to add a new value to a list but the index for it doesn't change. In other words, we're trying to 'extend' the list by adding new values. The right way to accomplish this is by:

```
colours = ['red', 'green', 'blue']
colours.append('pink')
print(colours)
The result will be:
['red', 'green', 'blue', 'pink']
```

We 'appended' a new value into the list. That new value went into the end of the list. This is very useful when we're building a list as with the case of a **for loop** we discussed earlier.

```
even_list = []
for num in range(1, 101):
    if num%2 == 0:
        print("Even")
        even_list.append(num)
    else:
        print("Odd")

print(even_list)
print("Counting even numbers: ")
print(len(even_list))
```

We started with an empty list and as we go through each number, the even numbers get appended into the list. After we've exhausted the numbers we should run, the list is now complete.

Aside from **append**, another useful and common **list operation** is concatenation. Simply, this is adding or combining the values of the lists (or just their slices) to form a new list. For example:

```
fave_series = ['GOT', 'TWD', 'WW']
fave_movies = ['HP', 'LOTR', 'SW']
fave_all = fave_series + fave_movies
print(fave_all)
```

Save this in a fave.py and run it. You should see this:

```
['GOT', 'TWD', 'WW', 'HP', 'LOTR', 'SW']
```

What happened here is that we concatenated the two lists just by adding them (thereby creating a new list with all the values of both).

Aside from concatenating full lists, we can also concatenate portions or slices of lists:

```
level_one = ['shell', 'math', 'data-types']
level_two = ['print', 'variables', 'input']
two_levels = level_one[0:2] + level_two[1:]
print(two_levels)
```

Save and run this and you should see:

```
['shell', 'math', 'variables', 'input']
```

Notice that we're only using the first two values of the first list and the last two values of the second list. In other words, we first access the desired portions of the lists and then concatenating them to form a new one.

## Summary & Review

In this chapter we've focused on creating lists and manipulating them. We saw how to access a certain value or slice of the list. We also learned how to do common list operations such as concatenation and append.

There are also other list operations that you may encounter in the near future. Here are some examples:

```
list.insert(0, 'indigo')  
list.extend('orange', 'violet')  
list.remove('pink')
```

We can already get an idea of what they do just by reading the code. However, it's still recommended to focus on what we've discussed this chapter as these are the most common ones you'll encounter especially when it comes to data science and machine learning (indexing, slicing, append, concatenation).

The goal here was to introduce you to the power of lists and what can you do with them. As you get more serious with learning Python and data science as a whole, you'll be able to apply what we've discussed (or even get creative on how to use lists). As you gain more skills and experience, you'll be able to piece them all together and use them conveniently.

## More Examples & Exercises

Here's a list of groceries:

```
my_list = ['eggs', 'ham', 'bread', 'milk', 'pasta',  
           'seafood', 'cereal']
```

Your tasks are:

1. Access the first item in the list.

2. Change the second item into 'spam'.
3. Print the new list.
4. Slice my\_list. Show the first four items.
5. Create a new list called 'your\_list' and add three different items.
6. Concatenate the my\_list and your\_list.

These are easy tasks and you can quickly test if you're doing them right. A valuable tip here is to write the expected output first and then write the code. Test and compare the expected output with what you see in your Terminal or Shell. Make changes until it's a full match.

## Recap and Some Advice

Practice and repetition is the key to learning and understanding Python (and any other programming language). Some of the concepts might not make sense at first (or they'll seem confusing) but if you endure, something will click and everything will look clear to you.

Aside from repetition and practice, it's also recommended to do some reading of other people's code (search in Github and type 'python'). Try to make sense of the lines of code. As you gain more experience, you'll immediately get a clue of what a block of code does.

Along the way you'll surely get stuck because of error messages and perhaps a lack of understanding. It's normal because programming is truly a complex task. That's actually a good thing because other people have likely encountered the same error messages and struggles you'll be dealing with. You'll quickly find the answers by using Google and StackOverflow.

Finally, experiment with new approaches about solving a problem. Perhaps you have a shorter or more efficient way to accomplish certain programming tasks. You can always test a block of code and see what it does. The key here is to start small and test the code right away. This way, you'll know if you're heading into the right direction.

## Resources

Here are the links and resources we've used to complete this first part of the book:

<https://www.python.org/>

<https://docs.python.org/3/>

<https://www.anaconda.com/download/>

<https://atom.io/>

<https://automatetheboringstuff.com/>

<https://developers.google.com/edu/python/>

<https://stackoverflow.com/>

<https://www.python.org/about/quotes/>

## The Zen of Python

Beautiful is better than ugly.

Explicit is better than implicit.

Simple is better than complex.

Complex is better than complicated.

Flat is better than nested.

Sparse is better than dense.

Readability counts.

Special cases aren't special enough to break the rules.

Although practicality beats purity.

Errors should never pass silently.

Unless explicitly silenced.

In the face of ambiguity, refuse the temptation to guess.

There should be one-- and preferably only one --obvious way to do it.

Although that way may not be obvious at first unless you're Dutch.

Now is better than never.

Although never is often better than right now.

If the implementation is hard to explain, it's a bad idea.

If the implementation is easy to explain, it may be a good idea.

Namespaces are one honking great idea -- let's do more of those!

—**Tim Peters**

## Thank you !

Thank you for buying this book! It is intended to help you understanding Python for Data Analysis. If you enjoyed this book and felt that it added value to your life, we ask that you please take the time to review it.

hat you please take the time to review it.

**Your honest feedback would be greatly appreciated. It really does make a difference.**

If you noticed any problem, please let us know by sending us an email at [review@aisciences.net](mailto:review@aisciences.net) before writing any review online. It will be very helpful for us to improve the quality of our books.



We are a very small publishing company and our survival depends on your reviews.

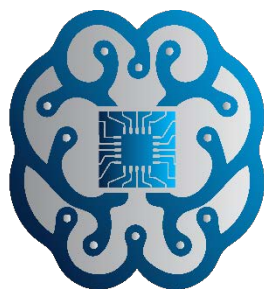
Please, take a minute to write us an honest review.

If you want to help us produce more material like this, then please leave an honest review on amazon. It really does make a difference.

<https://www.amazon.com/dp/B07FTPKJMM>







**AI SCIENCES**