THE PYTHON STARTER KIT

AN IN-DEPTH AND PRACTICAL COURSE FOR BEGINNERS TO PYTHON PROGRAMMING

CODE ADDICTS

The Python Starter Kit

An In-depth and Practical course for beginners to Python Programming. Including detailed step-by-step guides and practical demonstrations.

- Code Addicts -

Table of Contents

- Legal notice
- About this ebook
  - Why Spyder?
  - Top features of Anaconda and Spyder IDE
  - How to use this ebook in 10 Steps
  - Feedback, questions, or suggestions
- Chapter 1: Introduction to Programming with Python
  - Essential properties of a functional computer program
  - Why Python?
  - So, what is Python?
    - Python's design philosophy
  - Main online references and resources
    - Online Python interpreters
    - Smartphone Python interpreters
- Chapter 2: Setting up the Python programming environment
  - Starting Spyder with Anaconda
  - Hello, World!
  - Interactive vs script modes of programming
    - Python script mode programming
    - Python interactive mode programming with Spyder
  - The 10 basic style guides for Python code
    - 1. Python identifiers
    - 2. Case sensitivity
    - 3. Reserved words
    - 4. Indentation

- 5. Multi-Line Statements
- 6. Multiple statements on a single line
- 7. Quotation
- 8. Comments
- 9. Blank lines
- 10. Waiting for the user

Chapter 3: Python Variables and Basic Data Types

- 1. Variables
  - Assigning values to variables
  - Multiple assignment of values to variables
- 2. Basic Data Types: Numbers
  - Creating a number object
  - Number coercion from int to float and vice versa
- 3. Basic Data Types: Strings
  - Creating string objects
  - Manipulating strings

Chapter 4: Basic Operators in Python

- Types of Operators in Python
- Assignment operators
- Arithmetic operators
- Comparison Operators
- Logical operators
- Membership operators
- Other operators

Chapter 5: Python Lists

- What is a list?
- Creating a list
- Accessing list values

- Updating and deleting list values
- Basic list operations with built-in functions and methods

Chapter 6: Python Tuples
- What is a tuple?
- Creating a tuple
- Accessing tuple values
- Updating tuple values
- Basic tuple operations with built-in functions and methods

Chapter 7: Python Dictionary
- What is a dictionary?
- Creating a dictionary
- Accessing dictionary *key:value* pairs
- Updating a dictionary
- Basic dictionary operations with in-built functions and methods

Chapter 8: Decision Making in Python
- 1. The *if* structure
- 2. The if… else structure
- 3. The if… elif… else structure

Chapter 9: Loops in Python
- The *for* loop structure
  - Using else with the for loop
- The *while* loop structure
  - Using else with the while loop
- Loop control statements
  - The *break* statement
  - The *continue* statement

Chapter 10: Functions and Arguments
- Python function

        - Defining a function
        - Calling a function
    - Function arguments
        - Keyword arguments
        - Default arguments
        - Required arguments
        - Variable length arguments
- Chapter 11: Python Input and Output
    - Python input with input()
    - Python output with *print()*
        - Output formatting
- Chapter 12: Advanced Python Programming
    - Practical applications of the Python language in the modern world
        - GUI-based desktop applications
        - Scientific and computational applications and games
        - Web frameworks and applications
        - Script writing and automation
        - Machine learning and data science
        - Become an ethical hacker or penetration tester

Legal notice

About this ebook

Programming is typically taught as a unit in every computer-related university course, but more in-depth in Computer Science (CS), Information Science (IS), and Information and Communication Technology (IT and ICT) courses. Reputed institutions of higher learning around the world including MIT, Florida Tech University, Harvard University, University of Liverpool, and University of Birmingham where young future programmers attend use Python to teach computational teaching units.

That you have decided to advance your knowledge of computers and even learn how to write programs for your devices is a great feat that must be celebrated. You are now on a journey to acquire the programming and computational thinking knowledge that reputable universities around the world are teaching their learners – except that you study at your own pace and do not have to buy costly text books.

We have condensed this from hundreds of hours of video resources online, summarized notes from popular learning websites such as *Coursera* and *Edx*, and information on hundreds of websites, programming problem solving tools on the internet, and textbooks. We use a very practical and market-ready teaching approach that you can follow with ease and acquire the programming skills that you can apply on your job if you are looking to expand your marketability, or in the market if you are looking to build a career in programming.

This course takes about two to four weeks to complete, assuming you study two hours a day and have another one or more hours to practice and attempt the contained practice questions. There are many ways to study programming. We from Code Addicts understand the needs and the time constraints of our target audience, hence we focus on imparting the knowledge and skills using the most efficient and practical approach.

This ebook is an invaluable guide designed for absolute newbies to computer programming. It can also be used by students taking programming-related courses as a way to prepare for their semesters, get a different approach to learning, or to just have some fun.

It helps learners lay a solid foundation that could progress to popular career paths such as system development, web design, machine programming, game coding, and others. The skills you gain from this ebook is also invaluable if you are eying a DIY project such as building an autonomous car, a web scrapper, or an Android application.

Why Spyder?

There is a saying, “evolution does not re-invent; it uses what already is.” There are hundreds of amazing programming IDEs available for learners to download and use on the Internet today. These resources are built to make learning programming not only attractive to newbies; they also come with powerful tools available so programmers do not have to fumble around with a plain text editor or cram numerous syntaxes and commands.

For beginners like you, it means you learn programming by discovering what the code does so that you can understand its structure and composition better. Text editors typically used to introduce newbies to programming are just that – plain. Anaconda comes with countless features that have a huge influence in the efficiency and productivity of your studying including the Spyder, IDE—one of the best there is.

Professional programmers typically do not begin their projects from a plain text editor; they use IDEs specially tailored for the kind of project at hand. Most have a list of two or three favorite IDEs that they rely on to work on programming projects from start to finish. We will introduce you to Spyder Python IDE that cones with Anaconda.

Anaconda Python with the Spyder IDE, paired with the Sublime Tex 3 text editor, makes a very powerful platform to perform lightweight beginner and professional programming. You will get introduced to real-life programming with an elegant and powerful set of tools that provides all the resources you need to do your magic in one place. These tools take care of all the basic aspects of programming such as code completion and debugging so that you can concentrate on learning the code.

Top features of Anaconda and Spyder IDE

Anaconda and Spyder are our most preferred tools to teach newbies programming in Python because:

1. The interface of Anaconda with Spyder IDE is lightning fast and very intuitive.
2. It is easy to configure the Anaconda platform for your projects.
3. System offers tons of well-designed themes.
4. Anaconda is free to use and redistribute with friends and family.
5. This IDE supports both 32- and 64-bit Windows, MacOS, and UNIX-based operating systems including Linux Ubuntu, Redhat, CentOS, and others.
6. Anaconda's Anaconda Accelerate component and other tools including IOPro, MKL Optimizations, and NumbaPro offer high performance programming computing.

Just the way your choice of a word processor directly affects how fast and good of a writer you are, your choice of text editor or IDE greatly influences how effectively and quickly you master the Python language to become a bonafide programmer.

There are quite a number of other Python IDEs available today that you can try including Pydey with Eclipse, PTVS, Eric Python, VIM, Emacs, and PyCharm. We have invested a lot of time to research and try the many IDEs in the market and we are convinced you will reap the benefits of

When you complete learning Python Programming and Computational Thinking using this ebook, you can go ahead and check out these other IDEs to discover the strength points and best use cases of each. Note that these are just a few of the basic IDES invaluable for beginners, there are many other commercial IDEs developed with tools and resources for professional developers and enterprises.

How to use this ebook in 10 Steps

This book is made up of 12 chapters, the first introducing Python as a programming language. The recommended way to use this ebook to study Python programming is to:

1. Begin at Chapter 1 and follow the format of the ebook to the last chapter. However, if you are not a complete newbie to Python programming, feel free to skip to those sections or chapters that you wish to focus on.
2. Study the chapter introduction to discover what you will learn in that chapter. You can then quickly skim the chapter to the last *Chapter Summary* section to have a better feel of what you will learn in the chapter. You will also find some sample programming problems the chapter should help you solve.
3. Study and take notes to understand what is taught in the top sub-chapter then carry out the practice exercises as described in the guide using the Spyder IDE. We recommend that you enter the code EXACTLY as it appears to get the results expected during the first attempt.
4. Practice what you learn after every exercise, expanding your knowledge with own code and trying out different variations of the program code syntax and even variables and values. Be sure to save each exercise you do separately.
5. This ebook contains links and tips that you can use for reference. There are millions of free reference websites, online tools, documents, textbooks, wikis, and downloadable documents that you can refer to after each section.
6. Get used to the Anaconda and Spyder interfaces and explore the myriad of tools, resources and features built inside it to make your learning easier. Practice your exercises and examples ceaselessly until you understand why the code is written the way it is and why it behaves the way it does.

7. The internet is a programmers' playground. There are tons of amazing meet up places and forums e.g. *Github, Stack Overflow, Reddit*, or *The Python Challenge* that you can join other upcoming and seasoned programmers to learn from them and help others.
8. Practice and teaching someone else is the surest ways to hone your programming skills. IntroToPython.org, for instance, offers mini-lessons that you will find useful and PythonLearn.com hosts hundreds of videos that explain everything in a different way.
9. When you complete this course, you can begin to focus on a specific path of Python programming. The last chapter of this book introduces how you should expect to apply your Python skills in out there.
10. When you have adequately laid the foundations of Python programming and computational thinking, you can proceed to study the intermediate and advanced levels of studying Python. We will also offer you a chance to venture into ethical hacking with a new book, "Ethical Hacking with Python" besides a lot more complex Python paths that we will not touch in this newbie's ebook.

Feedback, questions, or suggestions

We have made every effort to ensure that the information contained in this ebook is accurate, relevant, and complete. However, because it is researched and written by humans, it may not be absolutely free of errors.

At Code Addicts, we always welcome all forms of feedback from our readers – compliments, complaints, corrections, and questions – regarding the content of our eBooks. We would like to know what you think about this book, what you like and what you do not, as well as what you found useful and what you did not.

Your feedback is important because it helps us create titles that will best help you and others learn the coding and hacking skills they seek fast and with less difficulty. Do not hesitate to contact us if you have feedback in form of comment, question, complain, or suggestion.

Chapter 1: Introduction to Programming with Python

Computer programming, often shortened to just programming or coding, is a process that begins with the original formulation of a computing problem and culminates in to the compilation of an executable computer program.

Writing a computer program is a simple process to a person who has studied the steps involved and has experience in them. These steps include problem analysis and understanding, thinking of possible solution scenarios, and formulating algorithms that will solve the generic problem that the program is designed to.

The process does not end here. Professional programming will also involve verifying of the requirements of the algorithm created including ensuring its precision and input and environmental parameters as well as the computing resources it needs to be implemented successfully. All this is done during the development lifecycle of the program algorithm (coding) using a target programming language.

It is no secret that software engineering and computer programming are today among the most popular activities for newly trained engineers and developers. As you strive to become a developer or a software engineer, whether to start your own company or make a career out of it, it is vital that you learn how to think like a computer scientist in order to learn how to create functional computer programs that solve problems the right way.

Essential properties of a functional computer program

With time, you will learn the many approaches used to develop a computer program and you will choose a favorite development lifecycle that works for you and is efficient in developing a program that solves the problem at hand. The important thing is that the program must satisfy several fundamental properties to be considered effective, hence valuable.

1) ***Usability***: The ergonomics and intuitiveness of a computer program is what makes it practical to use. The program you create should be easily usable by the intended user for the defined purpose, and even in some cases for unintended purposes.
2) ***Efficiency***: A good program is one that is lean and performs well using minimal system resources. It should use the least processing time, and primary and secondary storage spaces, and minimal network bandwidth to solve a problem and interact with the user and other systems.
3) ***Robustness***: This refers to how well the program you create anticipates problems when dealing with errors (not bugs). Ensuring the robustness of your programs is beyond the scope of this book.
4) ***Reliability***: How often are the results of a program correct? A good program must be reliable enough to always give the right output for the given input and parameters.
5) ***Readability***: When you write a program code, another human programmer must be able to easily read and comprehend it, to understand the purpose and control flow, and the intended operation of the source code.
6) ***Portability***: Considering that there is a wide range of different computer operating systems and hardware platforms, your source code should be able to compile or interpret and run on as many of them as possible.
7) ***Maintainability***: This is the ease by which a program may be modified and improved for its present and future developers. Maintenance involves removing bugs, updating software features,

patching security holes, customization, and adapting the code to new environments.

Why Python?

Python is a free programming language. However, what makes it the most preferred for learners and professional developers is that the programs developed on it are naturally excellent on all the seven of the above essential properties of a good computer program.

Python is also very easy to learn, as you will discover. Compared to other popular object-oriented programming language such C, C++, C#, Java, Python is has the simplest and most readable syntax. For instance, for our first program, we will learn that a simple code to tell the computer to print the statement ***"Hello World!"*** on the screen is simply:

```
print ("Hello, World!")
```

So, what is Python?

We can define Python as powerful object-oriented programming language that is easy to learn because it has very simple syntax that even a non-programmer who understands the English language can understand. It is one of the most popular programming languages in the world today, being a general-purpose language high level language with very efficient high-level data structures.

Python's elegant and simplistic syntax, along with its dynamic typing and interpreted nature, makes it the ideal language for beginners to programming and also a robust platform for use in rapid application development and scripting purposes on various platforms. Python's interpreter is freely available for download from python.org and its expansive standard libraries are available – on source compiled or in binary form – for all major computing platforms all over the internet.

Python.org has all the distributions of the latest and archived interpreter version as well as useful documentations and learning tools, and links to Python modules, programs and tools, and other additional documentation and resources by third parties.

Python's design philosophy

The design philosophy of Python emphasizes on code readability and its syntax is structured to allow programmers to code their concepts in fewer lines of code compared to how they are coded in other object oriented programming languages such as Java or C++.

Because of this property, the construct of the Python language has made it the most preferable platform on which to develop both small and large scale application programs. Besides, Python supports multiple programming paradigms besides object-oriented including imperative and procedural (functional) styles.

Python’s automatic memory management, dynamic type system, and large and comprehensive standard library that you can download and use off-the-shelf are a result of years of dedicated development by a massive community of enthusiasts, developers, and learners just like you scattered all over the world.

Python is just the language for you; you are on the right path to shaping the future of computing and humanity. Follow your curiosity, learn and practice, and strive to contribute to the Python community whenever and however you can.

Main online references and resources

In this book, we have invested a lot of time and effort to ensure that we explain concepts and code in the simplest and most concise manner. However, we do understand that it may take more than one take for a learner to grasp an idea the first time.

As you get started, should you feel the need to do further research before or after a lesson in the book, here are a few useful references and resources you will find highly invaluable:

- Read http://wiki.python.org/moin/BeginnersGuide/Programmers/ for official information for beginners to the Python language if you are already proficient with other programming languages. It is recommended you check out this resource before you begin the next chapter of the book.
- If you are a true beginner to programming and have no experience with any other programming language, you should read the official basics page on http://wiki.python.org/moin/BeginnersGuide/NonProgrammers/.
- Still standing on the ledge unsure whether to dive into the Python pool? The appetite page on Python.org is a great place to get motivated with an overview of the many reasons why Python programming is the best way forward for an aspiring programmer.
- The Intro to Python organization has compiled an in-depth and comprehensive series of excellent lectures suitable for all levels of learners. Go to http://introtopython.org/ and explore the many lectures developed by dedicated professionals. The site also features hundreds of exercises that you can use later to reinforce what you learn in this book.
- Do you find it easier to understand a concept watching a YouTube video? Python for Everybody offers great reference videos and presentation slides with lots of example that a learner such as yourself

will find very useful. Go to https://www.py4e.com/ to explore what is on offer.
- LearnPython.org is a website supported by DataCamp dedicated to offering online interactive Python tutorials for experienced programmers and beginners – generally anyone interested in learning Python programming. Go to http://www.learnpython.org/ and click on a chapter to begin learning.

Online Python interpreters

Are you unable to write and run Python code on your computer? Perhaps you are on the road and have access only to a phone? No problem, you can write and write Python code without a single installation.

Online interpreters brings the simplicity and adaptability of the Python interpreter to your web browser. You can use these online interpreters to quickly test a line of code, practice a syntax you learnt, or even use it as your primary coding playground. The most popular interpreters you should check out are:

- Official Python.org console.
- REPL.it.
- Trinket.io
- PythonTutor.com - This online interpreter comes with a very useful visual debugger. It even shows a step-by-step process of code execution and can point out where you went wrong in the code.
- Live.SymPy.org – The powerful SymPy project is math-oriented and features a symbolic mathematics Python library.
- CodeSkulptor.org and Skulpt.org are simple but helpful online interpreters

Smartphone Python interpreters

A few years ago, it would have been impractical to take your coding classes with you wherever you take your smartphone. There are quite a number of apps that have been developed for Android and iOS smartphones that you can install and keep writing and running Python code while on the move and offline. If you are an Android user, check out QPython or PyCo on the Play Store and if you are an iPhone user, go to http://www.pythonforios.com/ to see what apps are best for you.

You are now ready to begin the actual programming with Python! Let's get the ball rolling!

Chapter Summary

Q: In your own words, what is Python really? You can (and are encouraged) make comparisons to related technologies in your answer.

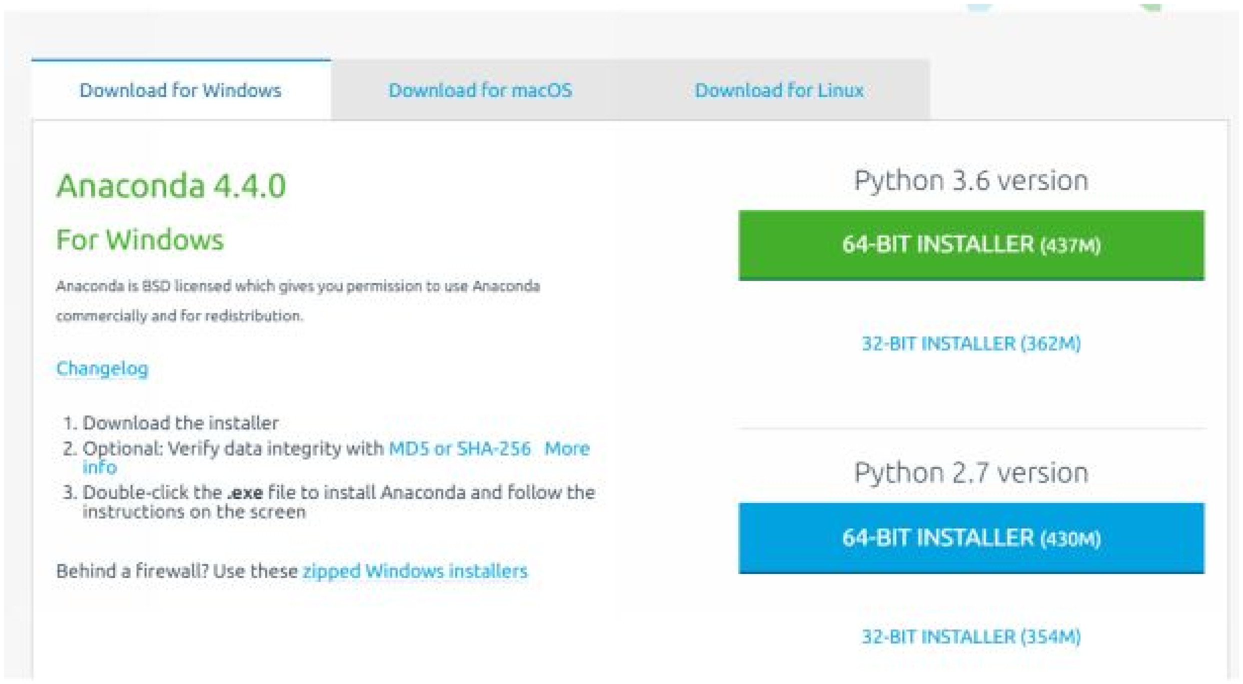
A: In your answer, be sure to cover the following points:

- Python is an interpreted language.
- Python is dynamically typed.
- Writing Python code is quick.
- Pyhton is a general purpose language.

Chapter 2: Setting up the Python programming environment

One of the top reasons why we chose to use Anaconda to introduce Python programming is because of the simplicity of setting up the programming platform. To begin, on your browser, go the Anaconda download page https://www.continuum.io/downloads/.

Step 1: On the download page, choose your type of operating system from Windows, OSX, and Linux tabs, then select whether to download a version for 64-bit or 32-bit.



Anaconda 4.4.0

For Windows

Anaconda is BSD licensed which gives you permission to use Anaconda commercially and for redistribution.

Changelog

1. Download the installer
2. Optional: Verify data integrity with MD5 or SHA-256 More info
3. Double-click the .exe file to install Anaconda and follow the instructions on the screen

Behind a firewall? Use these zipped Windows installers

Python 3.6 version

64-BIT INSTALLER (437M)

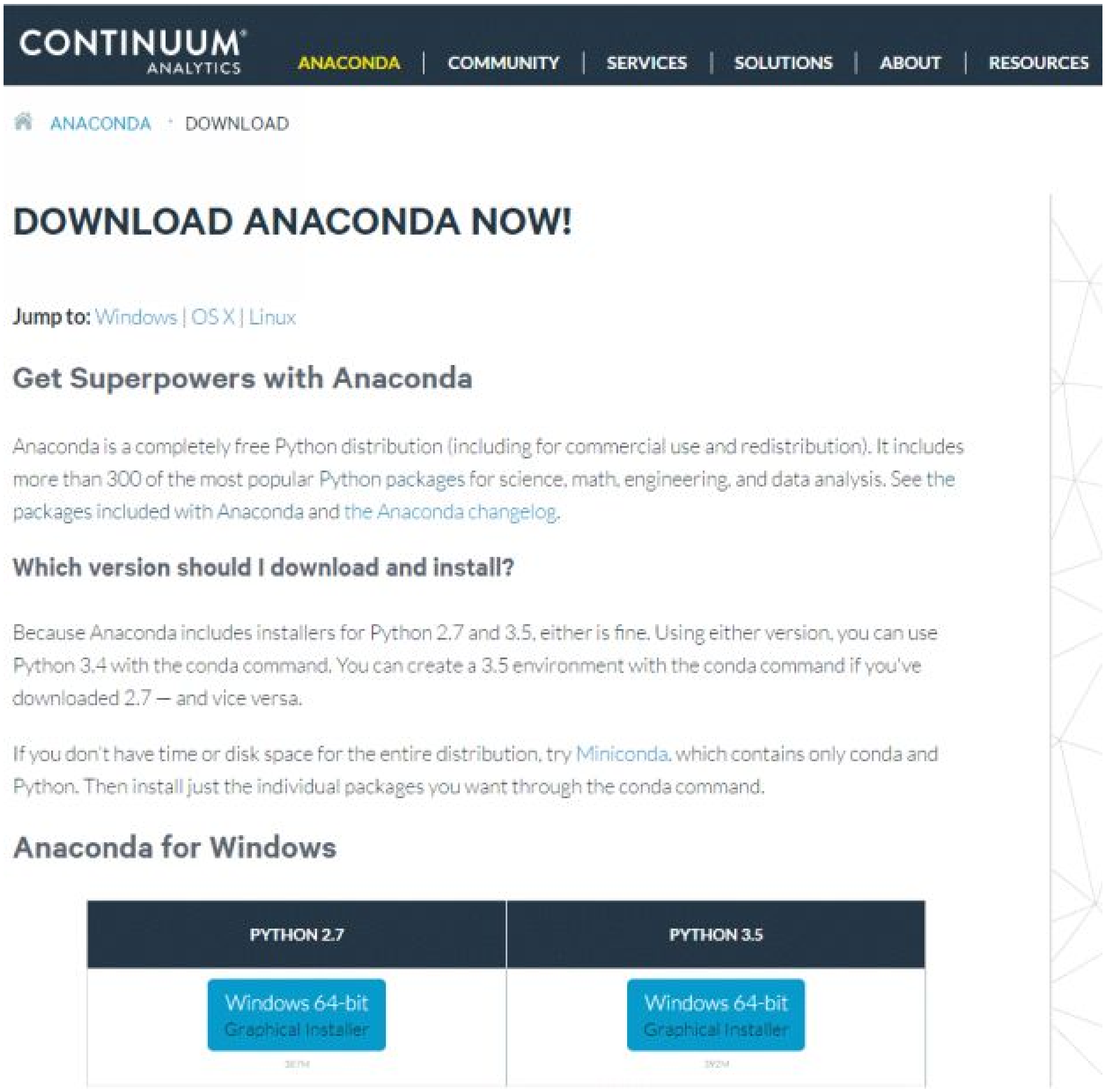
32-BIT INSTALLER (362M)

Python 2.7 version

64-BIT INSTALLER (430M)

32-BIT INSTALLER (354M)

Step 2: Next, choose the version of Python interpreter to download: the older Python 2.7 or the newer Python 3.6. Select the later (Python 3.6) then wait for the download to initialize.


CONTINUUM ANALYTICS
ANACONDA | COMMUNITY | SERVICES | SOLUTIONS | ABOUT | RESOURCES
ANACONDA · DOWNLOAD

DOWNLOAD ANACONDA NOW!

Jump to: Windows | OS X | Linux

Get Superpowers with Anaconda

Anaconda is a completely free Python distribution (including for commercial use and redistribution). It includes more than 300 of the most popular Python packages for science, math, engineering, and data analysis. See the packages included with Anaconda and the Anaconda changelog.

Which version should I download and install?

Because Anaconda includes installers for Python 2.7 and 3.5, either is fine. Using either version, you can use Python 3.4 with the conda command. You can create a 3.5 environment with the conda command if you've downloaded 2.7 — and vice versa.

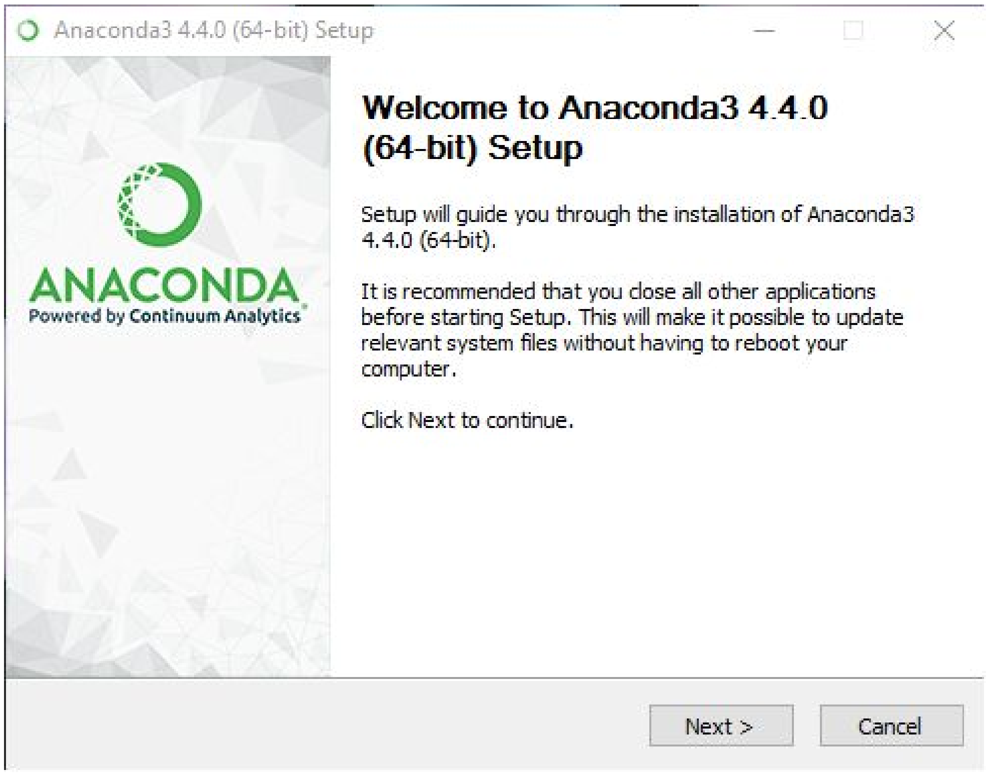
If you don't have time or disk space for the entire distribution, try Miniconda, which contains only conda and Python. Then install just the individual packages you want through the conda command.

Anaconda for Windows

PYTHON 2.7
Windows 64-bit Graphical Installer
PYTHON 3.5
Windows 64-bit Graphical Installer

Step 3: Wait for the download to complete. Note that because the Anaconda installation file is quite large, depending on your connection speed, it may take up to half an hour for the download to complete.

Step 4: Once the download process is complete, locate the downloaded installer (most likely in your downloads destination directory) then run the installer.


Anaconda3 4.4.0 (64-bit) Setup

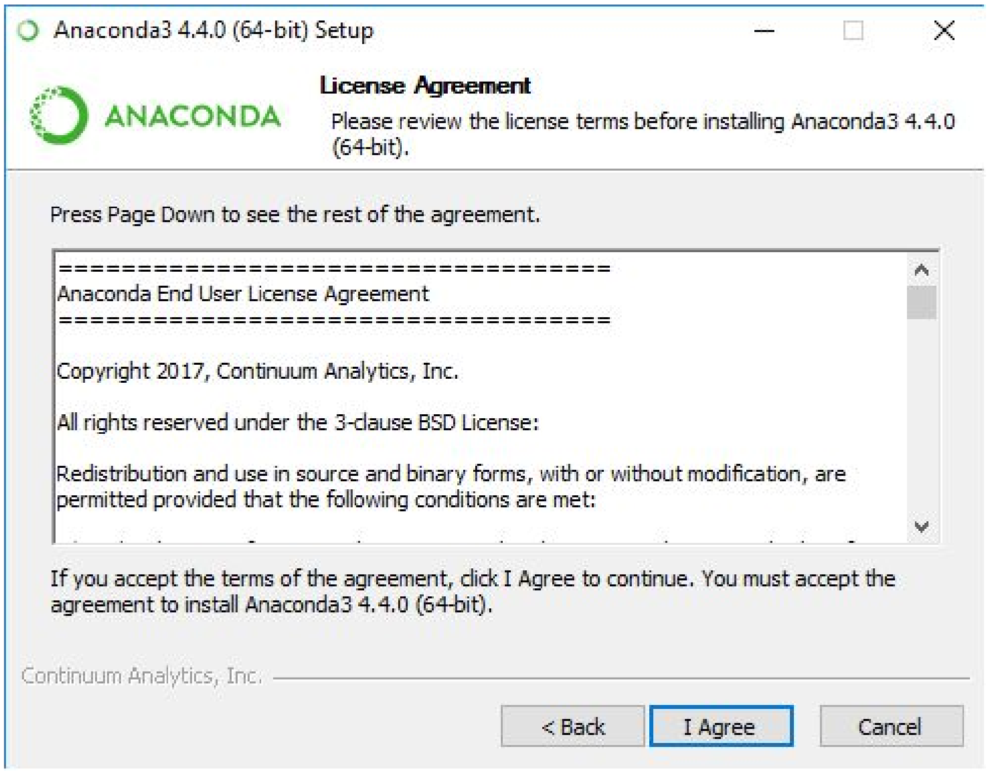
Welcome to Anaconda3 4.4.0 (64-bit) Setup

Setup will guide you through the installation of Anaconda3 4.4.0 (64-bit).

It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.

Click Next to continue.

Step 5: Click *Next* to begin the installation. On the next screen, take some time to read the end user license agreement then click *I Agree* to accept and proceed with the installation.


Anaconda3 4.4.0 (64-bit) Setup

License Agreement

Please review the license terms before installing Anaconda3 4.4.0 (64-bit).

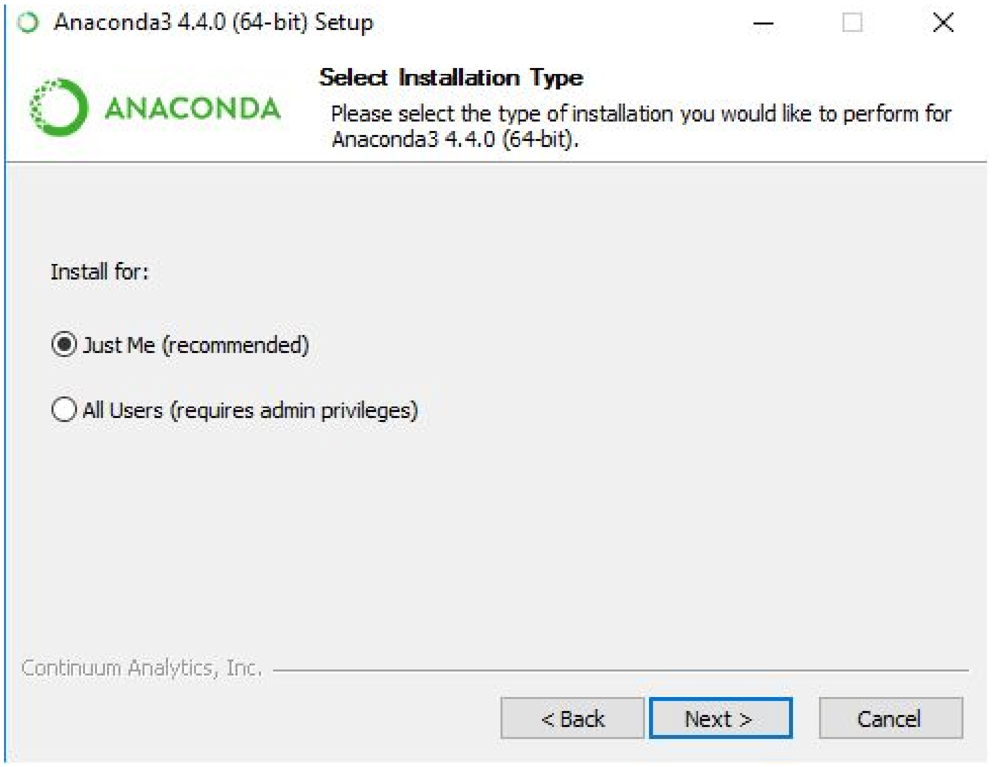
Press Page Down to see the rest of the agreement.

```
===================================
Anaconda End User License Agreement
===================================

Copyright 2017, Continuum Analytics, Inc.

All rights reserved under the 3-clause BSD License:

Redistribution and use in source and binary forms, with or without modification, are
permitted provided that the following conditions are met:
```

If you accept the terms of the agreement, click I Agree to continue. You must accept the agreement to install Anaconda3 4.4.0 (64-bit).

Continuum Analytics, Inc.

Step 6: Select the installation type in the next screen.


Anaconda3 4.4.0 (64-bit) Setup

Select Installation Type

Please select the type of installation you would like to perform for Anaconda3 4.4.0 (64-bit).

Install for:

[✓ Just Me (recommended)]

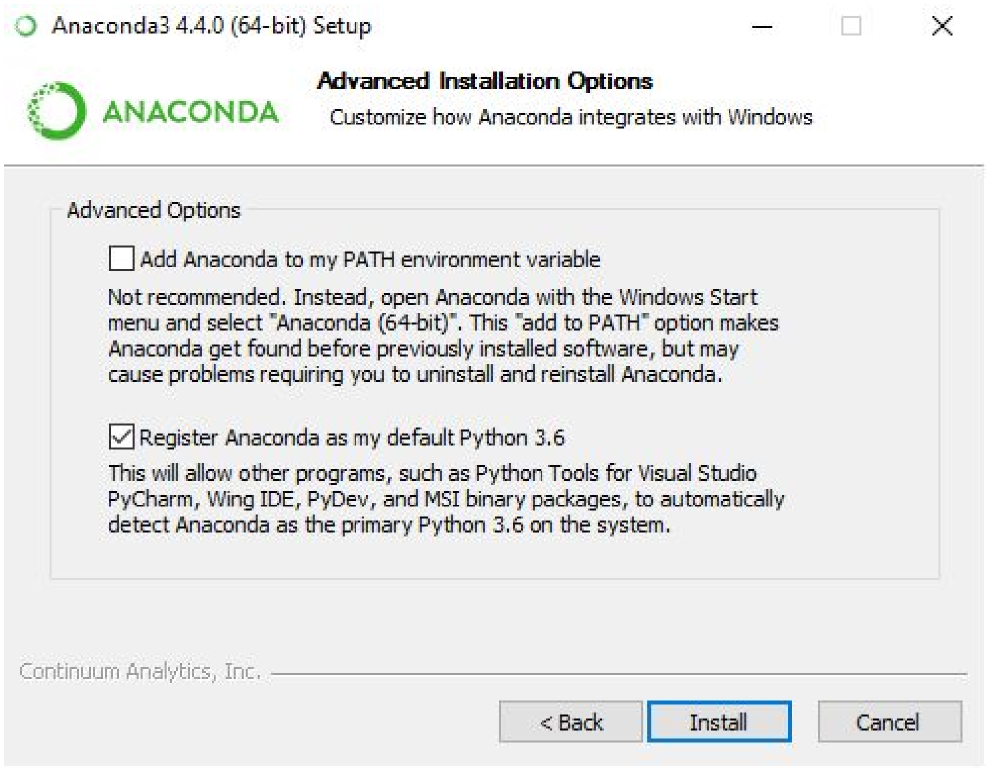
All Users (requires admin privileges)

Continuum Analytics, Inc.

If you are an OSX user, you may encounter an error that the program cannot be installed to the default location. Make sure you have selected *Install for Me Only* at this step to overcome this problem then proceed.

In the next screen, accept the default installation directory or click *Browse* to make changes if you must.

Step 7: Presented with the Advanced Installation Options, leave the options as-is and click install.


Anaconda3 4.4.0 (64-bit) Setup

ANACONDA

Advanced Installation Options

Customize how Anaconda integrates with Windows

Advanced Options

☐ Add Anaconda to my PATH environment variable

Not recommended. Instead, open Anaconda with the Windows Start menu and select "Anaconda (64-bit)". This "add to PATH" option makes Anaconda get found before previously installed software, but may cause problems requiring you to uninstall and reinstall Anaconda.

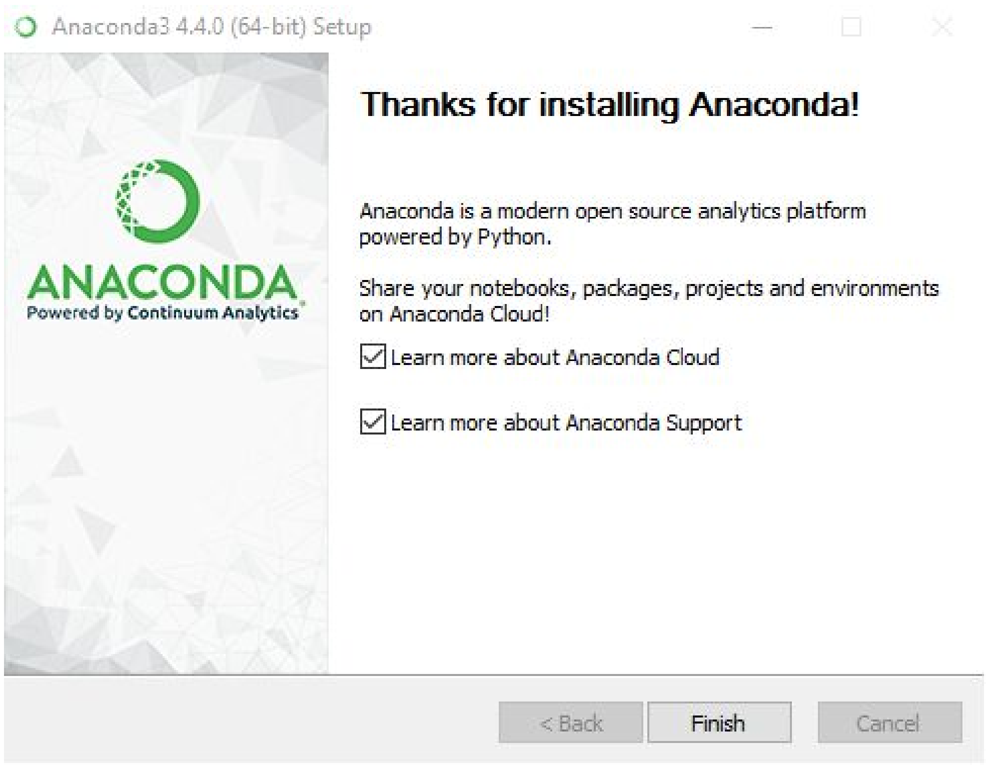
☑ Register Anaconda as my default Python 3.6

This will allow other programs, such as Python Tools for Visual Studio PyCharm, Wing IDE, PyDev, and MSI binary packages, to automatically detect Anaconda as the primary Python 3.6 on the system.

Continuum Analytics, Inc.

< Back | Install | Cancel

Step 8: When the installation of Anaconda is complete, you should see the following screen prompt:


Anaconda3 4.4.0 (64-bit) Setup

Thanks for installing Anaconda!

Anaconda is a modern open source analytics platform powered by Python.

Share your notebooks, packages, projects and environments on Anaconda Cloud!

[✓] Learn more about Anaconda Cloud

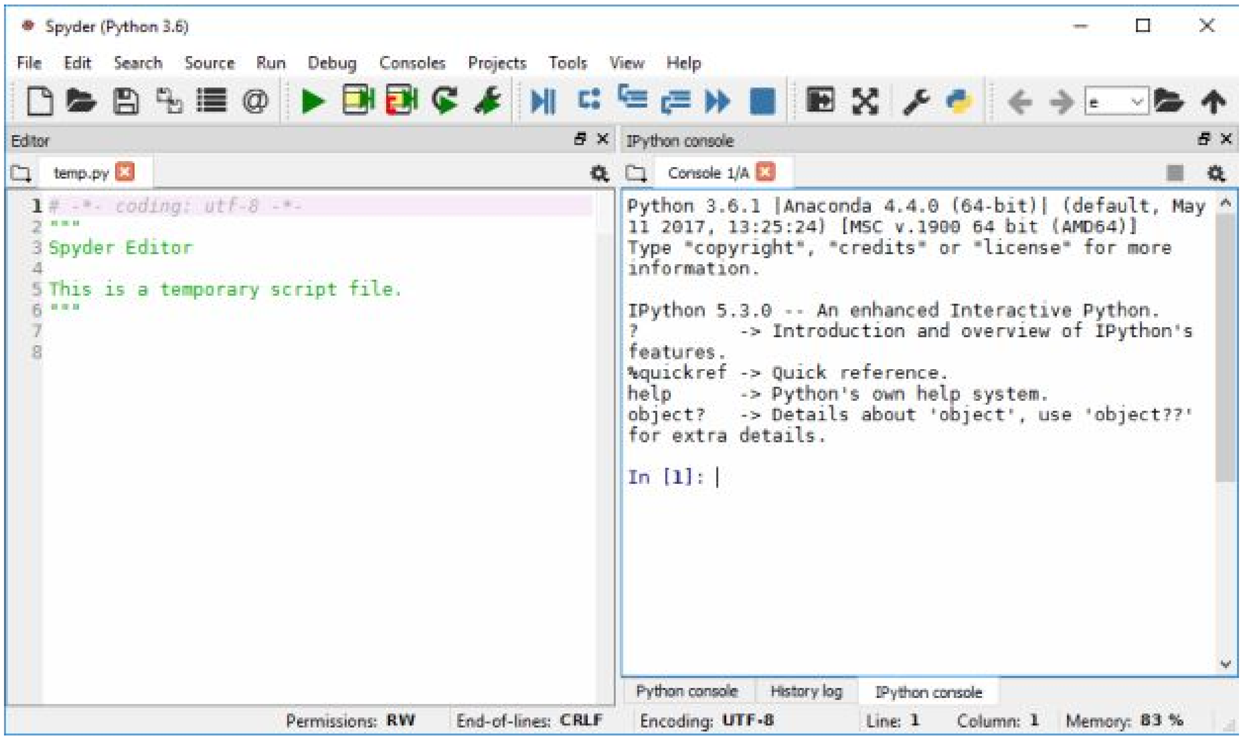
[✓] Learn more about Anaconda Support

Click *Finish* to complete the installation process.

Starting Spyder with Anaconda

Anaconda installs the Spyder Python IDE which you can access under your Applications menu. On Windows, you should find the Spyder icon and shortcut link under the Anaconda3 menu in the Start Menu and in OSX on the dock or program menu. If you use Linux, check under Anaconda3 in the installed packages.

When you start the Spyder IDE, you see its main program window which looks like this:



```
# -*- coding: utf-8 -*-
"""
Spyder Editor

This is a temporary script file.
"""
```

```
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May
11 2017, 13:25:24) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more
information.

IPython 5.3.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's
features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??'
for extra details.

In [1]: |
```

For our lessons, we suggest that you close all the open panes save for the *Editor* window on the left of the screen and the *Console* window on the right. The Editor window is where we will enter our code and the right window will display the results of the execution.

Because we have not started any Python scripts in the screenshot above, the landing editor window of a temporary script file is displayed. Do not be confused if you see something different.

Hello, World!

Since Brian Kernighan and Dennis Ritchie used it as an example in their *"The C Programming Language"* book, it has become a tradition for the first script or program a beginner to a programming language to write a Hello, World! program. For our first script, we will create and save a Hello, World! script then run it on Spyder.

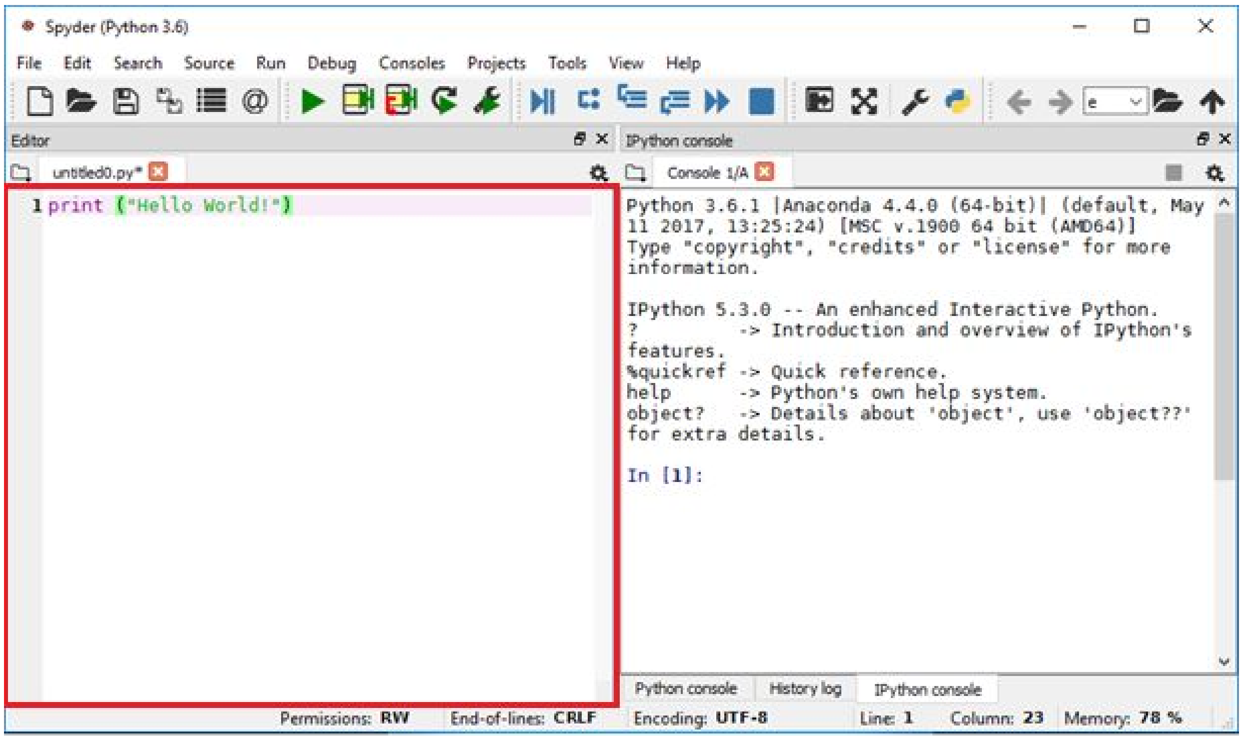
Exercise 1: Hello, World!

Start a new script on the editor window of Spyder by clicking on the new file icon, pressing Ctrl + N, or File > New. A new untitled editor tab will be added to the right of the current one. Erase all the information to have the cursor on the first line of the now blank editor.

Enter the following line of code EXACTLY as it appears without the number preceding the line.

```
1  print ("Hello, World!")
```

Your program script should appear like this:

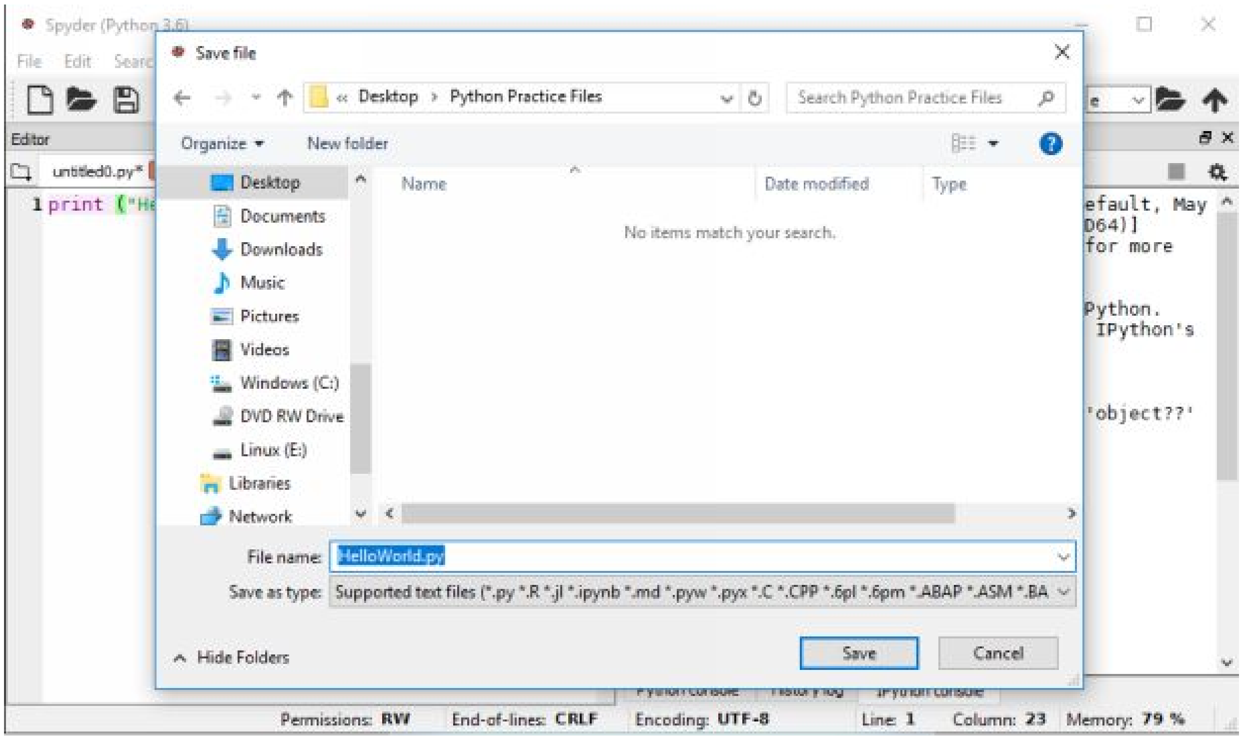


```
print ("Hello World!")
```

```
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May
11 2017, 13:25:24) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more
information.

IPython 5.3.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's
features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??'
for extra details.

In [1]:
```

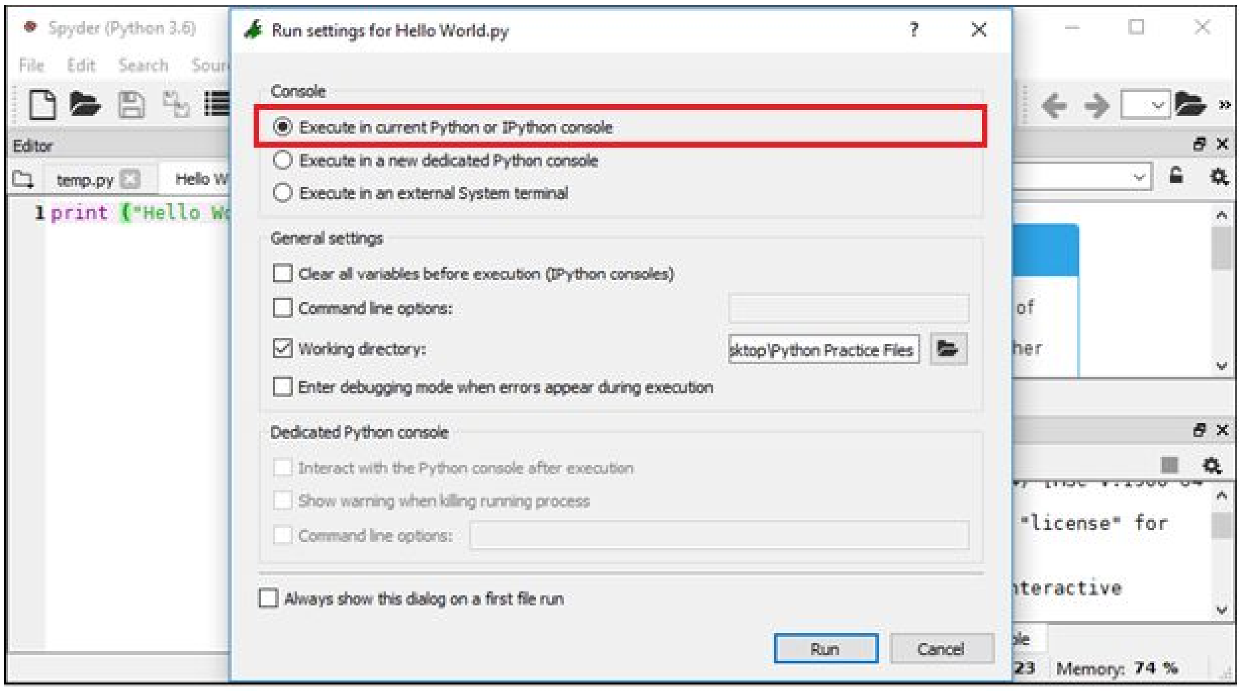
Next, we need to save the script file. Python scripts are plain text files that have an extension *.py*. When saving the file, be sure to add the *.py* extension at the end. Ignore all other suggestions of the program.

It is also important to give your script an appropriate and descriptive name for simpler future reference. For instance, our script will bear the name *HelloWorld*.py and the save screen will look like this:



Once you save a script, its tab header will change to the assigned filename. At this point, you can run the scrip by clicking on the green Run button on the run toolbar or by pressing the F5 key on the keyboard.

At first run, Spyder may prompt you to choose what run settings to use for your Hello World program.



```
print ("Hello Wo
```

Run settings for Hello World.py

Console

- [✓ Execute in current Python or IPython console]
- Execute in a new dedicated Python console
- Execute in an external System terminal

General settings

- ☐ Clear all variables before execution (IPython consoles)
- ☐ Command line options:
- ☑ Working directory: sktop\Python Practice Files
- ☐ Enter debugging mode when errors appear during execution

Dedicated Python console

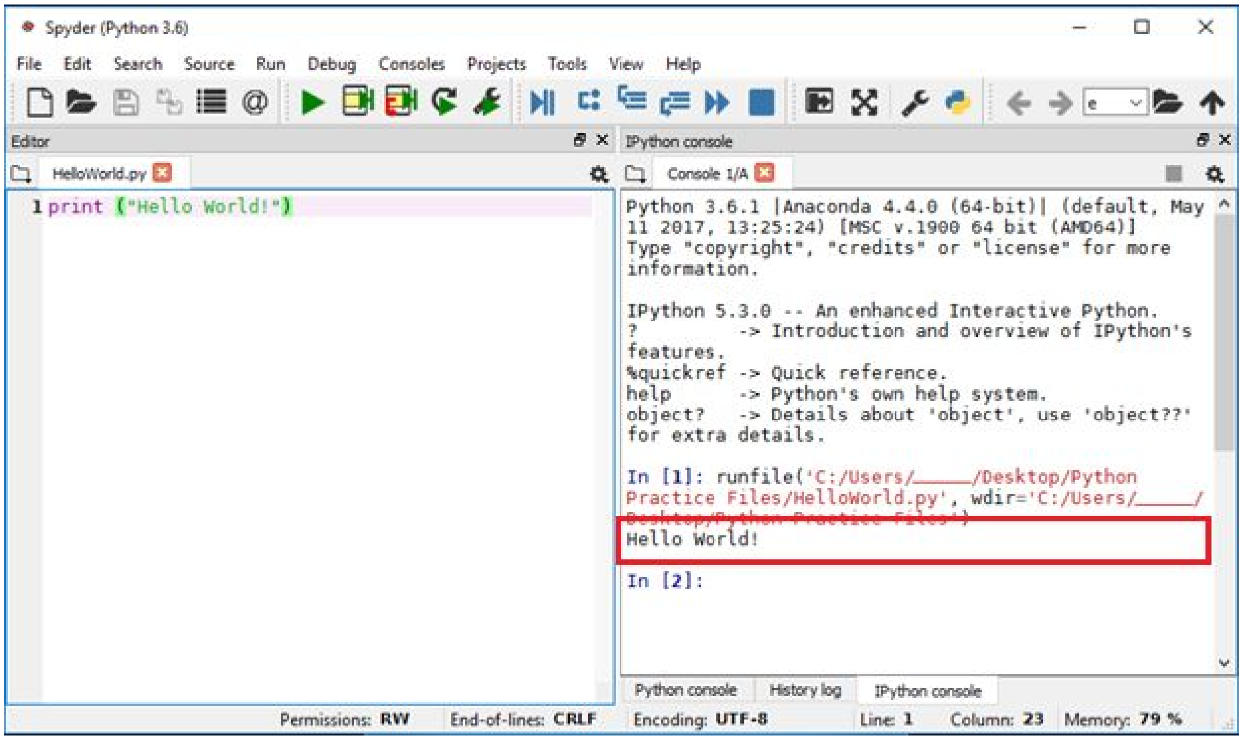
- ☐ Interact with the Python console after execution
- ☐ Show warning when killing running process
- ☐ Command line options:

☐ Always show this dialog on a first file run

Run Cancel

At this point of our course, choose to ‘Execute in current Python or IPython console’ to view the run script on the right pane of the window.

The script will be run (displayed on the IPython console window on the right of our screen.


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor
HelloWorld.py
IPython console
Console 1/A
Python console History log IPython console
Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 1 Column: 23 Memory: 79 %

```
print ("Hello World!")
```

```
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May
11 2017, 13:25:24) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more
information.

IPython 5.3.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's
features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??'
for extra details.

In [1]: runfile('C:/Users/_____/Desktop/Python
Practice Files/HelloWorld.py', wdir='C:/Users/_____/
Desktop/Python Practice Files')
Hello World!

In [2]:
```

The output screen displays more information than we need, but what is important is the displayed last line. If the console window displayed what you entered between the parentheses (brackets) and quotation marks (“”) in the editor window, then it is a success. You have officially earned the title “Programmer”!

Interactive vs script modes of programming

Python script mode programming

For our first exercise, we entered our code in a code editor then saved the code script before running it from the Spyder interpreter window. This form of programming is known as script or scripted programming.

For most of our exercises, we will save a script with our practice code and execute it from the Spyder window. The Python interpreter continues to run a loaded script until it reaches the last line, then the interpreter becomes inactive.

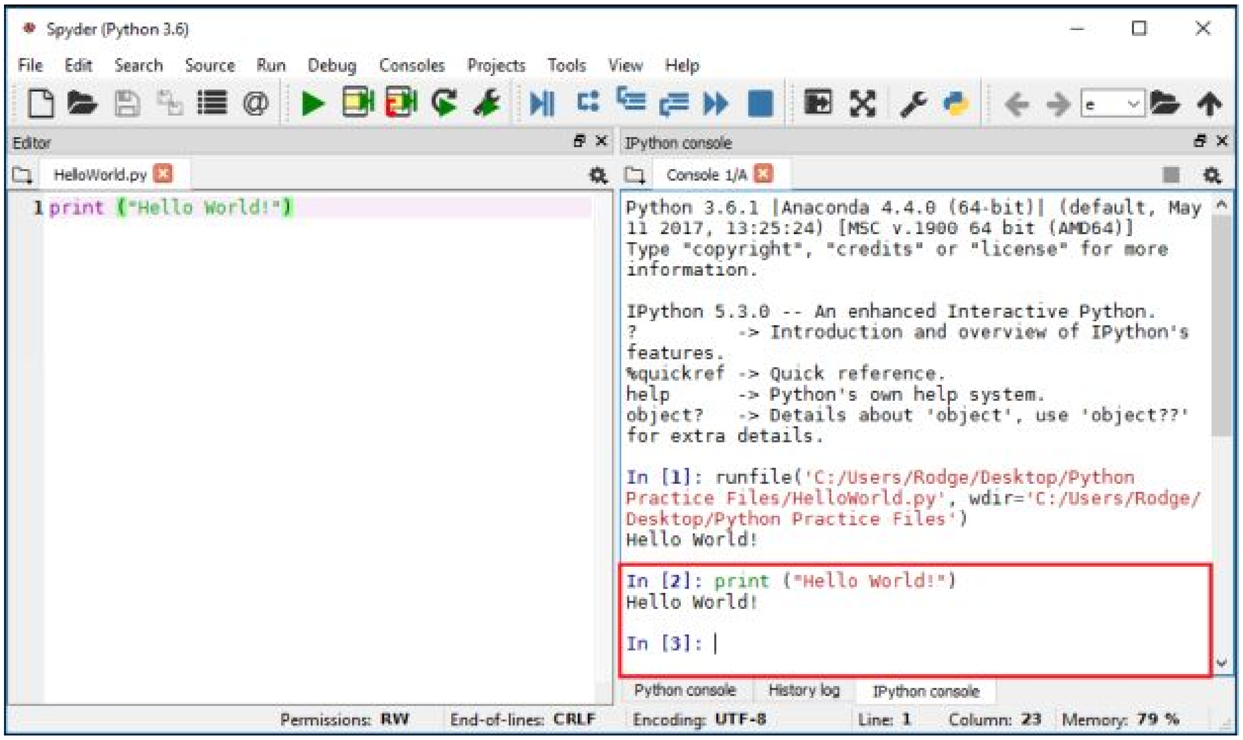
Python interactive mode programming with Spyder

Alternatively, we can write Python code directly on the interpreter and get it interpreted and executed immediately we press ***Return (Enter)*** without saving the script. This mode of programming is called interactive because the interpreter is invoked without passing a script file.

Click on the console window of our Spyder program next to the last **In** line colored blue. There should be the word **In** followed by a number in square brackets. Type the following line of code and press ***Return (Enter)***.

```
print ("Hello, World!")
```

You will notice that it produces the same result as when we saved a script with the same code before running it with the interpreter.


Spyder (Python 3.6)
File Edit Search Source Run Debug Consoles Projects Tools View Help
Editor
HelloWorld.py
IPython console
Console 1/A
Python console History log IPython console
Permissions: RW End-of-lines: CRLF Encoding: UTF-8 Line: 1 Column: 23 Memory: 79 %

```
print ("Hello World!")
```

```
Python 3.6.1 |Anaconda 4.4.0 (64-bit)| (default, May
11 2017, 13:25:24) [MSC v.1900 64 bit (AMD64)]
Type "copyright", "credits" or "license" for more
information.

IPython 5.3.0 -- An enhanced Interactive Python.
?         -> Introduction and overview of IPython's
features.
%quickref -> Quick reference.
help      -> Python's own help system.
object?   -> Details about 'object', use 'object??'
for extra details.

In [1]: runfile('C:/Users/Rodge/Desktop/Python
Practice Files/HelloWorld.py', wdir='C:/Users/Rodge/
Desktop/Python Practice Files')
Hello World!

In [2]: print ("Hello World!")
Hello World!

In [3]: |
```

For some exercises later in this book, we will ask you to use the interactive mode of programming to get interpreter output without saving a script file.

The 10 basic style guides for Python code

Python demands one thing before all else from a programmer: **CONSISTENCY**.

Before we can venture further into writing Python code and learning all the elements that make Python such a powerful programming language, there are a few coding conventions that come standard with every Python distribution. The Python style guide is all about consistency—the code must be consistent across a project and within a module or function. Note that these style guides evolve over with the language as additional conventions are discovered and older ones rendered obsolete.

The following are the most important Python coding guidelines and convention styles you must know and abide by at this point:

! Note: Do not strain to understand this logic at this point. Just understand the rule and we will learn how and why to apply them at a later chapter.

1. Python identifiers

An identifier in Python refers to the identity or name of a variable, module, class, function or other objects. Every identifier must begin with an alphanumeric letter (A to Z, a to z, or 0 to 9) or an underscore_ only or trailed by more uppercase or lowercase letters, numbers 0 to 9, or underscores.

Punctuation characters and other symbols such as $, @, !, and others are not valid characters to include in an identifier.

2. Case sensitivity

Python is case sensitive. This means that a letter in lowercase is not equal to the same letter in uppercase. **Age** and **age** are two different identifiers in Python the same way **Print** and **print** are totally different phrases.

3. Reserved words

Python has a list of words in lowercase that are reserved for the interpreter and cannot be used as identifiers. Here is a table of the reserved words:

and	assert	break	class	continue	def
del	elif	else	except	exec	finally
for	from	global	if	import	in
is	lambda	not	or	pass	print
raise	return	try	while	with	yield

4. Indentation

Unlike other high level programming languages such as C++ and Java, Python does not use curly braces or square brackets to group similar lines of code into a block. Instead, we use indentation. Simply indent similar lines of code using either tabs or four spaces. Related consecutive lines of code will be automatically identified as a block by the interpreter.

Note: Indentation is a very serious matter in Python. Proper ethics dictate that you either get used to using four spaces for a single level of indentation or consistently use tabs. You cannot use tabs in some areas and spaces in others, there must be uniformity and consistency in line indentation and this is rigidly enforced.

5. Multi-Line Statements

Again, unlike other high-level OOP languages, Python does not use a colon, semi-colon, or any other symbol to denote the end of a line. Statements typically end with the beginning of the next line. However, it is possible to write a single statement that extends over multiple lines by using the continuation character or backslash (\). However, statements contained within square brackets [], curly braces {}, or parentheses () do not need the line continuation character to extend beyond one line.

6. Multiple statements on a single line

You can enter multiple statements on a single line in Python by separating them with a semicolon (;). In such a case, neither of the two statements starts a new block and are a part of the same block. You cannot combine two statements of different blocks in a single line of code.

7. Quotation

You can use the single ('), double ("), or triple (''' or """) quotation marks to denote string literals in Python. The important rule is that you must use the same type of quotation mark at the start and end of the string. The triple quotation marks are used when the string literal spans across multiple lines as we will learn later on.

8. Comments

The comments you leave on the code to guide you and other users when reviewing the code begin with the hash sign (#) to the end of the line. A comment can occupy an entire line or start at the end of a statement code. The Python interpreter ignores all comments in a script.

9. Blank lines

Your Python script may contain blank lines with only whitespaces called blank lines. Just as with comments, the Python interpreter completely ignores blank lines in a program script. During interactive python programming, you may be required to enter an empty line to terminate a multiline statement.

10. Waiting for the user

When your Python script requires a user input at some point, the interpreter will stop execution at the particular point in the script and display a system or pre-defined prompt message and/or a blinking cursor as it awaits the user to take action. It is a good habit to create one or two lines before the actual prompt display to draw the user to where matters and to keep the console window active until the user is done with the application.

These are the ten essential basic style guiding points you need to know as you dive into more serious code-learning with Python. Make a point of reading these points frequently and notice how steeply your learning curve will rise.

Chapter Summary

Google has put together a very impressive style guide summary for beginners to Python. It is essentially a list of dos and don’ts for programming with Python and how to ensure consistency.

You can access this simplified but extensive guide on the Google Code website.

Chapter 3: Python Variables and Basic Data Types

We refer to Python as an object-oriented programming language because it is not “statically typed”. This means that we do not need to declare variables or their types before we use them in our programs.

Fixed values such as numbers, letters, and strings are considered “constants” because their values do not change while variables are the containers that hold them and may change based on the constants in use at the time.

1. Variables

A variable is basically a named memory location reserved to store values of a defined type. When you create a variable in Python, in essence you alert the interpreter to allocate some memory space, the size of which is determined by the data type of the variable.

The interpreter will determine what can be stored in the reserved memory hence by assigning variables different data types, you create variables in which you can store numbers, string characters, item lists, etc.

Assigning values to variables

You do not need to explicitly declare a variable to have a memory space reserved for it; the declaration will happen automatically when a value is assigned to the variable using the equal sign (=).

```
variable = value
```

In this statement declaration, *variable* is the left operand while *value* is the right operand. All variable value assignments take this format. The identifier of the variable goes to the left of the assignment operator (=) while its value is placed to the right.

Exercise 2: Creating a variable

Create a new script tab on the Spyder IDE and enter the following script exactly as it appears and run it to demonstrate how to assign a value to a variable.

```
name = "John"
age = 21
gender = "Male"

print ("Name: " + name)
print ("Age: " + age)
print ("Gender: " + gender)
```

In this exercise, we have created three variables – *name*, *age*, and *gender* by assigning them values *"John"*, *21*, and *"Male"* respectively. To ascertain this, we have displayed their values on the console using the print() function.

Multiple assignment of values to variables

Python is a very flexible language that is constantly evolving to make it more user-friendly and easy to use. One of the top new features of this language is that it allows you to assign a single value to several variables simultaneously and even assign multiple variables different values with one line of code. Consider Exercise3:

Exercise 3: Assigning multiple variables different values

```
name, age, gender = "John", 21, "Male"
print (name, "is a", age, "year old", gender)
```

Note, that you can assign values of different data types to multiple variables in one statement. In the exercise above, multiple variables with different data types are created simultaneously because all out variable identifiers make up the left operand of the assignment operator and the values make up the right operand.

Exercise 4: Assigning multiple variables the same value

```
x = y = z = 10
print ("The values of x, y, z are", x, y, z, "respectively.")
```

The three objects created when the Exercise 4 script is run are named *x*, *y*, and *z* and they are all pointed to the same memory location with a numeric values of *10*. This is basically a shorter way to assign each of the variables the same value in a different line.

2. Basic Data Types: Numbers

The Number data type stores numeric values. This data type is immutable, meaning that when the value of a number constant changes, the new number will be allocated a new object.

Python supports three types of numbers: integers, floating point numbers (or simply floats), and complex numbers, which we will not cover in this book.

An integer is a null number without the fractional part while a float may be an integer with a fractional part or a fraction of a number. Python tells **int** and **float** numbers apart by the presence or absence of a decimal point between two numbers in a sequence.

Creating a number object

You can create a number object by assigning a value to an identifier using an equal sign (=). For instance in the exercise below, we will create two variables named *myAge* and *myScore* with number data types.

Exercise 5: Creating number types

Create a new script on the Spyder IDE and enter the following code to define two different numbers:

```
myAge = 21
myScore = 92.7

print ("I am", myAge, "and I scored", myScore, "percent.")
```

Save the script as Exercise2.py or assign it a suitable name and run it. Do you notice how the interpreter replaces the identifiers *myAge* and *myScore* with the actual numeric values in the last print statement?

Python can tell you the type of a variable or a constant when you use the **type()** function. For instance, in our example, add the following one line of code to determine the types of data held by variables *myAge* and *myScore*:

Exercise 6: Discovering number types

```
myAge = 21
myScore = 92.7

print ("I am", myAge, "and I scored", myScore, "percent.")
print (myAge, "is of type:", type(myAge))
print (myScore, "is of type:", type(myScore))
```

When you run the Exercise 6 script, the last two lines should reveal that:

21 is of type: <class ‘int’>

92.7 is of type: <class ‘float’>.

You can also use the **isinstance()** to verify that a number of a specified type or not. For instance, we can check whether *myAge* and *myScore* number types are of integer type or not using the following code:

Exercise 7: Finding number type using isinstance()

```
myAge = 21
myScore = 92.7

print ("Is", myAge, "an integer?",isinstance(myAge, int))
print ("Is", myScore, "an integer?", isinstance(myScore, int))
```

In the last two exercises, you have learnt that in Python:

You can check the type of any value or variable using the **type()** function. Similarly, you can use the **isinstance()** function to check whether a variable or value is of a given type.

Number coercion from int to float and vice versa

The process of converting a number from one type to another is known as coercion. Python can convert a number in an expression made up of mixed types implicitly to a common type for evaluation. For instance, when adding a float and an integer, the integer will be internally converted to a float.

If you need to explicitly coerce from one number type to another then you can use the **int()** and **float()** functions respectively to convert to integer and float types.

Exercise 8: Number type coercion

```
distance = 14
time = 2.75

exact_distance = float(distance)
approx_time = int(time)

print ("I walked", distance, "kilometers in", time, "hours.")
print ("I walked exactly", exact_distance, "kilometers in about",
approx_time,
  "hours.")
```

In Exercise 8, you will notice that we converted the value of *distance* (14) into a float and assigned it the variable *exact_distance* and *time*, which was a float, into an integer with a new variable name *approx._time*.

This exercise teaches us that:

- You can explicitly coerce a number of type int to a foat by calling the **float()** function and vice versa by calling the **int()** function.
- The **int()** function will truncate the decimal part of the number, not round it off.

Furthermore:

- Negative numbers, just like positive ones, will be truncated towards 0.
- Integer numbers can be arbitrarily large. Floating point numbers can be accurate up to 15 decimal places.

3. Basic Data Types: Strings

After numbers, strings are the next most popular type of data in every programming language. Data of type string is defined with a single ('string'), double ("string"), or triple ('''string''' or """string""") quotes (quotation marks).

A string can be defined as a continuous set of characters enclosed in quotation marks. It gets its name from the ‘string’ that the alphabetic, numeric, and other character types form when they are combined to form words and sentences.

The single and double quotes are similar in all respect and you can choose to make it habit to use either of them. However, because it is easier to include apostrophes within the string content when using double quotes without the need of escape characters (we will cover these momentarily), we recommend that you make it a habit to use the “double quotes” to define strings.

Creating string objects

Just as we did with number objects, string objects can be created by assigning values to a variable identifier using the equal sign (=).

Exercise 9: Creating string objects

Type the following script on your Spyder editor window, save and run it.

```
name = "John"
occupation = "programmer"
city = "Oslo"

print (name, "is a", occupation, "from", city, ".")
print (name + "is a" + occupation + "from" + city + ".")
```

This exercise teaches us one thing about the **print()** function when it comes to strings: when two values are separated by a comma, there will be a space inserted between them in the printout but when a plus operator is used there will be no space inserted between the words.

How best would you arrange the print statement in the example for perfect spacing of words and characters?

Manipulating strings

We have been manipulating strings from the first “Hello World!” program we wrote in the first exercise, so you are not new to some of the operations we will cover in this section.

Python allows for the manipulation of subsets of strings accessed using the slicing operator [ ]. An indexing system that begins at 0 at the beginning of the string and ends at -1 is used to reference the sliced subsets.

Let us try this in the next exercises.

Exercise 10: Slicing a string and accessing sliced subsets

```
myString = "Python is the king of programming."

print (myString)
print (myString[0]) #Print the first character
print (myString[-1]) #Print the last character
print (myString[1:-2]) #Print from the second to the second last characters
print (myString[:17]) #Print from start to the 18th character
print (myString[17:]) #Print from the 18th character to the last
print (myString[0:6] * 3) #Iterate from the first to the seventh character
print (myString[15:] + myString[:8])
```

You can see in the above exercise that we can slice up a string into characters and refer to each subset using an index that starts at 0. We can also concatenate different string subsets using the concatenation operator (+) and iterate using the repetition operator or asterisk (*).

Exercise 11: More functions to manipulate strings

```
Text = "Python is king. I am learning it to be king too."

print ("Length of Text: ", len(Text)) #Returns character length.
print ("Instances of the word 'king' in Text: ", Text.count('king'))
print ("Number of spaces in Text: ", Text.count(' '))
print ("Find the word 'learn' in Text: ", Text.find('learn'))
print ("Split Text string on whitespace:", Text.split(' '))

Text2 = Text.replace('king', 'queen') #Replace 'king' with 'queen' in Text
print (Text2)
print ("Change Text to UPPERCASE: " + Text.upper())
print ("Capitalize Text2: " + Text2.capitalize())
```

There are many ways in which you can manipulate data of type string in Python. Exercise 11 will help you understand how to use some of them. With practice and further research, you will learn all the vital functions and methods that are applicable to your future programs.

Chapter Summary

We have learned that Python is a completely object oriented programming language. Because of this, variables can be created as independent objects on the fly.

Write a simple program that sets the variables below to their corresponding values.

myint to the value 7

myfloat to the value 1.23

my_bool to the value True

Chapter 4: Basic Operators in Python

An operator in programming is a symbol or symbols that are used to carry out logical or arithmetic computation on values of variables, also called operands.

On the right console of Spyder IDE, you can actually carry out most basic arithmetic and logical operations in interactive mode. Try these simple computations to see how it works:

```
In  [1]: 3 + 2
Out [2]: 5

In  [1]: 5 * 3
Out [2]: 15

In  [1]: 3 > 2
Out [2]: True

In  [1]: "x" == "X"
Out [2]: False

In  [1]: "now" in "Going home right now"
Out [2]: True

In  [1]: 11 % 3
Out [2]: 2
```

You will discover that you can pretty much carry out any basic calculation or comparison right on the Python interpreter.

Types of Operators in Python

To be very good at carrying out calculations and comparing operands, you must first learn about the different types of operators supported in Python. In this book, we will cover the five essential types of operators in detail.

1. Assignment operators
2. Arithmetic operators
3. Comparison operators
4. Logical operators
5. Membership operators

Assignment operators

In programming, we use assignment operators to assign values to variables. So far we have extensively used the most popular assignment operator the equal sign (=) to assign all kinds of values to different variables in our previous exercises.

```
x = 10
```

The line of code above is a reminder of how we use the equal sign operand to assign a value (the right operand) to a variable (the left operand). There are over ten assignment operators you will eventually learn to use in Python but we will only cover the basic seven in this book.

Exercise 12: Assignment operators in Python

```
x = 3
result = x

print (result)

result += x #Add AND
print ("Result of x += is ", result)

result -= x #Subtract AND
print ("Result of x -= is ", result)

result *= x #Multiply AND
print ("Result of x *= is ", result)

result /= x #Divide AND
print ("Result of x /= is ", result)

result **= x #Exponent AND
print ("Result of x **= is ", result)

result //= x #Floor divide AND
```

```
print ("Result of x //= is ", result)

result %= x #Modulus divide AND
print ("Result of x %= is ", result)
```

The table below summarizes what they do and how you can apply them on *Exercise 21*. For your practice, try the listed operators with different values and compare what they return with what you would expect from the equivalent operation equation.

Operator	Description	Syntax	Equivalent to
Assign =	Assigns values or values on the right side of the operator to that or those on the left side.	x = 3	x = 3
Add AND +=	Adds the value on the right side of the operator to the value on the left side then assigns the result to the operand on the left.	x += 3	x = x + 3
Subtract AND -=	Subtracts the value on the right of the operator from the value on the left then assigns the result to the left operand.	x -= 3	x = x - 3
Multiply AND *=	Multiplies the value on the right side of the operator by the value on the left side then assigns the result to left operand.	x *= 3	x = x * 3
Divide AND /=	Divides the value on the left side of the operator by the value on the right then assigns the result to left operand.	x /= 3	x = x / 3
Exponent AND	Calculates the exponential of the operands then assigns the	x **= 3	x = x ** 3

****=**	result to the value on the left side of the operator.		
Floor divide AND //=	Calculates a floor division of the value on the left of the operator by that on the right and assigns the result to the left operand.	x //= 3	x = x // 3
Modulus AND %=	Calculates the modulus from the value on the left of the operator divided by that on the right then assigns the result to left operand.	x %= 3	x = x % 3

Arithmetic operators

As the name implies, arithmetic operators are those used to carry out mathematical operations such as addition, subtraction, multiplication, and division among others. You can carry out basic arithmetic operations right on the Python interpreter in interactive mode as demonstrated at the start of this chapter.

Exercise 13: Arithmetic operators in Python

```
x = 3
y = 2.5

result = x + y
print ("The result of x + y is", result)

result = x - y
print ("The result of x - y is", result)

result = x * y
print ("The result of x * y is", result)

result = x / y
print ("The result of x / z is", result)

result = x % y
print ("The result of x % y is", result)

result = x ** y
print ("The result of x ** y is", result)

result = x // y
print ("The result of x // y is", result)
```

The next table best summarizes the seven basic arithmetic operators you should practice using in Python to be fully familiarized with how they are used.

Operator	**Description**	**Example**
Addition +	Adds the values of the two operands on either side of the operator.	x + y
Subtraction -	Subtracts the value of the right operand from the value of the left operand.	x - y
Multiplication *	Multiplies the values of the two operands on either side of the operator.	x * y
Division /	Divides the value of the operand on the left side by the value of the right operand.	x / y
Modulus %	Divides the value of the left operand by the value of the right operand and returns the remainder.	x % y
Exponent **	Takes the value of the operand on the left and raises it to the power of the value of the operand on the right.	x ** y
Floor Division //	Takes the value of the left operand and divides it by the value of the right operand and returns the quotient value of the result.	x // y

Comparison Operators

Comparison operators are used to compare two values and to return a Boolean (either **True** or **False**) and are mostly used in the decision-making structures of Python. Some of these operators may appear strange at first, but you have encountered and probably still use most of them in solving your day-to-day challenges.

Our next exercise below may jog your memory a little bit.

Exercise 14: Comparison operators in Python

```
result = "hello world" == "Hello World"
print ("Is 'hello world' same as 'Hello World'?", result)

result = 20.0 != 20.1
print ("Is 20.0 not the same as 20.1?", result)

result = 9 > 7
print ("Is 9 greater than 7?", result)

result = 5 < 4
print ("Is 5 less than 4?", result)

result = 12.5 >= 12.0
print ("Is 12.5 equal to or greater than 12.0?", result)

result = 10 < 20
```

```
17 print ("Is the value of x less than that of y?", result)
```

The script in Exercise 14 will show you that the result of any valid comparison is either True or False. We will learn more about and even get to practice with comparison operators when we cover decision making in Python in chapter 7. For now, study the tabulated summary of the comparison operators below and try using them all during your practice.

Operator	**Description**	**Example**
Equal ==	Checks whether the values of the operands on the left and right are the same.	x == y
Greater than >	Checks whether the value of the left operand is higher than the value of the right operand.	x > y
Less than <	Checks whether the value of the left operand is less than the value of the right operand.	x < y
Equal to or greater than >=	Checks whether the value of the left operand is equal to or higher than the value of the right operand.	x >= y
Equal to or less than <=	Checks whether the value of the left operand is less than or equal to the value of the right operand.	x <= y
Not equal !=	Checks whether the values of the operands on the left and right are not the same.	x != y

Logical operators

We have learnt that comparison operators check a condition by comparing two values and returns a Boolean True or False value. However, for a typical Python program, one condition is not enough and you will need to combine more than one comparison operators. This is done using logical operators.

There are three logical operators in Python: **and**, **not**, and **or**. Let us see how to use them in a simple exercise.

Exercise 15: Logical operators in Python

```
test1 = 10 > 5

test2 = 5 < 4


print ("Check test1 and test2 for True:", test1 and test2)


print ("Check test1 or test2 for True:", test1 or test2)


print ("test1 is:", test1, "while not test1 is:", not test1)

print ("test2 is:", test2, "while not test2 is:", not test1)
```

You can see from the exercise above that **and** returns **True** if both the operands it is checking are **True** and **False** if either or both of them return a False. The **or** operator on the other hand returns **True** if both or either of the operands are **True**. The **not** operator complements an operand. See the table below for a summary of these operators and how to use them.

Operator	Definition	Example
and	Returns **True** if both the left and right operand are True	x and y

or	Returns **True** if both or either the left or the right operands are True	x or y
not	Complements the operand – **True** becomes **False** and vice versa	not x

Membership operators

Membership operators are used to check whether a value or variable is present in a sequential data type such as a string, list, tuple, or dictionary. There are two membership operators: **in** and **not in**. Just like logical operators, membership check operators return the values True or False.

Exercise 16: Membership operators

```
greeting = "Hello John!"

print ("!" in greeting)
print ("Hello" not in greeting)
print ("Hi" not in greeting)
print ("Hi" in greeting)
```

Refer to the table below for a summary of what the in and not in membership operators are used for.

Operator	Meaning	Example
in	Returns True if the value being checked is found in the sequence.	x in y
not in	Returns True if the value being checked is not found in the sequence.	x not in y

Other operators

There are two other special types of operators you will encounter as you continue to learn Python beyond the basics. These are:

1. **Identity operators:** These operators are used to check whether two values or variables are stored on the same memory location.
2. **Bitwise operators:** Bitwise operators act on operands that are in binary digit form.

Now that we have discovered the five most important types of operators used in Python programming, our next step is to learn about the rest of the essential data types and to find out how these operators act on them.

Chapter Summary

Now that you know how to play around with numbers and strings, you can actually already write a program that may be useful.

Write a concise program that converts weight entered in pounds (lbs) into kilogram (kg). Note that 1lb is equivalent to 0.45359237 kg.

Chapter 5: Python Lists

Beginners to programming, in particular to Python, often have a difficult time remembering the difference between a list and a tuple data type or when to use one and not the other. Whereas both lists and tuples are containers for a sequence of objects, and each can have elements of different types (or type if it is just one), the items in lists are mutable while those in tuples are immutable (cannot be changed) as we will learn in the next chapter.

What is a list?

A list is an ordered sequence of items enclosed in square brackets [] and separated by commas. A list is mutable meaning that the values of its items and their index locations can be changed.

```
list = [item0, item1, item2, … ,item-2, item-1]
```

Lists are one of the most important data structures in Python that every functional program must have. It is a good thing that they are also one of the simplest to manipulate.

Just like with the strings we looked at in the previous chapter, lists can be accessed with indexes that begin at 0 for the first item to -1 for the last item.

Creating a list

To create a list in Python, simply enclose one or more comma-separated values between square brackets and assign it a value.

Exercise 17: Creating lists

```
Subjects = ["Math", "Physics", "Chemistry", "Biology", "History"]
myLists = ["Pencil", "Scissor", 2020, ["Fruits", "Snacks"], "Fare"]

print (Subjects)
print (myLists)
```

Did you notice that the second list in Exercise 12 has another list inside it? That type of list is known as a nested list, a list containing one or more lists in it.

Just like the string type we looked at in the previous chapter, a list type is a sequential data type in Python that we will learn to slice, concatenate, and iterate, and so on. First though, let us look at how to access its values.

Accessing list values

We will use the square brackets to slice the items in a list then access them by referring to their indexed positions on the list.

Exercise 18: Accessing values in a list

```
Subjects = ["Math", "Physics", "Chemistry", "Biology", "History", "Art"]

print ("Print Subjects[0:2]:", Subjects[0:2])
print ("Print Subjects[2:]:", Subjects[2:])
print ("Print Subjects[-3:-1]:", Subjects[-3:-1])
```

You will discover that the slicing and indexing methods we learnt when working with strings are used in the same way with lists. Perhaps you can practice creating a larger list then knock yourself out trying out all the slicing and accessing operations we have learnt so far.

Updating and deleting list values

We already mentioned that lists in Python are mutable, meaning that we can update the contents and the order of its sequential items. You can update a single or multiple elements of a list using an assignment operator (=) with the list name and index as the left operand and the new value as the right operand.

Exercise 19: Adding list items

```
Subjects = ["Math", "Physics", "Chemistry", "Biology", "History", "Art"]

print ("Current list items:", Subjects)
Subjects[3] = "Government"
Subjects[1] = "French"
del Subjects[-1]
print ("New list items:", Subjects)
```

When you run Exercise 14, you will notice that the list item "Government" is added at index 3 to replace "Biology" and second item "Physics" at *index [1]* is replaced with "French". We also used the **del** statement to remove the last item in our list. An alternate approach in deleting an item from a list is the use of the **remove()** method which we will cover briefly below.

Basic list operations with built-in functions and methods

There are a lot more operations we can carry out on a list including concatenation (joining together using +) and iteration (using the asterisk *). Generally speaking, because a list is a sequence type of data, all the operations we carried out on strings in the previous chapter can be applied to it.

Exercise 20: More operations on a list

This exercise will help reinforce what you already know about sequences by applying them to a list.

```
cont = ["Asia", "Africa", "America", "Europe", "Australia"]
oceans = ["Pacific", "Indian", "Atlantic"]

print ("Length of list 1:", len(cont))
cont[2] = "North America"
cont.append("South America")
print ("New length of list 1:", len(cont))
print ("List 1 + List 2:", cont + oceans)

oceans.remove("Indian")
print (oceans)
print ("List 2 * 3: ", oceans * 3)
print ("Are there 'ia' in either of the lists?", 'ia' in cont or Oceans)
```

Python includes a list of functions and methods that you can use to manipulate lists at an advanced level.

Exercise 21: Application of built-in functions and methods

Operation	Definition	Exercise example
Compare *cmp*	Compares the elements of two lists	cmp(cont, Oceans)
Length len	Returns the total length of the list	len(cont)
Maximum *max*	Returns the item with the most value in the list	max(cont)
Minimum *min*	Returns the item with the least value in the list	min(cont)
List *list*	Used to convert a string or tuple into a list.	list(Text)
Remove item *list.remove(item)*	Removes item from the list	cont.remove("Asia")
Item index *list.index(item)*	Returns the lowest index that item appears in the list	cont.index("Africa")
Extend list *list.extend(list2)*	Appends the contents of list2 to list	cont.extend("Antarctica")
Count items *list.count(item)*	Returns count of how many times item appears in list	cont.count("Europe")
Reverse list *list.reverse()*	Reverses the order of items in the list	cont.reverse()
Append item *list.append(item)*	Appends object item to list	cont.append("Arctic")
Sort items *list.sort([arg])*	Sorts objects of list as per the argument(s) if provided	cont.sort(0)
Insert item *list.insert(index, item)*	Inserts object item into the list at indicated index	cont.insert([6], "Madagascar")

Populate *list.pop(item=list[-1])*	Removes and returns last object or item from list	cont.pop("India" = oceans[-1])

Chapter Summary

Suppose list1 is [4, 2, 2, 4, 5, 2, 1, 0], which of the following statements are correct?

a) print(list1[0]) b) print(list1[:2]) c) print(list1[:-2]) d) print(list1[4:6])

Which of the following statements can you use to create a list?

a) list1 = list() b) list1 = [] c) list1 = list([1, 2, 3]) d) list1 = [1, 2, 3]

Chapter 6: Python Tuples

Tuples are a lot like lists in Python, after all, they are both sequential data types, except that tuples are immutable (the items in the sequence cannot be changed) while lists are mutable. Tuples are made up of items separated by commas and enclosed in parentheses () while lists are enclosed in square brackets.

What is a tuple?

Tuples are used to group any number of items into a single sequential compound value.

Whereas lists are used in situations where you have a homogenous whose length is not known or not fixed, tuples, because its sequential elements cannot be changed, are used in situations where the positions of each elements is crucial to the use of the list.

Creating a tuple

The process of creating tuple objects in Python is the same as those of creating objects of any other type such as numbers, strings, or lists we have already covered. Simply place values separated by commas and enclose them in parentheses or brackets.

The syntax takes this format:

```
Tuple_Identifier = (Item1, Item2, Item3, … , Item-2, Item-1)

Tuple_Identifier = Item1, Item2, Item3, … , Item-2, Item-1
```

The enclosing parentheses are not mandatory to create a tuple as we will see in *Exercise 17* below. In fact, any set of values in a sequence separated by commas and written with no identifying symbols such as square brackets to make them a list, default to a tuple. However, it is a good practice to always enclose the sequence items in parentheses.

Exercise 22: Creating tuples

```
weekdays = ("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat")
weekends = "Sun", "Mon"
years = (2017,)
decade = ()

print (type(weekdays), type(weekends), type(years), type(decade))
```

Note that when you create a tuple with a single element, it must have a comma like the *years* tuple in exercise 17. You can create an empty tuple by identifying a pair of parentheses containing nothing as we created the *decade* tuple in the previous exercise.

Accessing tuple values

Like with strings and lists, the values of a tuple are sequentially arranged and can be accessed by indices starting at 0 for the first item to -1 for the last item. We use square brackets [ ] to slice and manipulate tuples in as many ways as we did lists.

In the next exercise, you will learn that we access and manipulate the elements of a tuple data type just as we did strings and lists.

Exercise 23: Accessing the values of a tuple

```
weekdays = ("Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat")

print (weekdays)
print ("First day of the week is", weekdays[0])
print ("Print items [2:]:", weekdays[2:])
print ("What is item index [-1]?", weekdays [-1])
print ("Print from first to second last items:", weekdays[:-2])
```

You can see from Exercise 23 that the system of indexing we were introduced to with string object types is still used for the tuple types. Remember that the index location of the first item is 0 and not 1 and that accessing items [0:3] means the first (inclusive) through the fourth (exclusive). Our argument here, for instance would access items 0, 1, and 2 of the tuple.

Updating tuple values

We have already learned that a tuple is an immutable data type, meaning that its values cannot be changed, updated or deleted. However, should you need to make changes to the values of a tuple, you will have to take the portion of items you need from an existing tuple and create a new one. Let us try this in Exercise 24.

Exercise 24: Creating a new tuple object from values of existing tuples

```
1  tuple1 = (2002, 2006, 2010, 2014)
2  tuple2 = ("Brazil", "Italy", "Spain", "Germany")
3
4  #tuple1[0] = 2000 #This statement will generate an error.
5
6  tuple3 = tuple1 + tuple2
7  print (tuple3)
```

Exercise 24 above demonstrates how you can combine the values of two tuples to create a new tuple. Line 4 of the script is commented out but if you uncomment and run the script again you will get type error: *TypeError: 'tuple' object does not support item assignment*

Practice creating new tuples from specific value indexes rather than whole tuples as we did in this exercise.

Basic tuple operations with built-in functions and methods

Python comes with a set of tuple functions and methods that you should discover and use during your practice to understand what each does. The table below summarizes what they are and how they are used.

You can concatenate tuples using the add (+) operand and iterate using the asterisk (*) just like you can strings and lists, except that the results of such manipulation is a new tuple and not a string. Here is a table with examples of the operations you can try to understand how the various basic tuple operations work.

Exercise 25: Basic tuple operations

Operation	Description	Exercise example
Length (len)	Returns the length (number of items) of the tuple	len(weekdays)
Concatenate (+)	Combines two or more tuples to create a new one	tuple1 + tuple2
Repetition (*)	Repeats the tuple the defined number of times	tuple1 * 3
Membership (in)	Checks for the presence of a value in the tuple	"Fri" in weekdays
Comparison (cmp)	Compares the values of two tuples.	cmp(tuple1, tuple2)
Iteration for	Loops through the values of the tuple until a defined condition is met	for "Sun" in weekdays: print "YES"
Sequence (seq)	Converts a list or string into a tuple	tuple(Continents)
Maximum (max)	Returns the item with the highest value in the tuple	max(tuple1)
Minimum (min)	Returns the item with the lowest value in the tuple	min(tuple1)

Chapter summary

Suppose t = (1, 2, 4, 3), which of the following statements are incorrect?

a) print(t[3])   b) t[3] = 45   c) print(max(t))   d) print(len(t))

What will be the output of the following script?

```
my_tuple = (1, 2, 3, 4)
my_tuple.append( (5, 6, 7) )
print len(my_tuple)
```

a) 1   b) 2   c) 5   d) Error

Chapter 7: Python Dictionary

All the sequential data objects we have studied so far – string, list, and tuple – have single values that can be accessed with integer indices.

What is a dictionary?

A dictionary, or just ***dict***, though containing sequential data as well, is an associative array mapping type with mapped keys as values. The data it contains may be in sequence but they are not sorted.

The elements of a dictionary are mutable and can be of any other data type. Each is made up of a *key:value* pair matched by a colon (:) and just as with lists and tuples, separated by commas. The syntax of a dictionary is an identified object made up of one or more elements enclosed in curly braces {}.

The syntax takes this format:

```
myDict = {key1:value1, key2:value2, key3:value3}
```

Creating a dictionary

The most popular way to create a dictionary object in Python is to name an empty dictionary object in the same way we have created other object types throughout this book. Once the dictionary is created, you can add key:value pairs as necessary.

Exercise 26: Creating a dictionary then adding key:value pairs

```
1 myDict = {}

2

3 myDict ["Name"] = "Jane"

4 myDict ["Age"] = 21

5 myDict ["City"] = "London"

6 myDict ["Hobbies"] = "Cycling", "Coding", "Cooking",
"Gaming",

7

8 print (myDict)
```

In this exercise, we created an empty dictionary called myDict in the first line of the script. From the third to the sixth line, we added *key:value* pairs to the empty dictionary then at the end printed out the contents of the dictionary ***myDict***.

Another way to create a dictionary is to explicitly assign key:value pairs a named dictionary object as in exercise 27.

Exercise 27: Creating a dictionary by assigning key:value pairs

```
1 myDict = {["Name"] = "Jane", ["Age"] = 21, ["Gender" = "F"]
["City"] = "London"}
```

```

print (myDict)
```

You will notice that it does not matter what order the key:value pairs are entered into the dictionary, because they are accessed by keys and not indices, they are always printed in a different order.

One rule you must observe when creating a dictionary is that the keys must be unique but the values do not need to be. The keys are immutable while the values are may be of mutable data types such as strings, numbers, or lists.

Accessing dictionary *key:value* pairs

Dictionary data is not sorted, hence they can be accessed independently without indexing as we did with strings, lists, and tuples. The dictionary is a compound object type that cannot be sliced or indexed.

Python uses complex algorithms that map the keys of a dictionary. This makes accessing, retrieving, and updating dictionary key:value pairs much faster than other sequential object types by this process known as hashing.

The keys are used to access dictionary key:value pairs. The next exercise demonstrates how we can access the values of dictionary *weather*.

Exercise 28: Accessing dictionary key:value pairs

```
weather = {["Mon"] = "Cloudy", ["Tue"] = "Sunny", ["Wed" = "Rainy"}

print (weather["Wed"])
```

If you attempt to access a key:value pair of a missing key you will encounter an error *KerError: [Key]*

Updating a dictionary

You can add new key:value pairs to a dictionary and modify or delete the values of existing entries as demonstrated in the next exercise.

Exercise 29: Updating a dictionary

```
1 weather = {["Mon"] = "Cloudy", ["Tue"] = "Sunny", ["Wed" = "Rainy"}
2
3 weather["Thu"] = "Hot"
4 weather["Wed"] = "Gloomy"
5 del weather["Mon"]
6
7 print (weather["Wed"])
```

In this exercise we have created a dictionary called weather with day:weather type key:value pairs as you can see. In line 3, we have added a new key:value pair, in line 4 we update the value of the “Wed” key and in line 5 we delete the key:value pair for “Mon”. Does your script work as it should?

The **del** statement is used to explicitly remove a single item in the dictionary; to delete the entire dictionary, use **clear**.

Basic dictionary operations with in-built functions and methods

The table in your next exercise summarizes the vital operations you can carry out on a dictionary in Python at this stage. You will notice that we have already come across most of them earlier with the previous object types we have covered. Nevertheless, try all these operations with dictionaries to understand how each is applied.

Exercise 30: Dictionary functions and methods

```
janedoe = {["ID"] = 7234, ["Height"] = "5.5 ft", ["Weight"] = "212 lbs"}
johndoe = {["ID"] = 2932, ["Height"] = "6.0 ft", ["Weight"] = "190 lbs"}
```

Operation	Description	Exercise example
Compare cmp	Compares the key:value pairs of two dictionaries	cmp(johndoe, janedoe)
Length len	Returns the total length (number of key:value pairs) in the dictionary.	len(johndoe)
String str	Returns a printable string of the key:value pairs of the dictionary.	str(johndoe)
Clear dict.clear()	Removes all the key:value elements of the dictionary dict	johndoe.clear()
Copy dict.copy()	Returns a copy of the dictionary dict	johndoe.copy()
Items dict.items()	Returns a tuple format of dictionary dict key:value	johndoe.items()

	pairs	
Keys dict.keys()	Returns a list of the dictionary dict's keys	johndoe.keys()
Values dict.values()	Returns a list of the dictionary dict's values.	johndoe.values()
Update dict1.update(dict2)	Adds the key:value pairs of dictionary dict2 to those of dict1	johndoe.update(janedoe)

Chapter summary

Dictionaries are very important to application development in Python. The fact that dictionaries carry arbitrary values for unique keys means that you can use it to organize any kind of data in powerful ways.

Suppose d = {"john":40, "peter":45}, what command would you use to delete the entry for "john"?

a) d.delete("john":40) b) d.delete("john") c) del d["john"] d) del d("john":40)

Can you find out what tasks the built-in functions all(), any(), len(), cmp(), sorted() etc. perform in dictionaries?

Chapter 8: Decision Making in Python

One of the most basic but very important aspects of programming is the ability to empower the computer to make decision based on the results of a test condition. Python, just like any other high-level programming language, comes with powerful decision-making structures also called truth-testing or branching features of object-oriented programming that use the *"if"* statement.

1. The *if* structure

In the real world, we use decision making structures that are identical to those we program computer to use. Consider the following through process that a teacher uses to determine if a student passed a test based on the score.

```
If test score is higher than 50%:
        Student passed.
```

If we are to represent this in Python, our script would look like example below:

```
if test_score > 50:
  status = "Pass"
```

When we introduced comparison operators in Chapter 4, we touched on the fact that they are primarily applied to decision-making structures. In our example above, we only have a single test condition and that is if the *test_score* is higher than 50. If the condition returns true, then the variable status is assigned the value "Pass".

The syntax for the If statement looks like this:

```
if <condition>:
        <statements if True>
```

In Python programming, any **non-zero** or **non-null** values are considered **True** while **zero** and **null** values are **False**.

Exercise 31: If decision making statement

```
test_score = 70

if test_score > 50:
    status = "Pass"

print (status)
```

The if statement is made up of a Boolean expression followed by one or more statements. However, as you can see from the exercise above, this structure does not offer an alternative to when the condition returns a False. For this we need another structure.

Note: If the suite of the single if statement has only a single line, it may go on the same line as the statement header. The structure would then take this syntax form:

if <condition>: <statements if True>

```
test_score = 70

if test_score > 50: print ("Pass")
```

2. The if… else structure

This expanded decision making structure contains an if statement that checks a condition followed by a block of one or more statements that are executed when the condition returns True. There is also an else statement beneath the if statement block that is executed when the initial condition check returns a false.

The syntax for this statement takes this form:

```
if <condition>:
    <statements if True>
else:
    <statements if False>
```

Exercise 32: If decision making statement

```
test_score = 45

if test_score > 50:
    status = "Pass"
else:
    status = "Pass"

print (status)
```

3. The if… elif… else structure

The if… else decision making structure is ideal for use in a situation where when a condition is checked, the results are only two: *True* and *False*. In such a structure, the else statement is optional and there can only be one following an *if* statement.

We can go ahead and check more than one condition following the first if statement immediately after the else statement. This means our nested if… else if… else structure would have a syntax like this:

```
if <condition>:
    <statements if True>
else if:
    <statements if True>
else if:
    <statements if True>
else:
    <statements if False>
```

Rather than use else if, we can shorten it to elif. Our new decision making structure would then look like this:

```
if <condition>:
    <statements if True>
elif:
    <statements if True>
elif:
    <statements if True>
else:
```

<statements if False>

Exercise 33: If decision making statement

```
user_input = input("Enter student's score:")
test_score = int(user_input)

if test_score <0 or test_score >100:
    status = "Invalid score"
elif test_score > 80:
    status = "Excellent"
elif test_score > 60:
    status = "Pass"
elif test_score > 40:
    status = "Average"
elif test_score > 20:
    status = "Poor"
else:
    status = "Fail"

print ("Student status:", status)
```

In the exercise 33, we apply the if… elif… else decision making structure to a system that determines whether a student has performed “Excellent”, “Pass”, “Average”, or “Poor” based on the score captured from the user. The first if statement eliminates all invalid test_scores less than 0 and greater than 100 and the last else: statement assigns the status “Fail” to any test_score that does not meet any of the other test conditions.

Python also allows you to use an if, if… else, or if… elif… else structure inside another decision making structure to form a nested structure of separate decision making structures. You will learn to apply this system when you begin creating more complex Python programs.

Chapter Summary

In plain English, an if statement could be read as, “If this condition is True, execute the code in the clause.” In Python, an if statement consists of:

The if keyword

A test condition (an expression that evaluates to True or False)

A colon

The if clause starting an indented block of code on the next line.

The source code below is of a Python Program that checks if a number is a prime number. Can you explain how it works?

```
no=int(input("Enter number: "))

k=0

for i in range(2,no//2):

    if(no%i==0):

        k=k+1

if(k<=0):

    print("Number is prime")

else:

print("Number isn't prime")
```

Chapter 9: Loops in Python

Statements in a Python script are generally executed in the sequence they are written – the first line is executed first and the last will be executed last. However, in a situation where a block of code needs to be executed multiple times such as until a condition is met, we use loop control structures that repeatedly executes a statement or a group of statements.

Python has two loop structures: **for** and **while**.

The *for* loop structure

The ***for*** loop structure is used to iterate over iterable objects in Python such as sequential data types (string, list, tuple, or dictionary) in a process known as *traversal*. The syntax of a for structure looks like this:

```
for <value> in <sequence>:
        <statements>
```

The *for* loop structure will iterate the sequence until the last item in the sequence is reached.

Exercise 34: The for loop

```
ages = [12, 16, 18, 11, 21, 14, 17, 20, 17]

Tot = 0

for x in ages:
    print ("Current total is", Tot)
    Tot = Tot + x

print ("Final total is", Tot)
```

Can you follow the flow of the script in the exercise above to discover how it adds up the values in the ages list and what role x plays in the loop?

Using else with the for loop

We learned that when using the if decision making structure in Python, you can include an optional else block that is executed when the test condition returns false. We can use the same else statement to be executed if the for loop exhausts the items in the sequence. The syntax of this structure will look like this:

```
for i in <sequence>
    <statements>
else:
    <statements>
```

Exercise 35: The for loop with else

```
my_numbers = [5, 9, 11, 15]

for x in my_numbers:
    print ("Current number:", x)
else:
    print ("No more numbers!")
```

The *while* loop structure

The while loop structure in Python checks a test condition first, and as long as the result is True, iterate over a block of code until the test condition returns False. This iteration structure is used when the number of times the code needs to be looped over is not known beforehand.

The syntax of the while loop structure takes this format:

```
while <condition>:
        <statements>
```

You can see from the syntax format that the test condition on the first line is checked first and the indented statements that make up the body of the while loop executed only if the test condition evaluates to True. The test condition is checked again after the first iteration and the process repeats until the test condition evaluates to False.

Exercise 36: The while loop

```
total = 0
x = 0

while x <= 10:
    total = total + x
    print ("Value of x is", x)
    x = x + 1

print("The final sum is", total)
```

In the exercise above, we use the while loop structure to add up the values of numbers from 0 (the value of x) for as long as the value of x is equal to or less than 10. When the value of x is 10, the iteration is stopped and the final sum of the numbers added together is printed on the screen.

Note: One mistake that many beginner programmers make is failing to increase the value of the counter variable (in our case x = x + 1) resulting in an infinite loop. Check to make sure that the condition being tested will return False at some point to prevent infinite looping.

Using else with the while loop

Just as with the for loop, the while loop can have an optional else block with indented statements that are only executed when the test condition evaluates to False. The syntax of the while loop in such a case would take this form:

```
while <condition>:
    <statements>
else:
    <statements>
```

Exercise 37: The while loop with else

```
counter = 0

while counter < 5:
    print ("Looped:", counter)
    counter = counter + 1
else:
    print ("Limit reached!")
```

Loop control statements

So far we have learned that the *for* and *while* loop structures in Python both involve a condition evaluation process that returns a Boolean value. The loop is terminated when the condition in this process evaluates to False and execution is transferred to the next line in the script, which can be an else: step of the loop.

There are two other statements that we can use in Python to alter this normal flow of a **for** or **while** loop: *break* and *continue*.

The *break* statement

The *break* statement is used in Python to terminate the loop in which it is contained and transfers the flow of the program to the next line of code after the loop.

If the break statement is contained inside a nested loop (in a loop inside another loop), the innermost loop will be terminated and the flow of the program will be transferred to the next outer loop.

Exercise 38: break statement in a for loop

```
i = 0

for i in range(0, 10):
  print ("Value of i is", i)
  if i >10:
    break

print ("Loop broken!")
```

Exercise 39: break statement in a while loop

```
i = 0

while i < 10:
  print ("Value of i is", i)
  i = i + 1
  if i == 6:
    break

print ("Loop broken at 6!")
```

You will notice that the *break* statement works in the same way when used with both the **for** and **while** loops.

The *continue* statement

Unlike the break statement, the continue statement does not terminate a loop. Using the *continue* statement inside a loop causes it to interrupt the normal flow of the loop by skipping the loop statements for the current iteration only.

The loop will then test the loop condition and proceed with iteration or terminate the loop if the condition returns a False.

In the next two exercises, we will use the continue statement in both the *for* and the while loops. Redo the exercises as they are then try using the statements on your own more complex loops to fully understand why they are useful.

Exercise 40: continue statement in a for loop

```
i = 0

for i in range(0, 10):
  if i == 3 or i == 4:
    continue
  print ("Value of i is", i)
```

Exercise 41: continue statement in a while loop

```
1  i = 0
2
3  while i < 100:
4    i = i + 10
5    if i == 30 or i == 40:
6      continue
7    print ("Value of i is", i)
```

In this exercise, the loop will follow the normal program flow as long as the value of i, which increases by factors of 10, is less than 100. However, within the loop body, we use the if statement to check for when the value of i will be 30 or 40 after which the rest of the loop (line 7 of the script) will be skipped.

Be sure to do a lot of practice on these statements and loops in general to avoid getting mixed up which does what.

Chapter Summary

There are no programming languages without loops, after all, we need computers mostly for repetitive tasks and they have proven to be highly effective in repeating instructions. The syntax of loop structures in Python is among the simplest of all modern programming languages.

Can you find out more about these other different kinds of loops?

- Count-controlled loops
- Vectorized loops
- Numeric ranges
- Iterator-based loops

What will the following code output?

```
for i in range(5):
    if i == 5:
        break
    else:
        print(i)
else:
    print("Here!")
```

Chapter 10: Functions and Arguments

At the beginning of this book, I pointed out that one of the top things that makes Python an amazing programming language to learn and use is that it has a vibrant community of developers who are constantly churning out reusable code you can download and use at no charge.

Your first contribution to this community may be a function. In this chapter, we will cover functions and function arguments.

Python function

In Python, a function is a block of organized and reusable code that is written for a specific purpose. The use of functions makes Python applications highly modular and each part contains highly reusable blocks of code.

In-built functions we have already used such as **print()**, **type()**, and **isinstance()** are blocks of code written to perform specific tasks. You will also create functions of your own in the future. These types of functions are generally referred to as *user-defined functions*.

Defining a function

There are five simple rules to follow when defining a function in Python:

1. The function block must begin with the keyword **def** followed by the function name and parentheses that will contain arguments or parameters and at the end a colon (:).
2. The input arguments are placed inside the parentheses following the function name. If the function has defining parameters, they will be placed inside the parentheses instead.
3. An optional documentation string called docstring will be the first statement after the defining first line. The docstring explains the purpose of the function in plain language.
4. One or more lines of code make up the function body. All the statements must be indented.
5. An optional return statement at the end exits the function. A return statement may contain optional arguments to pass back program flow to the caller. Using the return statement with no arguments has the same effect as entering return None.

The syntax for a function takes the following format:

```
def function_name(arguments):
    """Enter the function docstring here. It can be a
    single line or it can cover multiple lines."""
    function_suite
    return [expressions]
```

For our next exercise, we will create a function that takes the user’s name and generates a personalized greeting.

Exercise 42: Creating a function

```
def hello(name):
  """This is a simple function that greets the
  person whose name is passed in as a parameter"""
  print ("Hello,", name + ". How are you today?")
```

Calling a function

When we defined the function hello in Exercise 42 above, we essentially gave it a name, specified the arguments it will need to work, and within the block, gave it a purpose to print a greeting. Find out what happens when you run the Python script in which one of your functions is contained. Does the function run like any other python script? It should not.

Although the structure and content of the function is complete, we cannot see the function work unless it is called from another function or from the Python prompt. A function is called by simply entering its name along with the required parameters or arguments enclosed in the parentheses to the right of the function name.

Modify your script to call the function as we do in Exercise 43 below:

Exercise 43: Calling a function

```
myname = input("Please enter your name: ")

```

```
def hello(name):
   """This is a simple function that greets the
   person whose name is passed in as a parameter"""
   print ("Hello,", name + ". How are you today?")

hello(myname)
```

The script above first prompts the user to enter a name, after which it calls the hello function we defined in the previous exercise, passing the name the user enters as the required name parameter.

Function arguments

A function can be called in Python using one of the following four types of formal arguments:

- Default arguments
- Keyword arguments
- Required arguments
- Variable-length arguments

Keyword arguments

Keyword arguments relate very well to function calls. When a function call contains a keyword argument, the caller matches the arguments with the parameter name. This makes it possible to place arguments out of order or even skip them entirely.

In such a case, the Python interpreter will match the values provided with the parameter keywords. Refer to Exercise 44 line 6 and notice how the details of client "Jane", though not in order, still print out fine.

Default arguments

A default argument assumes a default value if the argument is not provided when the function is called.

Consider Exercise 44 below:

Exercise 44: Calling a function with default arguments

```
1  def ClientInfo(name, gender = "Male"):

2     "This function fetches and prints client name and gender"

3     print ("Client name:", name, "Gender:", gender)

4

5  ClientInfo(name = "John")

6  ClientInfo(gender = "Female", name = "Jane")
```

```
7  ClientInfo(name = "Mark", gender = "Male")
```

When we defined the function, we included a default argument for gender to be “Male”. This means that when gender is not supplied when the function is called, the default value of “Male” will be used as in line 5 of our script above.

Required arguments

Required function arguments are those arguments that are passed to the function in their correct positional order. This means that the number of arguments passed to the function during the call must be exactly the same as the number defined in the function.

Let us put in place a require arguments in our *ClientInfo* function and run it.

Exercise 45: Calling a function with required arguments

```
def ClientInfo(name):
   "This function prints client name"
   print ("Client name:", name)

ClientInfo("John")
```

The **ClientInfo** function in our exercise above requires one argument: name. However, if you supply none or more than one arguments, you will encounter a syntax error: *TypeError: ClientInfo() takes exactly 1 argument (0 given)*.

Variable length arguments

Arguments that allow a functions to process more arguments than originally defined are called variable-length arguments. These arguments are not named within the function definition like required, default, and keyword arguments.

```
def function_name(<formal_args>, (*<var_args_tuple>)):
    """Enter the function docstring here. It can be a
    single line or it can cover multiple lines."""
    function_suite
    return [expressions]
```

The tuple of arguments for the function starts with the formal arguments. An asterisk * is placed before the name of the argument that takes all the non-keyword variable arguments. The tuple may remain empty if there are no additional arguments specified when the function is called.

Exercise below demonstrate how variable length arguments are used in a function.

Exercise 46: Defining a function with a variable length argument

```
def ClientInfo(name, *comment):
   "This function prints client name"
   print ("Client name:", name, "Details:\n", comment)

ClientInfo("John", "Lifetime member.")
ClientInfo("Mary", "Tel: 555-8923", "Special client.")
```

When you run the code in the exercise above, the interpreter will correctly recognize the first passed argument as a positional argument for “name”, and others after it as variable length arguments for **comment*.

Chapter Summary

Docstrings are the most convenient way of documenting Python modules, functions, classes, and methods. Google has made a Docstrings documentation format that is generally agreed to be the new standard for professional programmers. You can check out an example here: http://sphinxcontrib-napoleon.readthedocs.org/en/latest/example_google.html

Python supports the creation of anonymous functions at runtime, using a construct called __________?

Chapter 11: Python Input and Output

Python comes with numerous built-in functions, many of which we have been accessing and using directly from the Python prompt via the Spyder IDE. So far, we have been introduced to: **print():** a very important function to output information on the default computer console; and we have used the **input()** function in a few exercises to capture user information when the Python script is running.

In this chapter, we will go into more detail the **print()** output function and the **input()** input function.

Python input with input()

Because we had not properly introduced the **input()** function in this book, most of our exercises, and possibly your practice programs, have been static. This means that the value of variables is pre-defined or are hard-coded into the program source code.

Learning to capture the user's input will add a lot of flexibility to your programs. For this, we will use the input() function which has such syntax:

input("Message prompt: ")

The "Message prompt" is an optional string of text that will be displayed on the screen asking the user to key in the required data. The program will not proceed until the user enters the data and presses the return key.

Exercise 47: The input() function

```
name = input("Enter your name: ")
career = input("What is your profession? ")
location = input("Which city are you in? ")

print (name, "is a", career, "from", location, ".")
```

Our script above captures user information in a string format. The **input()** function captures data in string format by default. However, you can convert the data type to any other format before it is assigned a variable name as in Exercise 48 below.

Exercise 48: input() type conversion

```
name = str(input("Enter your name: "))
yob = int(input("What year were you born (YYYY)? ")
gender = str(input("Are you a man or woman? ")

age = 2017 - yob
print (name, "you are a", age, "year old", gender, ".")
```

In this exercise, we have converted the year of birth to an integer so that we can use it to calculate the age of the user. You can convert captured data to about any data type supported in Python, but remember that the user must enter the right types of data. Any type mismatch will generate an error.

Note that conversion of name and gender inputs to string type is not necessary but encouraged.

Python output with *print()*

You already know that the print() function is used to output information to the display console (screen). In intermediate and advanced levels of learning to program with Python, you will learn how this print() function can be used to output information into a file.

The Syntax of the Python print() function takes this format:

*print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)*

- The **objects* argument in the syntax represents the value or values to be printed.
- The *sep* is the character that us used to separate the values. The default separator is a space character.
- The *end* argument is printed after all the other values. In this case, it defaults into a new line.
- The *file* object is where the argument values are printed, the default being *sys.stdout* or the screen.
- You do not need to concern yourself with *flush* at this point.

Exercise 49: The print() syntax

```
print ("This is the print() function")
```

This is how plain your print() output should look.

Output formatting

The *str.format()* method is an in-built feature of Python that you can use to format your strings to make it more visually appealing and to minimize mistakes.

Exercise 50: Formatting the print() function

```
x, y = 10, 20

print ("The value of x is {} while that of y is {}".format(x, y))
```

In Exercise 50 above, we have formatted our with curly braces placeholders for the values of x and y. It is also possible to use a tuple index to specify the order in which the formatted elements are printed.

Exercise 51: Formatting the print() function

```
print ("My name is {0} from {1}".format('John', 'London'))
print ("I live in {1}. My name is {0}.".format('John', 'London'))
```

You can even use keyword arguments as we learned in the previous chapter to format your text output.

Exercise 52: Formatting the print() function

```
print ("My name is {name} from {city}".format('name'='John', 'city'='London'))
```

There is a lot more output formatting options that you will discover as you gain more experience programming in Python.

Chapter Summary

At this point of this course, you have all the knowledge you need to create a fully functional program. Here is a list of websites with sample programming problems you can take on (some of them have solutions) to develop your skills.

Python programming challenges on Using Python. Practice how to design, code, and test applications such as: an average calculator, an email validator, a Bingo game, a password reset program, and a color converter.

Explore over 30 challenges on w3resource. Choose specific topics of Python to find corresponding exercises with solutions or try out the various application challenges, some which will need libraries available on Anaconda.

Five well explained Python projects for beginners from northwestern.edu. Get ideas from a web educator on how to build: a dice rolling simulator, a guess the number application, a Mad Libs generator, a text-based adventure game, and even write your own version of the legendary Hangman game.

Chapter 12: Advanced Python Programming

If you are new to programming with Python, you have had the perfect introduction to the language. On the official Python website, you will find the language defined as:

> *Python is a programming language that lets you work more quickly and integrate your systems more effectively. You can learn to use Python and see almost immediate gains in productivity and lower maintenance costs.*

Python has been around for a quarter a century, but it has only started gaining traction beyond scientific computing for the past few years. Today, TIOBE index ranks Python as the fifth most popular programming language and this can be backed up by the massive value Python developers keep adding across industries, projects, and libraries.

Python programmers are in a high demand everywhere and if you follow through to study the language at advanced levels, you too could be joining the brigade that makes the world a better place, one line of code at a time.

Practical applications of the Python language in the modern world

Because of its short learning curve, easy implementation, and more importantly a readable and clear interface for developers and project maintainers, Python is mainly used to boost the rate at which software is developed today. We have already learned that Python comes with modules that are readily available on the internet and can be used for almost any web, mobile device, or desktop applications – this is one other thing that makes Python such an amazing general purpose programming language.

Besides, applications developed in Python are highly compatible with the cloud technologies used today. This explains why Google has bet on Python to unify most of its enterprise-level software including AdWords, Google sites, YouTube, and Google Code among others.

To guide you pick the most rewarding path as a Python developer, and to help you choose the right advanced Python courses to study next, here is a summary of some of the most practical applications of the Python language today.

GUI-based desktop applications

Python is becoming the most desirable choice for developing graphical desktop-based applications because of its simple syntax, compatibility with multiple operating systems, rich text processing tools, and modular architecture. Developers use GUI toolkits such as PyQt, wxPython, and PyGtk to create highly functional applications that run on any or all the top operating systems in the market.

Image processing and graphic design applications GIMP, Inkscape, Scribus, and Paint Shop Pro are all developed in Python. 3D animation packages including Houdini, Cinema 4D, 3Ds Max, Blender, Maya, and Lightwave all use Python to varying degrees.

Scientific and computational applications and games

The availability of powerful data scrapping and analysis tools such as Scientific Python and Numeric Python, coupled with the high processing speeds of Python scripts have made the language an integral part of applications that process scientific data. For instance, 3D modelling software FreeCAD and Abaqus which is a finite element method software, are both developed in Python.

Python also offers a wide range of modules, platforms, and libraries useful in building all types of games. These tools, which you will be introduced to in your next level of study, include the popular PyGame, an essential library that brings great game functionality for game developers and PySOy, a 3D game engine that supports Python 3. There have been numerous games developed in Python, some of which you may already be familiar with. They include Civilization-IV, Vega Strike, and Disne's Toontown Online.

Web frameworks and applications

You have probably heard of CherryPy, Bottle, Flask, TurboGears, and Django. All these are frameworks that come with the standard libraries and modules that simplify demanding web tasks such as content management, database interaction, and interfacing with various protocols such as SMTP, FTP, HTTP, XML-RPC, and POP among others. Most people who proceed to study Python at advanced levels – that is beyond what you have already learned in this book – eventually begin developing web applications and websites based on one or more of these popular frameworks.

A few of the popular web applications based on Python you should check out are: a content management system called Plone, ERP5, an open-source ERP system used in apparel, banking and even aerospace applications, Odoo consolidated suite of business applications, and Google's App engine.

Script writing and automation

Python is a scripting language. This means that it allows you to write scripts for passing text files and to generate sample inputs when testing an application, and scrap content from blogs and websites easily among other tasks. Python scripts easily and painlessly replaces Bash scripts and this explains why it is a great language to automate stuff.

Python tools such as Fabric, Ansible, and Salt are often used by beginners to automate every day repetitive tasks and basic processes such as sending out promotional emails. As you can see, Python has extensive use cases that every upcoming programmer should make an effort to know about.

Machine learning and data science

Machine learning is the hottest topic right now besides blockchain technology and IoT. Luckily, this is another niche where Python is picking up steam ahead of all other modern languages. Its flexibility and open-source libraries and tools, which are powerful for data analysis, manipulation, and visualization, make it easy for developers to create practical machine learning programs within a very short time. Good examples of such tools and libraries are Seaborn and Matplotlib.

It is also worth mentioning that Python is extensible in C/C++ and as a result, it is possible to run large-scale data mining operations and web scrapping faster. Libraries such as pandas, NumPy, and scikit-learn bring the features of Matlab such as data frames, and modelling and matrix operations to Python. Additionally, machine learning algorithms such as *PyBrain, SimpleCV, OpenCV*, and *Pylearn2* are readily available for developers to use.

Become an ethical hacker or penetration tester

Because of how simple it is to learn, Python is increasingly becoming the hacker’s favorite language. We are at an age where novice hackers can follow simple tutorials on the internet to write and deploy key loggers on Windows servers. In our next book, you get a chance to learn how to hack local systems and deploy active surveillance tools on target computers, and how to take command of infiltrated computers. The book also covers network hacking demonstrations using packet sniffing and interception and man-in-the-middle attack ARP poisoning techniques. Be sure to check it out.

To wrap it up, Python is a go-to programming language for many things from animation and enterprise level web applications to data mining and prototyping. It's neat, simple and procedural and can easily be used as a functional language.

How do you plan to learn the next level of Python programming? Whatever your next step is, the Anaconda programmer suite you installed has all the tools to pull it off.

#