

PYTHON PROGRAMMING

**A STEP BY STEP GUIDE FOR BEGINNERS WITH
TIPS AND TRICKS**



SIENA SYLVIA

PYTHON PROGRAMMING

A STEP BY STEP GUIDE
TIPS AND

PYTHON PROGRAMMING

A Step By Step Guide For Beginners With Tips and Tricks

By

Siena Sylvia

ABOUT THIS BOOK

If you want to learn Python programming FAST and WELL, this book has been written to help you. Are you are just starting out with programming? You will find that this book explains complex concepts in an easy to understand way. The examples are carefully chosen to demonstrate each idea so that you can gain a deeper understanding of the language. If you are an skilled programmer, this book provides a good foundation for exploring Python.

While this book is intended to be used in an introductory programming course, it is also useful for those with previous programming experience who wish to learn Python. If you are among those individuals, you should be able to skim the first several chapters. This book contains information on the features of Python that allow you to do big things with surprisingly small code.

In preparing this book, the Python documentation at www.python.org was essential.

TABLE OF CONTENTS

[Introduction](#)

[Chapter One: What Is Python?](#)

- [What Is Python?](#)
- [The Origin Of Python](#)
- [Why Use Python?](#)
- [The Advantages Of Python Programming](#)
- [Some Drawbacks Of Python](#)
- [Common Terms You Should Know With Python](#)
- [Comparing Python To Other Languages](#)

[Chapter Two: Getting Started With Python](#)

- [Installing The Interpreter](#)
- [Access Python On Your Machine](#)

[Chapter Three: Understanding The Basics And Writing Your First Program](#)

- [Keywords](#)
- [Identifier Names](#)
- [Flow Of Control](#)
- [Semi-Colons And Indentation](#)
- [Letter Case](#)
- [Writing Our First Program](#)
- [A Bit More On Comments](#)

[Chapter Four: The World Of Variables And Operators](#)

- [What Are Variables?](#)
- [Naming A Variable](#)
- [The Assignment Sign](#)
- [Basic Operators](#)

[Chapter Five: Data Types And Type Casting In Python](#)

- [Floating-Point Numbers](#)
- [Complex Numbers](#)
- [Strings](#)
- [Boolean Type, Boolean Context, And “Truthiness”](#)
- [Built-In Functions](#)
- [Type Casting In Python](#)

[Chapter Six: Making Choices And Decisions](#)

- [Making Simple Decisions Using The If Statement](#)
- [Choose Alternatives Using The If ... Else Statement](#)
- [Using The If ... Elif Statement In An Application](#)
- [Using Nested Decision Statements](#)
- [Performing Repetitive Tasks](#)

[Chapter Seven: Error Management](#)

[Knowing Why Python Does Not Understand You](#)
[Considering The Sources Of Errors](#)
[Sorting When Errors Occur](#)

[Conclusion](#)

INTRODUCTION

We have specifically written this book to help you quickly learn Python programming and learn it well. If you are newbie to programming world, you will find that this book explains complex concepts in an easy to understand way. The examples are carefully chosen to illustrate each concept, so that you can better understand the language. If you are a professional in coding, this book provides a good foundation for exploring Python.

Getting to grips with coding can be difficult. You might have seen some of the more popular coding languages, like C ++ or Java, and were a little scared of what you saw. Maybe the pages were filled with letters and symbols that you didn't understand and were frustrated and just wanted to leave. Many people are afraid of programming and think it is very difficult for them. But by coding with the Python programming language, you will discover that it can be easier than ever to learn how to write code and even read it like a pro.

This guide will provide some of the basic concepts needed to help you get started programming in Python. We will begin to talk a bit about what Python programming is and some of the steps to download the program, if it is not already on your computer, and we will give you more information to really understand why this program is so exceptional. Next, we'll move on to some keywords that will come in handy when you start the program, and we'll also talk about the pros and cons of using Python for all of your programming and coding needs.

The remaining part of the tutorial is devoted to talking about some of the different things you can do in the Python program and some examples of how each would work. We talked about adding comments in the code, working with strings and integers and spending time talking about variables so that they will show up right in the program. It's a great idea to experiment with the process a bit. Python allows you to easily test your strings to understand what will work and what requires some more practice.

Getting started with programming can seem like a challenge. You may worry that you cannot understand everything and all those crazy programming languages may have scared you in the first place. This guide will spend more time analyzing the Python language and exploring how easy it is to get started with this simple program.

CHAPTER ONE: WHAT IS PYTHON?

The computer world has attracted many types of people. Some are interested in making money by creating their own personal programs to sell to others. Some people like to play and learn different things about how the computer works. And still others have dedicated their lives to programming, making it the product that earns their income every month, whether they work to fix computers, work in a company to safeguard computers, or do some other computer-related things.

When it comes to information technology, nothing will be easy. Before you can run a program on your computer, you need to give it the right code to make it work. There are several options for creating code that computer technology can choose from, including Java, C ++, and Python. Here, we'll explore Python a bit and why it's often preferred over the other two programming options.

Before you start using Python to meet your programming needs, it is important to start learning more about it and all the benefits you will get from using this program. Python is a high-level programming tool which means it is easy to use and read even as a newbie. The philosophy behind the language is readability and has a kind of syntax that allows the programmer to express his concepts without having any code pages in it. Compared to using other popular code, such as Java and C ++, this can make Python much easier to complete.

The philosophy of this code language is quite easy to use. It believes that a simple design is much better than a complex design and that readability is important. It is a great language for beginners as they will be really able to read and understand the code they enter. With other options, it can take a long time to try and get the right code, adding lots of other symbols to make it work. But with Python it's much simpler, and you might find it easier to read the lines and see what you're doing.

Some of the Python features that may interest you include:

- An elegant syntax that will make programs so easy to read.
- Easy to use language, so the program works without a lot of bugs. Suppose you are doing ad hoc programming activities or prototype development, as it works well without issues with program maintenance.
- It has a large library that will work with other programming tasks, such as editing files, searching for text, and connecting to web servers.
- Python is really interactive. This makes it easier to test small pieces of code to see if they work. You can also bundle it into a development environment called IDLE.
- If you want to extend the programming language, it is easy to extend it to other modules, such as C or C ++.
- Python programming can be done on any unit, including Unix, Linux, Windows, and Mac OS X.
- The software is free and you will not have to pay anything to download and use Python in your life. You can also make changes and redistribute this product. It's licensed, but it's an open-source license that others can use.
- Although Python is a simple programming language, it contains some advanced features, such as list comprehension and generators.
- Errors can be quickly detected in this schedule. Because data types are dynamically typed, when you mix types that don't match, you will notice an exception.
- Codes can be grouped into packages and modules, if required. You can choose from a wide variety of basic data types, including dictionaries, lists, strings, and numbers.

WHAT IS PYTHON?

Python is a globally-used high-level programming language created by Guido van Rossum in the late 80s. Python Programming Language strongly

focuses on the readability and simplicity of your code, allowing programmers to quickly develop applications.

Like all high-level programming languages, Python code looks like the English language that computers cannot understand. The code we write in Python needs to be interpreted by a special program called a Python interpreter, which we will need to install before we can code, test, and run our Python programs. We will see how to install the Python interpreter in the next chapter.

There are also many third-party tools like py2exe or PyInstaller that allow our Python code to package into independent executable programs for some of the most popular operating systems such as Mac OS and Windows. This let us distribute our Python programs without the need for users to install the Python interpreter.

THE ORIGIN OF PYTHON

The start of modern Python programming began in December 1989. The maker of this program was Guido van Rossum, who started programming as a hobby. At that time, Van Rossum was working on a project with the Dutch research institute CWI, which was later closed. Van Rossum was able to make use of some of the basics of this new language, known as the ABC language, to work in Python.

One of the areas where this programming language excel is that it is really easy to extend it to make it more complex or simpler, and it was able to support multiple platforms. Both were important at the time when personal computers became popular. And because Python was designed to communicate with different file formats and libraries, it became a hit as well.

Python has grown a bit from the start and more tools have been added to make programming more functional. Besides making Python easy to use, van Rossum has worked on initiatives that encourage coding education for everyone, not just a few. Using Python to code can simplify things and help get rid of some of the fears associated with complex computer code, as it doesn't sound so scary.

Over the years, Van Rossum decided to create open-source code for Python. This allowed everyone to access and make changes to Python so that if something happened to van Rossum, all was not lost. With the launch of Python, Python 2.0 was released in 2000 to make it more community-oriented and to have a transparent development process. There are still newer versions of Python 2.0 out there, but Python 3 has taken the world by storm and most predict it will be the normal version in use for years to come.

Python 3

This Python version was released in 2008. It's not just a program update, but a complete change. Even though there are a lot of cool features that come with this version, it is not backward compatible, so you must choose between Python 2.0 and Python 3. For simplicity, the programmers have made a small marker in the program that shows a programmer that must be changed between the two programs while loading. Despite this, most have preferred to stick with Python 2.0 for now.

WHY USE PYTHON?

As you will probably guess, there are several coding programs that you can decide to use. Python is one of the best options available even though there are some benefits for using other languages. It has a lot of options, it is very easy to use, and can also be used on a variety of platforms without having to change things. Some of the perks you'll love about Python include:

Readability

Python was designed to work with the English language, which makes it easier to read. There are also strict scoring rules in the program, so you don't just look at the parentheses everywhere. Python also ensures that the programmer knows how to format everything, thanks to a set of rules, which makes it easy to create code that everyone has to follow.

Libraries

Python has been in existence for over 25 years, and because it is one of the easiest codes to use, there are several codes that has been written using the system. The fact that this system is open source is a good news, so the code

is available to any programmer. You can install the Python program on your system and use it for personal use. Whether you use the codes to complete a product or to write some of its code, the Python library is easy to use. The desired codes will be installed in the libraries, and since the program has been around for a long time, they will cover pretty much anything you want from server automation to changes to an image.

Community

Because Python is so popular, the Python community is quite large. There are conferences with many networks and workshops available for these programming products and many places you can visit, both offline and online, to ask questions or to learn more about the program. You can try some of these places if you are new to Python, as it can assist you to learn more and meet new individuals.

If you want to start writing code, Python is one of the best options you can do. Getting started is easy, and since it will work on several different platforms, it will definitely work on your home computer. Since it's easy to read, you'll find that coding doesn't have to be a challenge, and you can build your own or learn from others quickly.

THE ADVANTAGES OF PYTHON PROGRAMMING

Python is definitely one of the best programming languages you can choose to use. The simplicity of activating this program will no doubt be enjoyed by beginners and start writing their own code, even without experience, and there is a lot to enjoy when you are a pro or even an expert. Some of the benefits you get when starting Python include:

Easy to use and read

When it comes to the programming language, there is nothing quite as easy to use as Python. Other languages are quite clunky and difficult to look at. You can look at them and notice that they have tons of square brackets and even some words that you don't even recognize. Just scare someone who isn't used to programming just because all the words seem a little intimidating.

Python is a little different. Instead of all the crazy parentheses, it uses indentations, which creates an easier-to-read page that isn't a

mess. Instead of words you don't understand, use English. Other special characters are all kept to a minimum, so you can see the code page and not feel overwhelmed by the process.

It is one of the simplest programming tools you can use. It looks good on the page and will use as much space as possible to make what you need to know easier to read. There are also several places with comments so you can get specifics if a program is too confusing for you. In short, it is one of the best programming languages to use to really get ahead or even learn to program.

Use English as the primary language

Since English is the language this program is based on, it is really easy to read. There aren't many words that you won't understand, and you won't need to spend time trying to figure out what it's saying to you. The program is entirely in English and you will love its simplicity.

It is already available on some computers

In most cases, Python is already available on your computer. Mac OS X systems and those with Ubuntu will already have Python preloaded. To get started, simply download a text interpreter. In terms of installing Python on the Windows operating system, all you need to do is download the program from the internet.

Python works well with all of these programs, even though it hasn't been installed from the start.

Can work with other programming languages

At first, you'll probably only be using Python on its own. It's a great program for learning and growing. But over time, you may decide to try something new that Python can't do on its own. Fortunately, Python is able to work with many other programming languages, such as C ++ and JavaScript, so you can play around, learn a little bit more, and get the code you're looking for, even if Python can't do all the work.

You can test things out with the interpreter

When downloading Python, you will also need to download a text interpreter. Text interpreter makes it a lot easier for Python to read your information. You can use simple products already on your computer, such

as Windows Notepad, or search for another interpreter that might be a bit simpler.

After you've decided on which interpreter you want to use, it's time to start writing the code. Some of those unfamiliar with the code may be concerned about trying to get it to work. This is another point where Python can simplify things. It can take the words you type and spit them out, with the help of the interpreter, in seconds. You have the ability to test what you are doing while working on it!

There are many advantages to using the Python program. Beginners will love the availability of this program and how easily they will learn a few simple commands in no time. Even those who have been planning for a while will be in awe of how it all works!

SOME DRAWBACKS OF PYTHON

While there are many reasons to love Python, it's important to realize that there are some negatives to be aware of. These negatives include:

It doesn't have much speed

For those who want to work with a program at high speed, Python is not always the best option for you. It's an interpreted language, so it will slow down compared to some of the other options which are compiled languages. However, it depends on what you are translating.

There are benchmarks with Python code that can be run faster using PyPy than other codes.

Fortunately, this slow speed issue and Python have been fixed. Programmers strive to speed up the interpretation speed of Python, so you don't have to compare it so much with others. Over time, it is hoped that Python can perform at the same speed as C and C ++ or even some of the latest published programming languages.

Not available on most mobile browsers

Python programming language is a great option to use if you have a normal computer. It is available on server platforms and many desktop to help you create the code you are looking for. But it is not ready for mobile computing. Since there is such a huge increase in income and in the number

of people entering the mobile phone industry, it is said that this programming language does not follow the trends like others.

Maybe in the future, Python will decide to go to the future and develop a version that can work well with multiple mobile devices. Until then, programmers should be happy with the usage of their desktops and laptops.

Design restrictions

If you want to work with a program that consist of a lot of design options, the Python programming language can not be the right option for you. The design language doesn't depend on what you find in some of the other options. Since you are working with a dynamically typed program, several tests are necessary and may have several errors that will not appear until you run the program.

Blocking the global interpreter means that only one thread can be accessed to access Python internally at a time. This might not be more important, as it's easy to assign tasks to different processes, but the design isn't as good as some of the other options you want.

Remembering that indentation is important in Python is a good way to work with the design. Other programming languages will use lots of parentheses to show the difference between lines and information in the program, but Python will depend on indentations. Make sure to use it to avoid problems and errors that may arise.

Python can be one of the best languages you use for writing your own code and having fun. While this program comes with a lor of benefits, especially over others that are not that easy to read, it is vital to understand the positives and negatives of each option before you start!

COMMON TERMS YOU SHOULD KNOW WITH PYTHON

Before going too deep into programming with Python, it's important to understand some of the words that can make programming easier to understand. In this section, it will take you some time to review the various common Python programming words that we have talked about a little bit in this guide, to avoid confusion and to help you start your first code.

Class: Class is a template used to create user-defined objects.

Docstring: it is a string that will appear lexically before the expression in a module, a function or a class definition. The object will be available for documentation tools.

Function: This is a block of code that is called when using a calling program. It is best to use it to provide an independent calculation or service.

IDLE: indicates the Python integrated development environment. This is the interpreter and editor environment that you can use in conjunction with Python. It's good for those who are just getting started with this and can work for those on a budget. This is a clear code example and won't waste a lot of time or space.

Immutable: it is an object in the code to which a fixed value is assigned. This can include tuples, strings, and numbers. You cannot edit the object and will need to create a new object with a different value and store it first. In some cases like dictionary keys, this may come in handy.

Interactive: One thing a lot of Python newbies love is that it's so interactive. You can try different things on the interpreter and see how they react immediately to the results. It's a good way to improve your programming skills, try out a new idea you have, and more.

List: This is a built-in Python data type. It contains an editable sequence of ordered values. It may also include immutable values of strings and numbers.

Modifiable: These are the objects which can change their value in the program, but which can keep their original id().

Subject: In Python, this is all data with a state, such as an attribute or a value, as well as a defined behavior or method.

Python 3000: Python 2 and Python 3 are the two main types of Python available. Many people have joined Python 2 because Python 3 has no previous functionality and likes to use databases from the previous version. Python 3000 is a mythical Python option that allows this functionality in reverse, so you can use it and Python 2.

String: This is among the most basic types you'll find in Python, which will store text. In Python 2, strings store text so that the string type can be used to store binary data.

Triple-quoted string - This is a string with three instances of single or double-quotes. There may be something like `''' I love tacos' '`. They are used for several reasons. They can help you to have single and double quotes in a string and they make it simpler to go over a few lines of code smoothly.

Tuple: This is a data type that was incorporated into Python. This data type is an immutable ordered sequence of values. The sequence is the only immutable part. It can contain editable values, like a dictionary, in which the values can change.

Type: this is a category or type of data represented in programming languages. These types differ in their properties, including immutable options and the mutable options, as well as their methods and functions. Python includes some of them, including dictionary, tuple, list, floating-point, long, integer, and string types.

COMPARING PYTHON TO OTHER LANGUAGES

Comparing two languages can be dangerous, because choosing a language is a matter of taste and personal preference, as is a quantifiable scientific fact. Therefore, before being attacked by the rabid protectors of the languages that follow, it is important to realize that I also use several languages and that I discover at least some degree of overlap between them. There is not the best language in the world, just the language that is best for a specific application.

C

Many people claim that Microsoft copied Java to create C#. That said, C# has some advantages (and disadvantages) compared to Java. The main (undisputed) intention behind C # is to create a better kind of C / C ++ language - a language easier to learn and use. However, we are here to talk about C # and Python. Compared to C #, Python has the following advantages:

- Much easier to learn
- Smaller code (more concise)
- Fully supported as open-source
- Better multiplatform support
- Easily allows the use of multiple development environments
- Easier to extend with Java and C / C ++
- Improved scientific and technical support

Java

For years, programmers have been looking for a language they could use to write an application once and run it anywhere. Java is designed to work properly on all platforms. He has some tips that you will find later in the book to achieve this magic. For now, you must know that Java has been so successful everywhere that other languages have tried to copy it (with more or less success). Even in this case, Python has significant advantages over Java, as shown in the following list:

- Much easier to learn
- Smaller code (more concise)
- Improved variables (storage boxes in the computer's memory) that can hold different types of data depending on the needs of the application during execution (dynamic typing).
- Development time is faster.

Perl

PERL originally stands for Practical Extraction and Report Language. Today, people call him Perl and let him go. However, Perl still has its roots in that it excels at getting data from a database and presenting it as a report. Of course, Perl has been expanded to do a lot more than that - you can use it to write all kinds of applications. (I even used this for a web service application.) Compared to Python, you'll find that Python has the following advantages over Perl:

- Easier to learn
- Easier to read
- Enhanced data protection
- Enhanced Java integration

- Less platform-specific biases

CHAPTER TWO: GETTING STARTED WITH PYTHON

Since we know a portion of the advantages of picking this program, the time has come to get started with it. Before you can get familiar with a portion of the extraordinary advances that are expected to make this program to create code for you, the time has come to set up the environment. For some people who have a PC with Mac OS X or Ubuntu, you will already have Python installed on the system. This can make things simpler to begin as you will simply need to tap on the symbol to begin.

Windows PCs will need to install Python from scratch. While Python works fine and dandy on Windows PCs, it doesn't come preinstalled so you should do this.

INSTALLING THE INTERPRETER

To create applications, you need another application unless you want to get a low level and write applications in machine code - a tough experience that even real programmers avoid as much as possible. Writing an application using the Python programming language requires some necessary applications. These applications allow you to work with Python by creating Python code, providing the necessary help information, and allowing you to execute the code you are writing. This chapter helps you get a copy of the Python application, install it on your hard drive, look for installed applications so you can use them, and test your installation to see how it works.

Download the version you need

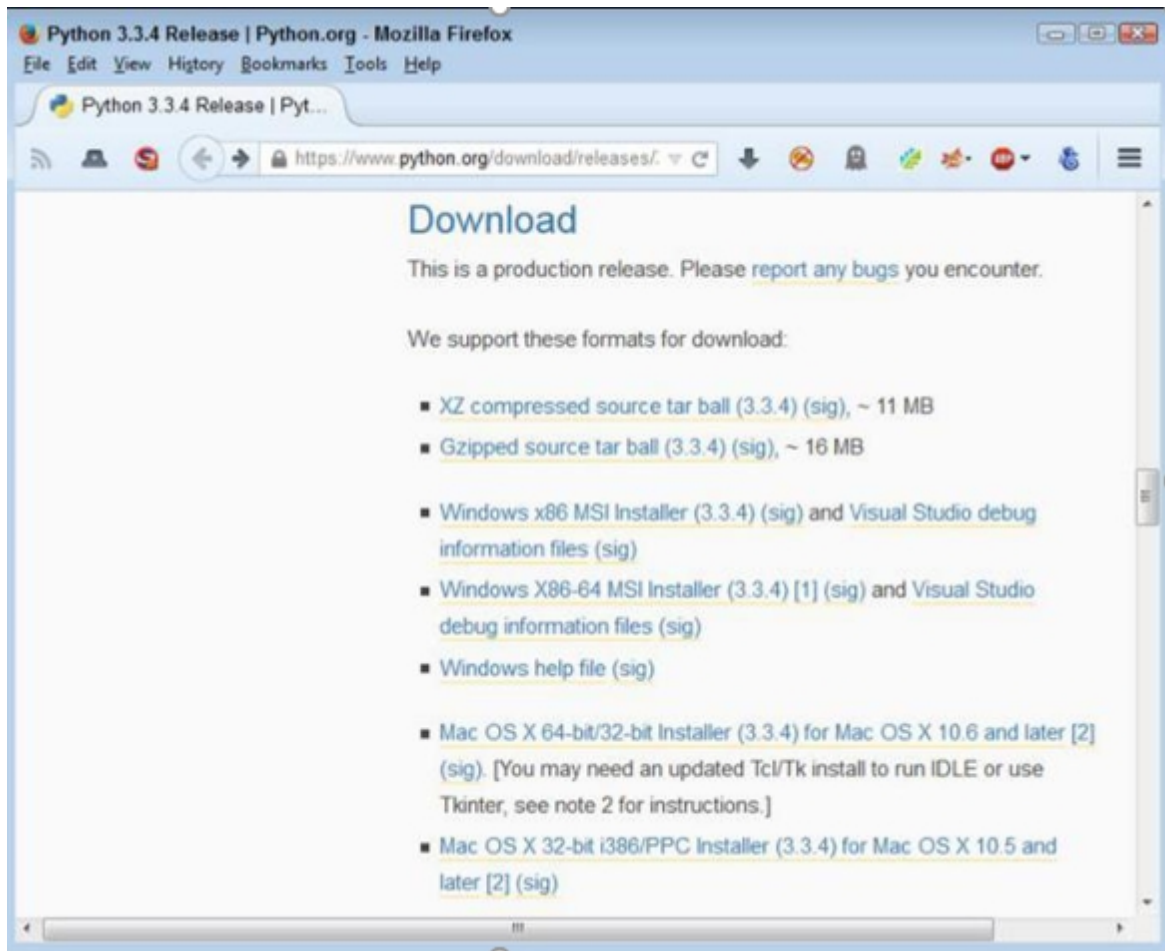
Each platform (a combination of hardware and operating system software) is governed by special rules when running applications. The Python application hides these details. You enter the code that runs on any platform supported by Python, and Python applications translate that code into something that the platform can understand. However, for the translation to

take place, you must have a version of Python that works on your specific platform. Python supports these platforms:

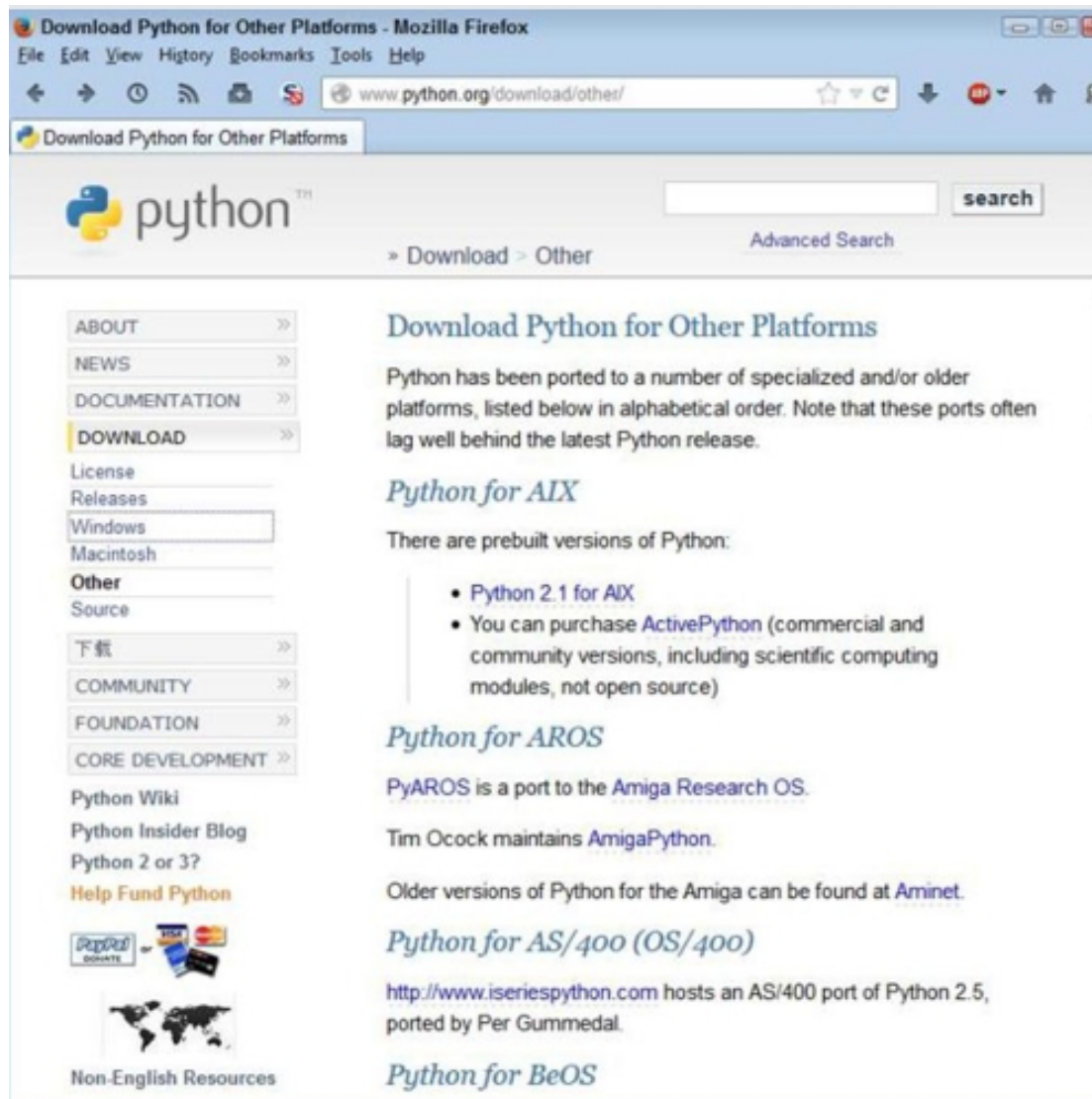
- Amiga Research OS (AROS)
 - IBM Advanced Unix (AIX)
 - 400 Application System (AS / 400)
 - Hewlett-Packard Unix (HP-UX)
 - BeOS
 - Linux
-
- Microsoft Disk operating system (MS-DOS)
-
- Mac OS X (pre-installed with the operating system)
 - MorphOS
 - OS 390 (OS / 390) and z / OS
 - Operating system 2 (OS / 2)
 - PalmOS
 - Psion
 - playground
 - QNX
 - series 60
 - Windows CE / Pocket PC
 - RISC OS (originally Acorn)
 - Solaris
 - 32-bit Windows (XP and later)
 - Virtual memory system (VMS)
 - 64-bit Windows

Wow, that's a lot of different platforms! This book has been tested with Windows, Mac OS X, and Linux platforms. However, the examples can also work with these other platforms because they do not depend on any code specific to that platform.

To get the right version for your platform, you need to access [HTTP: // www. python.org/download/releases/3.3.4/](http://www.python.org/download/releases/3.3.4/). Since the download section is initially hidden, you must scroll down the page.



If you want to use another platform, click the second link on the left side of the page. You discover a list of Python installations for other platforms. Many of these installations are run by volunteers and not by people who create Python versions for Windows, Mac OS X, and Linux. Be sure to contact these people when you have questions about the installation, as they know how to help you get a good setup on your platform.



Installing Python

After downloading your copy of Python, it's time to install it on your system.

The downloaded file contains everything you need to get started:

- Python interpreter
- Help files (documentation)
- Command Line Access
- IDLE (Integrated Development Environment) application
- Uninstaller (only on platforms that need it)

Working with Windows

The process of installation on a Windows system follows the same procedure that is used for other types of applications. The main difference is finding the file you downloaded so that you can start the installation process. The following procedure should work correctly on any Windows system, whether you use the 32-bit or 64-bit version of Python.

Find the copy of Python you have downloaded on your system.

The filename varies, but it usually appears under the following names: python-3.3.4.amd64.msi for 64-bit systems and python-3.3.4.msi for 32-bit systems. The version number is inserted in the file name. In this case, the file name refers to version 3.3.4, which is the version used for this book.

Double-click on the installation file.

(You can see an Open File - Security Warning dialog box asking if you want to run this file.) Click Run if this dialog box appears.) A Python Setup dialog box similar to that shown in Figure 2 is displayed. 3 The precise dialog box you see depends on the version of the Python installer you download.

Choose a user installation option, and then click Next.

The installation prompts you for the name of an installation directory for Python. Using the default destination will save you effort and time later. However, you can install Python anywhere.

Using the Windows \ Program Files (x86) folder or Program Files is problematic for two reasons. First, the folder name has space, which makes access difficult from the application. Second, the folder usually requires administrator access. You will have to always fight with the Windows User Account Control (UAC) feature if you install Python in any folder.

Type a destination folder name, if needed, and then click Next. Python asks you to customize your installation.

Enabling the Add python.exe option to the path will save you time. This feature allows you to access Python from the command prompt window. Do not worry too much about how you use this feature at the moment, but it's a good feature to install. The book assumes that you have enabled this feature. Do not worry about the other features you see in Figure 2-5. They are all enabled by default, giving you maximum access to Python features.

(Optional) Select on the down arrow next to the Add python.exe to path option and choose the options and will be installed on the local drive.

Click Next.

You see the installation process begins. A User Account Control box may appear asking you if you want to perform the installation. If you notice this dialog box, click Yes. The installer continues, and a Setup Complete dialog box appears.

Click Finish.

Python is ready to use.

Working with Mac

Python is perhaps already installed on your Mac system. However, this installation usually takes a few years, regardless of the age of your system. For this book, the installation will probably work properly. You will not test the limits of Python programming technology - you will learn how to use Python.

The latest version of OS X at the time of this publication (Mavericks, or 10.9) comes with Python 2.7, which is very useful for working with book examples.

Depending on how you use Python, you may want to update your installation at some point. Part of this process involves installing the GCC (GNU Compiler Collection) tools so that Python has access to the low-level resources you need. The following steps begin installing a new version of Python on the Mac OS X system.

Click on the link for your version of OS X:

- Python 3.3 Mac OS X 32-bit i386 / PPC installation program for 32-bit versions on the Power PC processor
- Python 3.3 Mac OS X 32-bit / 64-bit x86-64 / i386 installation program for 32-bit or 64-bit versions on Intel

The Python disk image starts to download. Be patient: downloading the disk image takes several minutes. You can easily know how long the download will take because most browsers provide a method to monitor the download process. Once the download is complete, Mac will automatically open the disk image for you.

The disk image looks like a folder. In this folder, you see several files like python.mpkg. The python.mpkg file contains the Python application. Text files contain information about the latest compilation, licenses, and annotations.

Double-click on python.mpkg.

You see a welcome dialog that informs you about this particular Python build.

Click Continue three times.

The installer displays the latest notes on Python, the license information (click Accept when asked about license information), and finally a target dialog box.

Select the volume (hard disk or other media) that you want to use to install Python, and then click Continue.

The Installation Type dialog box appears. This dialog box performs two tasks:

Click Customize to change the feature set installed on your system.

Click Change Installation Location to change the location where the installer places Python.

The book assumes you are performing a default installation and have not changed the installation location. Still, you can make use of these options in case you want to use them.

Click Install.

The installer can request your administrator password. Enter the administrator username and password, if necessary, in the dialog box and click OK. You see a Python Installation dialog box. The contents of this dialog will change as the installation process progresses. This will tell you which part of Python works with the installer.

When you have installed the software successfully, you will see a Successful Setup dialog box.

Click Close.

Python software is ready to use. (You may decide to close the disk image at this junction and delete it from your system.)

Working with Linux

Python software some with some versions of Linux. For example, if you have an RPM (Red Hat Package Manager) -based distribution (such as CentOS, SUSE, Yellow Dog, Red Hat, and Fedora Core), you probably already have Python on your system, and there is absolutely nothing else for you to do.

Depending on the version of Linux that you use, the version of Python varies, and some systems do not include the Interactive Development Environment (IDE) application. If you own an earlier version of Python (version 2.5.1 or earlier), you may want to install a newer version to access IDLE. Most book exercises require the use of IDLE.

Using the default Linux installation

The default installation of Linux runs on any system. However, you must work in the terminal and enter the commands to complete it. Some of the actual commands may vary depending on the version of Linux. The information on <http://docs.python.org/3/install/> provides useful tips that you can use in addition to the following procedure.

Click on the link that matches your Linux version:

Compressed source archive Python 3.3.4 (any version of Linux)

Python 3.3.3 xzip python fonts (better compression and faster download)

You will be prompt to either open or save the file, choose Save.

Python source files are being downloaded. Be patient: downloading source files take a minute or two.

Double-click on the downloaded file.

The Archive Manager window opens. Once the files are extracted, you see the Python 3.3.4 folder in the file manager window.

Double-click the Python 3.3.4 folder.

The file manager extracts the files from the Python 3.3.4 subfolder from your folder.

Open a copy of the terminal.

The terminal window is displayed. If you have never created software on your system before, you must install the basics of the compilation, SQLite

and bzip2. Otherwise, the installation of Python will fail. Otherwise, you can go to step 10 to start using Python without any delay.

Press Enter after typing the following "sudo apt-get install build-essential."

Linux installs the necessary Build Essential support for creating packages (see <https://packages.debian.org/squeeze/build-essential> for more details).

Press Enter after typing the following "sudo apt-get install libsqlite3-dev."

The SQLite support needed by Python software for database manipulation is installed by Linux (see <https://packages.debian.org/squeeze/libsqlite3-dev>).

Press Enter after typing the following "sudo apt-get install libbz2-dev."

The bzip2 support required by Python software for file manipulation is installed by Linux (see <https://packages.debian.org/sid/libbz2-dev> for more details).

Type CD Python 3.3.5 in the Terminal window and press Enter. The terminal changes directories in the Python 3.3.5 folder of your system.

Type ./configure and press Enter.

The script starts by checking the type of system build, then performs a series of tasks depending on the system you are using. This process can take few minutes because there is a long list of things to check.

Type make and press Enter.

Linux runs the creation script to create the Python application software. The manufacturing process may take a minute - this depends on the processing rate of your system.

Type sudo make altinstall and press Enter.

The system may prompt you for your administrator password. Enter your password and press Enter. At this point, several tasks occur when the system installs Python on your system.

ACCESS PYTHON ON YOUR MACHINE

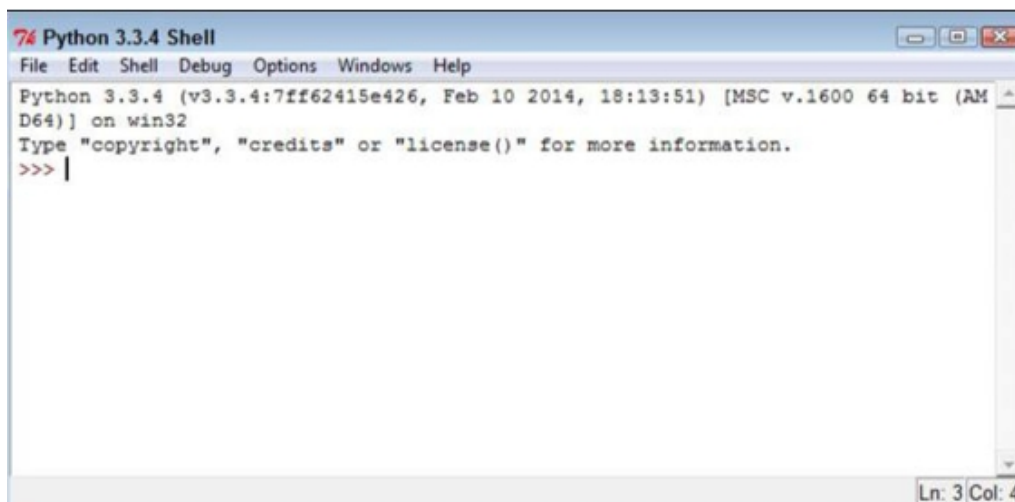
After installing Python on your system, you need to know where to find it. In a way, Python makes every effort to facilitate this process by performing certain tasks, such as adding the Python path to machine path information

during installation. Even then, you need to know how to access the installation described in the following sections.

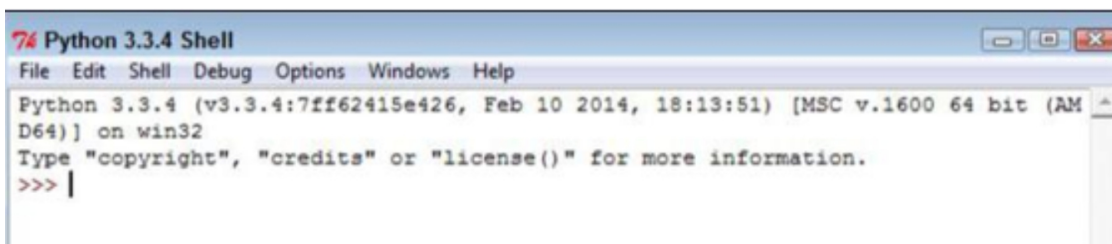
Using Windows

A Windows installation creates a new folder on the Start menu containing your Python installation. You can launch it by navigating through Start> All Programs> Python 3.3.5 The two items of interest in the folder when building new applications are Python (command line) and IDLE (Python GUI).

Clicking on IDLE (Python GUI) generates an interactive graphical environment When you open this environment, IDLE automatically displays certain information to make sure you have the right application open. For example, you see the version number of Python (which is 3.3.4 in this case). It also tells you what kind of system you are using to run Python.



The Python (command line) option will open a command prompt and executes the Python command. Again, the environment automatically shows information such as the host platform and the Python version.



A third way to access Python is to open a command prompt, enter Python, and press Enter. You can adopt this approach when you want to gain

additional flexibility over the Python environment, load items automatically, or run Python in an environment with higher privileges (in which you get additional security rights). Python provides a large set of command-line options that you can see by typing `Python /?` at the command prompt and press Enter. Do not worry too much about these command-line options. You will not need it for this book, but it is useful to know that they exist.



```
Administrator: Command Prompt
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Windows\system32>Python /?
usage: Python [option] ... [-c cmd | -m mod | file | -l [arg] ...
Options and arguments (and corresponding environment variables):
-b      : issue warnings about str(bytes_instance), str(bytearray_instance)
         and comparing bytes/bytearray with str. (-bb: issue errors)
-B      : don't write .pyc files on import; also PYTHONDONTWRITEBYTECODE=x
-c cmd  : program passed in as string (terminates option list)
-d      : debug output from parser; also PYTHONDEBUG=x
-E      : ignore PYTHON* environment variables (such as PYTHONPATH)
-h      : print this help message and exit (also --help)
-i      : inspect interactively after running script; forces a prompt even
         if stdin does not appear to be a terminal; also PYTHONINSPECT=x
-m mod  : run library module as a script (terminates option list)
-O      : optimize generated bytecode slightly; also PYTHONOPTIMIZE=x
-OO     : remove doc-strings in addition to the -O optimizations
-q      : don't print version and copyright messages on interactive startup
-s      : don't add user site directory to sys.path; also PYTHONNOUSERSITE
-S      : don't imply 'import site' on initialization
-u      : unbuffered binary stdout and stderr, stdin always buffered;
         also PYTHONUNBUFFERED=x
         see man page for details on internal buffering relating to '-u'
-v      : verbose (trace import statements); also PYTHONVERBOSE=x
         can be supplied multiple times to increase verbosity
-U      : print the Python version number and exit (also --version)
-W arg  : warning control; arg is action:message:category:module:lineno
         also PYTHONWARNINGS=arg
-x      : skip first line of source, allowing use of non-Unix forms of #!cmd
-X opt  : set implementation-specific option
file    : program read from script file
-       : program read from stdin (default; interactive mode if a tty)
arg ... : arguments passed to program in sys.argv[1:]

Other environment variables:
PYTHONSTARTUP: file executed on interactive startup (no default)
PYTHONPATH   : ';' separated list of directories prefixed to the
               default module search path. The result is sys.path.
PYTHONHOME   : alternate <prefix> directory (or <prefix>;<exec_prefix>).
               The default module search path uses <prefix>\lib.
PYTHONCASEOK : ignore case in 'import' statements (Windows).
PYTHONIOENCODING: Encoding[:errors] used for stdin/stdout/stderr.
PYTHONFAULTHANDLER: dump the Python traceback on fatal errors.
PYTHONHASHSEED: if this variable is set to 'random', a random value is used
               to seed the hashes of str, bytes and datetime objects. It can also be
               set to an integer in the range [0,4294967295] to get hash values with a
               predictable seed.

C:\Windows\system32>_
```

To use this third method of running Python, you must include Python in the Windows path. That's why you want to choose the Add python.exe to path option when installing Python on Windows.

This same technology allows you to add environment variables specific to Python, such as

- PYTHONSTARTUP
- PYTHONPATH
- PYTHONHOME

- PYTHONCASEOK
- PYTHONIOENCODING
- PYTHONFAULTHANDLER
- PYTHONHASHSEED

None of these environment variables are used in the book. However, you can read more about them at <http://docs.python.org/3.3/using/cmdline.html#variables-environment>.

Using Mac

When working with a Mac, you probably already have Python installed and do not need to install it for this book. However, you must always know where to find Python. The following sections explain how to access Python, depending on the type of installation you perform.

Find the default installation

The default installation of OS X does not include a specific Python folder in most cases. Instead, you must open Terminal by selecting Applications → Utilities Terminal. As soon as the terminal is open, you can type Python and press Enter to access the Python command line version. As with Windows, using Terminal to open Python has the advantage of using command line options to change how Python works.

Locating The Updated Version Of Python That You Have Installed

After running the installation on the Mac system, open the Applications folder. In this folder, you will find a Python 3.3 folder containing the following:

Pasta extras

IDLE application (GUI development)

Python Launcher (development of interactive commands)

Update Sh ...

Double-click the IDLE application to open an interactive graphical environment. There are several minor cosmetic differences, but the contents of the window are identical. Double-click the Python launcher to open a command-line environment. This environment uses all Python defaults to provide a default runtime environment.

Even if you install the latest version of Python on your Mac, you do not have to use the default environment. You can still open Terminal to access Python command-line switches. However, when you launch Python from the Mac Terminal application, you must ensure that you do not access the default installation. Don't forget to add /usr/local/bin/Python3.3.5 to your shell search path.

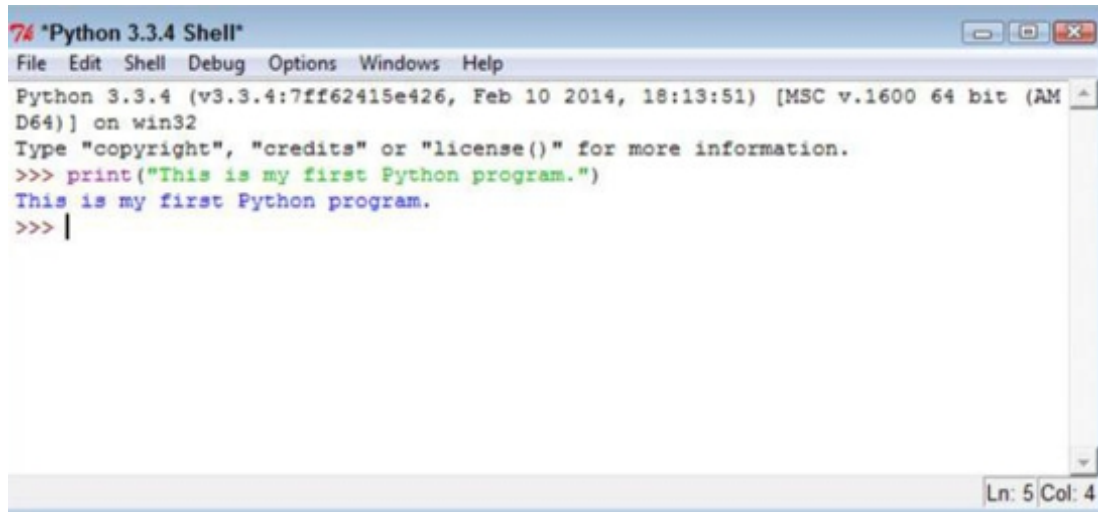
Using Linux

After you have successfully installed Python, you can find a Python 3.3 subfolder in your folder. The directory of Python 3.3 on your Linux system is usually the /usr/local/bin/Python3.3 folder. This information is vital because you may need to change the path to your system manually. Linux developers must type Python3.3, not just Python when working in the Terminal window to access the Python 3.3.4 installation.

Test your installation

To make sure your installation is usable, you must test it. It's essential to know that your installation will work as expected when you need it. Of course, that means writing your first application in Python. To begin, open a copy of IDLE. As mentioned earlier, IDLE automatically displays the Python version and host information when you open them.

To see the work in Python, type `print ("This is my first program in Python")` and press Enter. Python reveals the message you just typed as shown below. The `print ()` command displays on the screen everything you say to display. The `print ()` command used in this book quite often displays the results of the tasks you request from Python. This is among the commands you work with often.

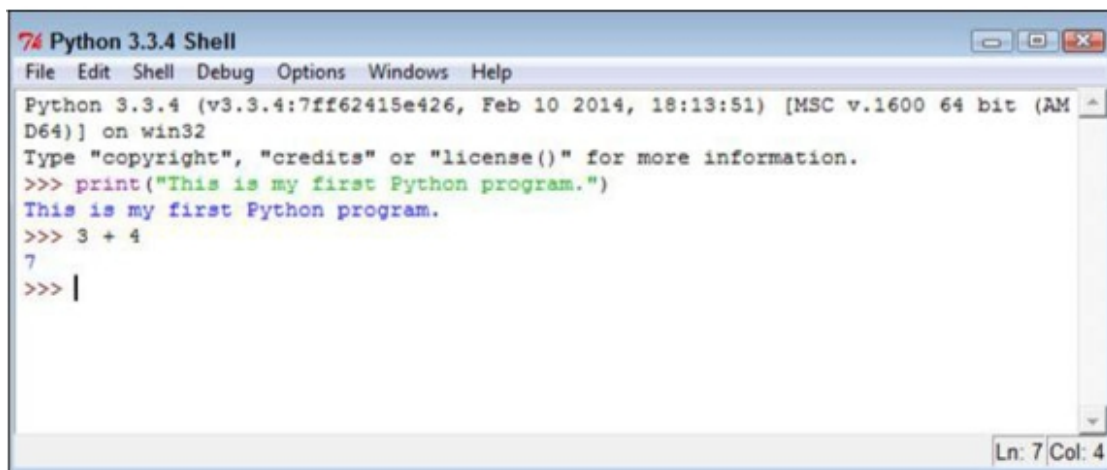


```
Python 3.3.4 Shell
File Edit Shell Debug Options Windows Help
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("This is my first Python program.")
This is my first Python program.
>>> |
```

Note that the IDLE color encodes the different entries so you can see and understand them more easily. The color codes indicate that you have done well. Four color codes are shown above (although they are not visible in the printed edition of the book):

- Green: Defines the content sent to a command.
- Purple: shows that you have typed a command
- Blue: Displays the result of a command.
- Black: Define non-command entries

You know that Python is working now because you have been able to give it a command and have reacted by reacting to that command. It might be interesting to see an additional order. Enter $3 + 4$ and press Enter. Python responds by issuing 7. Note that $3 + 4$ appears in the black type because it is not a command. However, the seven is still in the blue type because it is produced.



```
Python 3.3.4 Shell
File Edit Shell Debug Options Windows Help
Python 3.3.4 (v3.3.4:7ff62415e426, Feb 10 2014, 18:13:51) [MSC v.1600 64 bit (AMD64)] on win32
Type "copyright", "credits" or "license()" for more information.
>>> print("This is my first Python program.")
This is my first Python program.
>>> 3 + 4
7
>>> |
```

Now you need to end your IDLE session: type `quit ()` and press Enter. IDLE can display a message. Well, you don't have the intention to kill anything, but you will do it now. Click OK, and the session will be completed.

Note that the `quit ()` command has parentheses later, just like the `print ()` command. All commands have parentheses like these. That's how you know its orders. However, you have nothing to say to the command `quit ()`, so leave the area between the white spaces of the theses.

CHAPTER THREE: UNDERSTANDING THE BASICS AND WRITING YOUR FIRST PROGRAM

Now is the time to learn a little more about Python programming and how to make it work for you. You need to know a bit more about the different keywords and variables that come with Python, in order to write the word you want and make the program work in a certain way. Let's look at some of these basics of Python programming so you can get started immediately with your new code.

KEYWORDS

When you just started working on a new computer coding program, you will notice that each language on the computer will have certain keywords. These are the words intended for a specific command or purpose in the language and you should avoid using them elsewhere. If you use these words in other parts of your code, you may receive an error warning, or the program will not function correctly. Python-only keywords include:

- Pass
- And
- Or not
- None
- Nonlocal
- Lambda
- In import
- Is
- If
- From
- Global
- For
- False
- Finally

- Except
- Elif
- Else
- Del
- Continue
- Def
- Class
- Assert
- Break
- As
- With
- Yield
- While
- Raise
- True
- Try
- Return

IDENTIFIER NAMES

When creating a new Python program, you will be working on creating certain entities, a combination of functions, classes, and variables. Everyone or f will have a name also known as an id. There are a few regulations you should follow when forming an identifier in Python, including:

- It must contain upper or lower case letters or a combination of two, numbers and an underscore. You shouldn't see any spaces inside.
- The identifier cannot start with a number
- The identifier cannot be a keyword and must not include any.

If you refuse to follow any of these regulations, the program will close and display a syntax error. In addition, work is needed to create human-readable identifiers. Although the identifier can make sense to the computer and be overcome without causing problems on the computer, it is the human being who reads the code to use it alone. If the human eye does not understand what you are writing in a particular place, you may

have problems. Some rules that we must follow when creating a human-readable identifier include:

- The identifier must be descriptive: you must choose the name that will describe what is inside the variable or what it does.
- Beware of unnecessary abbreviations, as they always make things difficult.

While there are multiple ways to write your code, you need to be careful and follow a rule. For example, `MyBestFriend` and `mybestfriend` work in the programming world, but choose the one you like and do the same every time you work on the program to avoid confusion. You can also add underlined characters to that or numbers, just be sure to keep things consistent.

FLOW OF CONTROL

When working in the Python language, you write instructions in a list format, just like you would when writing a shopping list. Your computer will start with the first few instructions before examining each one in the order in which they appear in the list. You will then be asked to perform the desired checks in your shopping list, to ensure that the computer is reading correctly. Your computer stops reading this list only after completing the final instructions. This is called the control flow.

This is an important way to start. You want to ensure that your flow of control is even and smooth for computer playback. With this, it is easy for the program to do what you want without much hassle and ensure that the computer program doesn't fail, cause problems, or something goes wrong.

SEMI-COLONS AND INDENTATION

By examining some of the other languages on your computer, you will notice that there are many keys used to organize the various blocks of code or to start and end the statements. This allows you to remember to indent the code blocks in these languages to make the coder to read, even though the computer can read the various codes without the indentations just fine.

This type of coding can be difficult to read. You will see some unnecessary information that the computer needs to read the code, but it can be difficult to read. Python makes use of a different way of doing this, mainly to help the human eye read what you have. You will need to identify the code for this to work. An example of this is:

```
# this function definition starts a new block
def add_numbers (a, b):
    c = a + b
    # such as
    return c
# this function definition starts a new block
if it is Sunday
    print (It's Monday!)
# and this one is not inside the block
    print ("Print this no matter what.")
```

Additionally, many languages use a semicolon to know when an instruction ends. However, with Python, you will use the line ends to notify the computer when the instructions are complete. You can make use of a semicolon if you have instructions on the same line, but this is generally considered an incorrect form in the language.

LETTER CASE

Most computer languages will treat upper and lower case letters the same, but Python is one of the few that distinguishes between upper and lower case. This means that lowercase and uppercase letters will be treated differently in the system. Also remember that all reserved words will use lowercase letters except None, False, and True.

These fundamentals will make it easier to get started with Python programming. You need some time to go through the program to familiarize yourself with it. It is not compulsory to become an expert, but familiarizing yourself with some of the text interpreters and some of the other areas of the program can make it easier to use and learn how the

different buttons will work before you even start. Try some of the examples above first to get you started.

Python strives to keep things as basic as possible, as it understands that the majority of its users will be beginners or that they are tired of other complex languages. As we have seen here, and in the following chapters, there are some simple commands you can bring forward to make the program work in a specific way. Study them and you can create a great program without too much work.

WRITING OUR FIRST PROGRAM

We will write our code using the IDLE program provided with our Python interpreter.

To do this, we first start the IDLE program. You start the IDLE program just like you start any other program. For example, in Windows 10, you can search for it by typing "IDLE" in the search box. Once found, click on IDLE (Python GUI) to launch it. You will be introduced with Python Shell

Python Shell let us make use of Python in interactive mode. This means that we can only enter one order at a time. The Shell waits for the user's command, executes it and returns the result of the execution. After that, Shell waits for the next command.

Type the following into the Shell. The lines that starts with `>>>` are the commands you must type while the lines after the commands show the results.

```
>>> 5 + 4
9
>>> 5 > 4
True
>>> print('Hello World')
Hello World
```

When you type `5 + 4`, you send a command to Shell, asking it to evaluate the value of `5 + 4`. Then the Shell returns the response `9`. When you type `5 >`

4, you ask Shell if 5 is greater to 4, the Shell answers true. Finally, print is a command that requires the Shell to display the Hello World line.

Python Shell is a very handy tool for testing Python commands, especially when we are new to the language. However, if you quit Python Shell and reenter it, all commands you entered will disappear. Additionally, Python Shell cannot be used to create a real program. To code a real program, you must write the code to a text file and save the file with a .py extension. This file is referred to as a Python script.

To create a Python script, click File> New File from the top menu of our Python shell. This will show up the text editor that we will use to write our first program, the "Hello World" program. Writing the "Hello World" program is like a rite of passage for all new programmers. We will use this program to familiarize ourselves with the IDLE software.

Enter the below code in the text editor (not in the Shell).

```
#Prints the Words "Hello World"
print ("Hello World")
```

Note that the line #Prints the words "Hello World" in red, while the word "print" is in purple and "Hello World" in green. This is how the software makes our code easier to read. The words "print" and "Hello World" have different purposes in our program, so they are displayed in different colors. We will go talk more about this in the following chapters.

The line #Print the words "Hello World" (in red) is not part of the program. This is a comment we write to make our code more readable for other users. This line is always ignored by the Python interpreter. To add comments to our calendar, we type a # sign in front of each comment line, like this:

```
# This is a comment
# This is also a comment
# This is another comment
```

On the contrary, we can additionally make use of three single quotes (or three double quotes) for multiline comments, like this:

```
'''
This is a comment
```

```
This is also a comment
This is yet another comment
'''
```

Now click File> Save As ... to save the code. Save it with the .py extension.

Finished? Over there! You have effectively written your first Python program.

Finally, click Run> Run Module to run the program (or press F5). The words Hello World should be shown in your Python shell.

However, remember he used Python 2 in the video; therefore, some commands will cause an error. If you wish to test your codes, you need to add () to the printing instructions. Instead of writing "Hello World", you should write `print ("Hello World")` . Additionally, you need to change `raw_input()` to `input()` . We'll cover `print ()` and `input ()` in the next chapter of this book.

A BIT MORE ON COMMENTS

With python, you can do many great things. The comment is one of the most interactive options that you will find when you start programming and is very easy to use. In this section, it will take you some time to discuss more about comments and some other aspects of Python, so that you can get started and make your code amazing quickly.

In Python programming, a comment begins with the # sign and continues to the end of the line. For example:

```
# This would be a comment
print (Hi, are you okay?)
```

This told the computer to print "Hi, are you okay?" All comments are not executed in the Python interpreter as it is more of a footnote in the program to assist the programmer, or others who may be using the code, special things about the code. Basically, I'm here to tell you what the program should do and how it will work. It's a bit more detailed and can be useful without interfering with the functioning of the code.

It is not compulsory to leave a comment on each line, only when necessary. If the programmer feels that something should be explained better, he inserts a comment, but doesn't expect to see it everywhere. Python does not support comments that span multiple lines; therefore, if you have a longer comment on the program, find out how to split it into multiple lines with the # sign in front of each part.

Writing and Reading

Some programs display the desired text on the screen or may require certain information. You can start the program code by telling the reader the subject of your program. Giving a name or title can make things easier, so that the other programmer knows what's in the program and can choose the right path for them.

The best way to display the correct information is to display a literal string that will include the "print" function. For the uninitiated, sequence literals are basically lines of text that will be enclosed in quotes, single, and double. It doesn't really matter what type of quote you use, but if you use a type at the beginning of the sentence, you should use it at the end. Therefore, if there are double quotes at the beginning of the sentence, be sure to follow the double quotes at the end.

When you want the computer to print a word or phrase on the screen, all you need to do is "print" and then the next phrase. For example, if you want to represent "Welcome!" Would you do it

```
Print("Welcome!")
```

This will cause "Welcome" to appear in your program for others to use. The print function will have its line, so you will notice that after inserting it, the code will automatically put it on a new line.

If you want the visitor to take a certain action, you can follow the same type of idea. For example, suppose you want the person to enter a specific number so that they can step through the code that you would use in the string:

```
first_number = input('insert the first number')
```

When using the input function, it will not automatically display on a new line. The text will be inserted closely after the prompt. You will also be required to convert the string to a number for the program to work. It is also

not a must to have a specific parameter for this. If you make the following choice with just the parentheses and nothing inside, you will get the same result, and sometimes you will make it easier.

Files

Most of the time, you will use the print function to get a sequence of characters to print on the screen. This is the default for the print function, but you can also use this same function as a good way to write something to a file. A good example of this is

```
With open('myfile.txt', 'w') as myfile: Print("Hello!",file=myfile)
```

Now, this might seem like a simple equation, but in the above sequence there are a lot of things happening that you need to be careful about. Where you opened myfile.txt to save and then assigned to the variable called myfile. So in part two you wrote about Hello! For the file as a new line, we then inform the program that the changes will not be saved until the file is opened.

Obviously, you don't need to use the print function to complete the job. The writing method will also work well. For example, you can replace printing with saving, as in the following example, to achieve the same results.

```
With open('newfile.txt', 'w') as newfile: newfile.write("Hello!")
```

Right now, we have learned how to print a sequence of words in the program and even save it to a specific file. In addition to these choices, you can use the read method to open a specific file and then read the data there. If you want to open and play a specific file, use this option:

```
With open ('myfile.txt', 'r') as myfile:  
data = myfile.read ()
```

With this option, you can ask the program to read the contents of the files in variable data. This can make it easier to open the programs you want to read.

BUILT-IN TYPES

Your computer can process a lot of information, including numbers and characters. The types of information that will be used by the Python program are called types and the programming language will contain many different types to make it easier. Some of them include strings, integers, and

floating point numbers. Programmers have the ability to even define these different types using classes.

The types will consist of two distinct parts. The first part is a domain which will contain a possible set of values and the second part is a set which contains possible operations. Both can be executed at any value. An instance of this is when a domain that is an integer type, it can only contain whole numbers that include addition, division, subtraction, and multiplication.

One thing you should never forget about python is that Python is a dynamic type program. This implies that it is not a must to specify types to variables when creating them. The same variables may be used to store values of different types, the Python program would identify this and display an error. It won't make an attempt to figure out what you wanted; on the contrary, it will exit without trying.

Integers

If you want to use whole numbers as a type, keep them as whole numbers. They can be negative or positive numbers, as long as there are no decimal places with these figures. If there is a decimal point within the digit, even though the number is 3.0, you will then need to use it as a floating point number.

Python has the ability to display these integers in the "print" function, but only if it is the only argument.

```
Print(5)
```

```
# Let's add two numbers together
```

```
Print (4+1)
```

If you are using whole numbers, you cannot place the two next to each other. This is mainly due to how Python is a strongly typed language and you won't recognize them if you combine them. If you want to join the number and the sequence, check if the number has become a sequence.

Operator Precedence

One thing you must keep track of when working in Python is operator precedence. For example, if you have $2 + 3 // 5$, Python might interpret it as $(2 + 3) // 5$ or $2 + (3 // 5)$. Python has a way that will help you classify the

operation correctly, so that you get the right information. For example, when it comes to integer operations, Python handles parentheses first. Then it'll take care of the things that have **, then *, and then //, then %, +, and finally -.

If you are writing an expression that contains a series of operations, you should keep these signs in mind. This will tell Python how to examine the numbers so you can get the correct answers now. Remember that most arithmetic operators will remain associative; therefore, write like this so that Python can read it. The only exception is the ** function. For example:

```
# ** is right-associative 3**4**5  
# will be evaluated right to left:  
3**(4**5)
```

Strings

Although a string may seem complicated, in Python it is essentially a sequence of characters. They will work the same as a list, but contain a bit more specific text functionality.

Formatting strings can be difficult when it comes to writing your own code. Some messages will not be corrected as a string and sometimes there are values stored in variables. There is a particular way to get this to work right for string formatting. An example of this is:

```
Name = "Aristotle" Age = 30  
Print("Hi! My name is %s." % name)  
Print("Hi! My name is %s and I am %d years of age." % (name,  
age))
```

The symbols that have a % first are known as placeholders. The variables that go into these placeholders will be placed after the % in the order where they are arranged in the string. If you are working with just a single string, you will not require a wrapper, but if you have more than one you have to put them in a tuple with a () enclosing it. The marker symbols will start with different letters, depending mainly on the type of variable you are using. For example, age will be an integer by name is a string. All of these variables be converted into the string before we can add them into the rest.

Escape sequences

Escape sequences can be used to indicate special characters that may be difficult to type on the keyboard. Additionally, they can be used to indicate characters that can be reserved for something else. For instance, using `\n` in the sequence can complicate the program so you may use the escape sequence in place of that like the following example:

Print ("This is a single line. \nThis is another line.")

Triple quotes

We've spent some time talking about single and double quotes, but there are times when you will need to type the triple quotes. It is used when you need to define a literal value that spans multiple lines or a value that already contains many quotes. To try this, just make use of a single and a double set or three singles. The same rule applies to the triple quote and all others. You will have to start and end the sentence with it.

String operations

One of the string operations that you can use a lot is concatenation. This is used to join a pair of strings and you'll notice it's there with the `+` symbol. Python can help you with many functions and will work with strings to create a variety of operations. They will have some useful options that can do a lot more in the Python's program

In the Python program, strings are called immutable. This implies that once the string is created, it cannot be edited. It may be necessary to assign a new value to a specific existing variable, if you want to make changes.

You can know a lot of things to get familiar with Python. It might be plain language, but you want to learn how it works, how to write things correctly, and even how to leave a comment that other people can understand when they review the code. At first it might appear a little intimidating, but in no time and with a little practice, you'll be able to write it and write your code quickly.

CHAPTER FOUR: THE WORLD OF VARIABLES AND OPERATORS

Now that we're through with the introductory stuff, it's time to get down to the real stuff. Here in this section, we will talk about all about variables and operators.

In particular, you will learn what variables are and how you can name and declare them. Also, we will learn about some common operations that can be performed on them. Are you ready? Here we go.

WHAT ARE VARIABLES?

Variables are names given to the data we need to store and manipulate in our programs. For example, let's say your program needs to remember a user's age. To do this, we can name this data `userAge` and set the variable `userAge` using the following statement.

```
userAge = 0
```

After defining the `userAge` variable, the program will allocate a certain area of the computer's storage space to store this data. Therefore, it is possible to access and modify this data by consulting it with its name, `userAge`. Each time you declare a new variable, you must give it an initial value. In this example, we give it a value of 0. We can always alter this value in our program later.

We can also define several variables simultaneously. To do this, simply write

```
userAge, userName = 40, 'Peter'
```

This is equivalent to

```
userAge = 40
```

```
userName = 'Peter'
```

NAMING A VARIABLE

A variable name in Python can only contain letters (a - z, A - B), numbers, or underscores (_). However, the first character cannot be a number.

Therefore, you can name your variable `userName`, `user_name` or `username2`, but not `2userName` .

Additionally, there are reserved words that you cannot use as a variable name because they already have pre-assigned meanings in Python. These reserved words include words like `print`, `input`, `if`, `while`, etc. We will learn about all of them in the next chapters.

Lastly, variable names are case sensitive. `Username` is different from `userName` .

There are two conventions for naming a variable in Python. We can use the underscores or use camel case notation. The camel case is the practice of writing compound words with a mixed casing (for example `thisIsAVariableName`). This convention that will be used in the rest of the book. However, another common practice is to use underscores (_) to separate words. If you like you can also name the variables as follows: `this_is_a_variable_name` .

THE ASSIGNMENT SIGN

Note that the `=` sign in the `userAge = 0` statement has a different meaning than the `=` sign we learned in math. In programming, the `=` sign is known as the assignment sign. This means that we are assigning the value on the right side of the `=` sign to the variable on the left. A good way to understand the `userAge = 0` statement is to think of it as `userAge <- 0` .

The instructions `x = y = x` has very different meanings in programming.

Confused? An example will probably clarify this.

Enter the following code into your IDLE editor and save it.

```
x = 15
y = 20
x = y
print ("x =", x)
```

```
print ("y =", y)
```

Now run the program. You should get this result:

```
x = 20
```

```
y = 20
```

Although x has an initial value of 15 (declared in the first row), the third-row `x = y` gives the value of y to x (`x <- y`), thus changing the value of x to 20 while the value of y remains unchanged.

Now modify the program by changing ONLY ONE statement: change the third line of `x = y` to `y = x`. Mathematically, `x = y` and `y = x` means the same thing. However, this is not the case in programming.

Run the second program. Now you will have

```
x = 15
```

```
y = 15
```

We see that in this example the value x remains at 15, but the value of y is changed to 15. This is because the instruction `y = x` affects the value of x to y (`y <- x`). y becomes 5 while x remains unchanged at 5.

BASIC OPERATORS

Operators are the symbols that tell the interpreter to perform a specific operation, such as arithmetic, comparison, logic, etc.

The different types of Python operators are listed below:

- Relational operators
- Arithmetic operators
- Bitwise operators
- Logical operators
- Assignment operators
- Association operators
- Identity operators

Relational operators

A relational operator in python is used to compare two operands in order to decide on a relationship between them. Returns a Boolean value (false or

true) depending on the condition.

- > Returns True when the right operand is less than the left operand
- < Returns True when the right operand is greater than the left operand
- == Returns True if both operands are equal
- >= Returns True when the right operand is less than or equal to the left operand
- <= Returns True when the left operand is less than or equal to the right operand
- != Returns True if both right and left operands are not equal

Arithmetic operators

An arithmetic operator accepts two operands as input, carry out a calculation, and gives the result.

Consider the expression "b = 4 + 5". Here, 4 and 5 are the operands and + is the arithmetic operator. The output of the operation is stored in the variable b.

- + Performs Addition on the operands
- - Performs Subtraction on the operands
- * Performs Multiplication on the operands
- / Performs Division on the operands
- % Performs a Modulus on the operands
- ** Performs an Exponentiation operation
- // Performs a Floor Division operation

Note: To get the result in a floating type, one of the operands must also be of type float.

Bitwise operators

A bitwise operator performs operations on bitwise operands.

Consider a = 3 (in binary notation, 10) and b = 4 (in binary notation, 11) for the following uses.

- & Performs bitwise AND operation on the operands
- | Performs bitwise OR operation on the operands
- ^ Performs bitwise XOR operation on the operands

- `~` Performs bitwise NOT operation on the operand. Flips every bit in the operand
- `>>` Performs a bitwise right shift. It shifts the bits of left operand, right by the total number of bits specified as the right operand
- `<<` Performs a bitwise left shift. It shifts the bits of left operand, left by the total number of bits specified as the right operand

Logical operators

A logical operator in python is used to make a decision based on multiple conditions. The logical operators that are used in Python are and, or and not.

- `and` Gives True if both right and left operands are True
- `or` Gives True if any one of the operands are True
- `not` Gives True if the operand is False

Assignment operators

An assignment operator in python is used to assign values to a variable. This is usually combined with other operators (such as arithmetic, bit by bit), in which the operation is carried out on both operands and the output is assigned to the left operand.

Let's consider the following examples,

`a = 20`. Here `=` is an assignment operator and the output is stored in the variable `a`.

`a += 50`. Here `+=` is an assignment operator and the output is stored in the variable `a`. It is equal to `a = a + 10`.

- `=` `a = 20`. The value 20 is assigned to the variable `a`
- `+=` `a += 10` is equivalent to `a = a + 10`
- `-=` `a -= 10` is equivalent to `a = a - 10`
- `*=` `a *= 10` is equivalent to `a = a * 10`
- `/=` `a /= 10` is equivalent to `a = a / 10`
- `**=` `a **= 10` is equivalent to `a = a ** 10`
- `%=` `a %= 10` is equivalent to `a = a % 10`
- `//=` `a //= 10` is equivalent to `a = a // 10`

- `|=` `a |= 10` is equivalent to `a = a | 10`
- `&=` `a &= 10` is equivalent to `a = a & 10`
- `^=` `a ^= 10` is equivalent to `a = a ^ 10`
- `<<=` `a <<= 10` is equivalent to `a = a << 10`
- `>>=` `a >>= 10` is equivalent to `a = a >> 10`

Membership Operators

A membership operator is used to identify the membership in any sequence (strings, lists, tuples).

not in and ***in*** are membership operators in python.

in returns True if the stated value is found in the sequence. Gives False otherwise.

not in returns True if the stated value is not found in the sequence. Gives False otherwise.

```
a = [1,2,3,4,5]

#Is 3 in the list a?
print 3 in a # prints True

#Is 12 not in list a?
print 12 not in a # prints True

str = "Hello World"

#Does the string str contain World?
print "World" in str # prints True

#Does the string str contain world? (note: case sensitive)
print "world" in str # prints False

print "code" not in str # prints True
```

Identity operators

An identity operator in python is used to check if two variables share the same memory location.

is ***not*** and ***is*** are identity operators.

is gives True if the operands refer to the same object. Gives False otherwise.

is not gives True if the operands do not refer to the same object. Gives False otherwise.

Please note that two values when equal, need not imply they are identical.

```
a = 3
b = 3
c = 4
print a is b # prints True
print a is not b # prints False
print a is not c # prints True

x = 1
y = x
z = y
print z is 1 # prints True
print z is x # prints True

str1 = "FreeCodeCamp"
str2 = "FreeCodeCamp"

print str1 is str2 # prints True
print "Code" is str2 # prints False

a = [10,20,30]
b = [10,20,30]

print a is b # prints False (since lists are mutable in Python)
```

CHAPTER FIVE: DATA TYPES AND TYPE CASTING IN PYTHON

Here in this section, we will first take a look into some basic data types in Python, especially the integer, float, string, etc. Next, we will explore the concept of type casting. Finally, we'll cover three other advanced data types in Python: list, tuple, and dictionary.

The first is a about of the basic data types incorporated into Python.

Here's what you'll learn in this chapter:

- You will learn about several **Boolean**, **numeric**, and **string** types that are built in Python. By the end of this tutorial, you will know what objects of these types look like and how to represent them.
- You will also get an overview of **functions** embedded in Python. These are pre-written codes that you can call upon to do useful things. You've seen the built-in `print ()` function, but there are many more.

INTEGERS

In Python 3, there are no limits to the duration of an integer value. Obviously, it's limited by the amount of memory in your system, like all things, but on top of that an integer can be as long as you need it:

[illegible]

123123123123123123123123123123123123123123123125

Python interprets decimal digits sequence without a prefix as a decimal number:

```
>>> print(20)
```

20

The following strings can be appended to an integer value to indicate a base other than 10:

Binary

- 0b (zero + lowercase letter 'b')
- 0B (zero + uppercase letter 'B')

Octal

- 0o (zero + lowercase letter 'o')
- 0O (zero + uppercase letter 'O')

Hexadecimal

- 0x (zero + lowercase letter 'x')
- 0X (zero + uppercase letter 'X')

For example:

```
>>> print(0o10)
8
>>> print(0x10)
16
>>> print(0b10)
2
```

The underlying type of a Python integer, regardless of the base used to specify it, is called int:

```
>>> type(10)
<class 'int'>
>>> type(0o10)
<class 'int'>
>>> type(0x10)
<class 'int'>
```


Note: This is a good time to say that if you want to display a value in a REPL session, you need not use the `print ()` function. Simply type the value at the `>>>` prompt and press Enter to display it:

```
>>> 10
10
>>> 0x10
16
>>> 0b10
2
```

Many examples in this section will use this function.

Note that this does not work in a script file. A value that appears alone on a line in a script file will do nothing.

FLOATING-POINT NUMBERS

The `float` type in Python indicates a floating-point number. Floating point values are specified with a decimal point. Optionally, the character `e` and the `E` followed by a positive or negative integer can be added to specify scientific notation:

```
>>> 5.3
5.3
>>> type(5.3)
<class 'float'>
>>> 5.
5.0
>>> .3
0.3
>>> .5e7
5000000.0
>>> type(.5e7)
<class 'float'>
```

```
>>> 5.3e-5
0.00053
```

Deep Dive: floating point representation

The following is a bit more detailed information on how Python internally represents floating point numbers. You can easily use floating point numbers in Python without understanding them at this level; so don't worry if it sounds too complicated. The information is presented here if you are curious.

Almost all platforms represent floating point Python values as 64-bit "double-precision" values, according to IEEE 754. In this case, the maximum value a floating-point number can have is about 1.8×10^{308} . Python will report a number greater than the inf string:

```
>>> 1.79e30
1.79e+30
>>> 1.8e30
inf
```

The closest a nonzero digit can be to zero is approximately 5.0×10^{-324} . Something closer to zero than this is really zero:

```
>>> 5e-32
5e-32
>>> 1e-32
0.0
```

Floating point numbers are internally represented as binary (base-2) fractions. Most decimal fractions cannot be represented precisely as binary fractions; therefore, in most cases, the floating-point number internal representation is an approximation of the actual value. In practice, the difference between the represented value and the actual value is very small and generally should not cause major problems.

COMPLEX NUMBERS

Complex numbers are stated as <real part> + <imaginary part> j. For instance:

```
>>> 3+4j
(3+4j)
>>> type(3+4j)
<class 'complex'>
```

STRINGS

Strings are character data sequences. The string data type in Python is called str.

Literal strings can be delimited using single or double quotes. All characters between the opening delimiter and the corresponding closing delimiter are part of the string:

```
>>> print("This is a string.")
This is a string.
>>> type("This is a string.")
<class 'str'>
>>> print('I am too.')
I am too.
>>> type('I am too.')
<class 'str'>
```

A Python string can contain any desired character. The only limit is the memory resources of the machine. A string can also be empty:

```
>>> ""
"
```

What if you want to include a quotation mark character in the string itself? Your first impulse may be to try something like this:

```
>>> print('This string is made up of a single quote (') character.')
SyntaxError: Invalid syntax
```

As you can see, it doesn't work very well. The string in this example is opened with single quotes; therefore, Python assumes that the next single quote, in parentheses that should be part of the string, is the closing delimiter. The trailing single quotes are therefore scattered and cause the indicated syntax error.

If you wish to include both types of quotes in the string, the easiest way is to delimit the string with the other data type. If a string must contain single quotes, enclose it in double quotes and vice versa:

```
>>> print("This string is made up of a single quote (') character.")
```

This string is made up of a single quote (') character.

```
>>> print('This string is made up of a double quote (") character.')
```

This string is made up of a double quote (") character.

Escape sequences in strings

Sometimes we want Python to interpret a string or character differently in a string. This can happen in two ways:

- You can remove the special interpretation that certain characters are usually given in a string.
- You can apply a special interpretation to characters in a sequence that would normally be taken literally.

You can do this by introducing a backslash (\). A backslash in a string indicates that one or more of the following characters should be treated specially. (This is known as an escape sequence, because the backslash causes the following string to "escape" its usual meaning.)

Let's see how it works.

Suppressing the meaning of special characters

You have seen the challenges you can face when trying to include quotes in a string. If a sequence of characters is surrounded by single quotes, it is not possible to directly specify a single quote character in the sequence, because for this sequence, the single quotes will have a special meaning: ending the string:

```
>>> print ('This string is made up of single quotes ('). ')
```

SyntaxError: Invalid syntax

Specifying a backslash before quotation marks in an "escape" string forces Python to remove its usual special meaning. Therefore, it is simply interpreted as a single quote character:

```
>>> print ('This string is made up of single quotes ( \ ').')
```

This string is made up of single quotes (').

The same thing works on a string also delimited by double quotes:

```
>>> print ("This string is made up of double quotes ( \ ").")
```

This string is made up of a double quote character (").

Here is an array of escape sequences that cause Python to suppress the usual special interpretation of a character in a string:

\'

Usual Interpretation of Character(s) After Backslash - Terminates string with single quote opening delimiter

“Escaped” Interpretation - Literal single quote (') character

\"

Usual Interpretation of Character(s) After Backslash - Terminates string with double quote opening delimiter

“Escaped” Interpretation - Literal double quote (") character

\newline

Usual Interpretation of Character(s) After Backslash - Terminates input line

“Escaped” Interpretation - Newline is ignored

\\

Usual Interpretation of Character(s) After Backslash - Introduces escape sequence

“Escaped” Interpretation - Literal backslash (\) character

Usually, a newline character ends the line entry. So, hitting on Enter in the middle of a string, Python thinks it's incomplete:

```
>>> press ('a
```

SyntaxError: EOL while scanning string literal

To split a string into multiple lines, include a backslash before each new line and new lines will be ignored:

```
>>> print ('b \
... c \
... d ')
bcd
```

To add a literal backslash in a string, escape with a backslash:

```
>>> print ('foo \\ bar')
foo \ bar
```

Applying special meanings to characters

Next, assuming you need to create a string containing a tab character. Some text editors may let you insert a tab character directly into your code. But many programmers believe this is a bad practice for several reasons:

- The computer can distinguish between a sequence of space characters and a tab character, but you can't do this. For a human reading your code, the tab and space characters are visually indistinguishable.
- Some text editors are configured to automatically remove tab characters by expanding them to the appropriate number of spaces.
- Some Python REPL environments do not include tabs in the code.

In Python (and almost every other popular computer language), a tab character can be specified by the `\t` escape sequence:

```
>>> print ('foo \t bar')
foo bar
```

The escape sequence `\t` let the `t` character to lose its typical meaning, that of a literal letter. Instead, this is interpreted as a tab character.

Here are some lists of escape sequences that let Python to give special meaning instead of interpreting it literally:

- `\a` - ASCII Bell (BEL) character
- `\b` - ASCII Backspace (BS) character
- `\f` - ASCII Formfeed (FF) character
- `\n` - ASCII Linefeed (LF) character
- `\N{<name>}` - Character from Unicode database with given <name>
- `\r` - ASCII Carriage Return (CR) character
- `\t` - ASCII Horizontal Tab (TAB) character
- `\uxxxx` - Unicode character with 16-bit hex value xxxx
- `\Uxxxxxxxx` - Unicode character with 32-bit hex value xxxxxxxx
- `\v` - ASCII Vertical Tab (VT) character
- `\ooo` - Character with octal value ooo
- `\xhh` - Character with hex value hh

Examples:

```
>>> print("a \ tb")
a  b
>>> print("a \ 141 \ x61")
aaa
>>> print("a \ nb")
a
b
>>> print('\ u2192 \ N{rightwards arrow}')
→ →
```

This type of escape sequence is often used to enter characters that are not easily generated by the keyboard or which are not easily printable or readable.

Raw strings

R or r precedes a raw string literal, which specifies that the escape sequences of the associated sequence are not translated. The backslash is left in the string:

```
>>> print('foo\nbar')
```

```
foo
bar
>>> print(r'foo\nbar')
foo\nbar
>>> print('foo\\bar')
foo\bar
>>> print(R'foo\\bar')
foo\\bar
```

Triple-Quoted Strings

There is yet another way to delimit strings in Python. Strings with three quotation marks are placed in corresponding groups of three single quotation marks or three double quotation marks. The escape sequences always work with triple-quoted strings, but you can include single quotes, double quotes and new lines without escaping them. This gives a convenient way to create a string with single and double quotes:

```
>>> print("""This string is made up of a single (') and a double (")
quote.""")
```

This string is made up of a single (') and a double (") quote.

Since new lines can be added without escaping them, this also creates multiple line sequences:

```
>>> print (""" "This is a
string that stretches
across multiple lines "" "")
It's a
string that stretches
across multiple lines
```

In the next Python Program Structure tutorial, you will see how to use quoted strings to add an explanatory comment to Python code.

BOOLEAN TYPE, BOOLEAN CONTEXT, AND “TRUTHINESS”

Python 3 provides a boolean data type. Boolean objects can have one of two values, True or False:

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

As you will see in future tutorials, Python expressions are often evaluated in the Boolean context, in the sense that they are interpreted to represent truth or falsehood. Sometimes a value that is true in the Boolean context is called "true", while a value that is false in the Boolean context is called "falsey". (You may also see "falsey" spelled "falsy".)

The "truth" of an object of type Boolean is obvious: Boolean objects equal to True are true and those equal to false are false (false). But non-Boolean objects can also be evaluated in the Boolean context and determined to be true or false.

We have learnt more about evaluating objects in the Boolean context when we encountered logical operators in the previous Python Operators and Expressions tutorial.

BUILT-IN FUNCTIONS

The Python interpreter supports many built-in functions: sixty-eight, starting with Python 3.6. You will cover many of these topics in the discussions that follow, as they will emerge in context.

Math

- `divmod()` - Returns quotient and remainder of integer division
- `abs()` - Returns absolute value of a number
- `max()` - Returns the largest of the given arguments or items in an iterable
- `pow()` - Raises a number to a power

- `sum()` - Sums the items of an iterable
- `min()` - Returns the smallest of the given arguments or items in an iterable
- `round()` - Rounds a floating-point value

Type Conversion

- `bin()` - Converts an integer to a binary string
- `ascii()` - Returns a string containing a printable representation of an object
- `bool()` - Converts an argument to a Boolean value
- `complex()` - Returns a complex number constructed from arguments
- `chr()` - Returns string representation of character given by integer argument
- `float()` - Returns a floating-point object constructed from a number or string
- `int()` - Returns an integer object constructed from a number or string
- `hex()` - Converts an integer to a hexadecimal string
- `oct()` - Converts an integer to an octal string
- `repr()` - Returns a string that contains a printable illustration of an object
- `ord()` - Returns integer representation of a character
- `str()` - Returns a string version of an object
- `type()` - Creates a new type object or returns the type of an object or

Iterables and Iterators

- `any()` - Returns True if any elements of an iterable are true
- `all()` - Returns True if all elements of an iterable are true
- `enumerate()` - Returns a list of tuples containing values and indices and from an iterable
- `iter()` - Returns an iterator object
- `filter()` - Filters elements from an iterable
- `len()` - Returns the length of an object

- next() - Retrieves the next item from an iterator
- map() - Applies a function to every item of an iterable
- range() - Generates a range of integer values
- zip() - Creates an iterator that aggregates elements from iterables
- slice() - Returns a slice object
- reversed() - Returns a reverse iterator
- sorted() - Returns a sorted list from an iterable

Composite Data Type

- bytes() - Creates and returns a bytes object (similar to bytearray, but immutable)
- bytearray() - Creates and returns an object of the bytearray class
- dict() - Creates a dict object
- list() - Creates a list object
- frozenset() - Creates a frozenset object
- object() - Creates a new featureless object
- tuple() - Creates a tuple object
- set() - Creates a set object

Classes, Attributes, and Inheritance

- delattr() - Deletes an attribute from an object
- classmethod() - Returns a class method for a function
- getattr() - Returns the value of a named attribute of an object
- isinstance() - Determines whether an object is an instance of a given class
- hasattr() - Gives True if an object has a given attribute
- issubclass() - Determines whether a class is a subclass of a given class
- setattr() - Sets the value of a named attribute of an object
- property() - Returns a property value of a class
- super() - Gives a proxy object that delegates method calls to a sibling or parent class

Input/Output

- `input()` - Reads input from the console
- `print()` - Prints to a text stream or the console
- `format()` - Converts a value to a formatted representation
- `open()` - Opens a file and returns a file object

Variables, References, and Scope

- `globals()` - Returns a dictionary representing the current global symbol table
- `dir()` - Returns a list of names in current local scope or a list of object attributes
- `id()` - Returns the identity of an object
- `vars()` - Returns `__dict__` attribute for a module, class, or object
- `locals()` - Updates and returns a dictionary representing current local symbol table

Miscellaneous

- `compile()` - Compiles source into a code or AST object
- `callable()` - Returns True if object appears callable
- `eval()` - Evaluates a Python expression
- `hash()` - Returns the hash value of an object
- `exec()` - Implements dynamic execution of Python code
- `help()` - Invokes the built-in help system
- `staticmethod()` - Returns a static method for a function
- `memoryview()` - Returns a memory view object
- `__import__()` - Invoked by the import statement

TYPE CASTING IN PYTHON

Sometimes in our program it is necessary to convert from one data type to another, for example from integer to string. This is called a type casting.

There are three functions built into Python that allow us to type cast. These are the `int ()`, `float ()` and `str ()` functions.

The `int ()` function in Python accepts an appropriate float or string and converts it to an integer. To change an entire float, we can type `int`

(5.712987). We will get 5 as the result (anything after removing the decimal point). To change an entire string, we can type `int ("4")` and we will get 4. However, we cannot type `int ("Hello")` or `int ("4.22321")`. An error will be displayed in both cases.

The `float ()` function accepts an integer or an appropriate sequence and changes it to a float value. For example, if we type `float (2)` or `float ("2")`, we get 2.0. If we type `float ("2.09109")`, we get 2.09109, which is a float and not a string, because the quotes are removed.

The `str ()` function converts an integer or floats to a string. For example, if we type `str (2.1)`, we get "2.1".

Now that we've covered the basic data types in Python and how to type cast, let's move on to the more advanced data types.

List

List refers to a collection of data that is normally related. Instead of storing this data separately as variables, we can store it as a list. For example, suppose our program stores the age of 5 users. Instead of storing them as *user1Age*, *user2Age*, *user3Age*, *user4Age* , and *user5Age*, it makes more sense to store them as a list.

To declare a list, type *listName* = [*initial values*] . Note that we use square brackets [] when declaring a list. Multiple values are separated by a comma.

Example:

```
userAge = [31, 32, 33, 34, 35]
```

We can also declare a list without giving it an initial value. We simply write *listName* = []. What we have now is an empty list with no items. We need to use the `append ()` method mentioned below to add items to the list.

The individual values in the list can be accessed from their indexes, and indexes always start with ZERO, not 1. This is common practice in almost all programming languages, such as C and Java. Therefore, the first value has an index of 0, the next has an index of 1, and so on. For example, *userAge* [0] = 31, *userAge* [1] = 32

You can also access the values from a list at the back. The last entry in the list has an index of -1, the last second has an index of -2, and so on. So, `userAge [-1] = 35`, `userAge [-2] = 34` .

You may assign a list or part of it to a variable. If we write `userAge2 = userAge`, the variable `userAge2` becomes `[31, 32, 33, 34, 35]`.

If you write `userAge3 = userAge [3: 5]`, you assign index 2 items to index 4-1 of the `userAge` list for the `userAge3` list. In other words, `userAge3 = [33, 34]`.

The 2: 4 notation is known as the slice. Whenever we use slice notation in Python, the element from the initial index is always included, but the element is always excluded at the end. Therefore, notation 2: 4 refers to the index to index the elements 2 4-1 (that is to say, index 3), due to which `userAge3 = [23, 24]` and not `[23, 24, 25]`.

The section notation includes a third number referred to as the stepper. If we specify `userAge4 = userAge [1: 5: 2]`, we get a secondary list which consists of every second digit from index 1 to index 5-1 because the walkthrough is 2. So `userAge4 = [22, 24]`.

Additionally, section notations have useful defaults. The default for the first digit is zero, and the default for the second number is the list size that is being divided. For example, `userAge [: 4]` gives values from index 0 to index 4-1, while `userAge [1:]` gives values from index 1 to index 5-1 (because the size of `userAge` is 5, that is, `userAge` has 5 elements).

To modify the elements of a list, we write `listName [index of the element to modify] = new value` . For instance, if you decide to change the second element, write `userAge [1] = 5` . Your list becomes `userAge = [21, 5, 23, 24, 25]`

To add items, we use the `append ()` function. For example, if you type `userAge.append (99)`, add the value 99 to the end of the list. Your list is now `userAge = [21, 5, 23, 24, 25, 99]`

To delete the items, enter the `listName [index of the item to delete]`. For example, if you write `userAge [2]`, your list will now become `userAge = [21, 5, 24, 25, 99]` (the third item is removed)

To fully appreciate how a list works, try running the program below.

```

# declaring the list, the elements of the list can be of different data types
myList = [1, 2, 3, 4, 5, "Hello"]
#print the entire list. print (myList)
# You will receive [1, 2, 3, 4, 5, "Hello"]
#print the third element (remember: index starts at zero).
print (myList [2]) # You will receive 3
#print the last item. print (myList [-1]) # You will receive "Hello"
#assign myList (from index 1 to 4) for myList2 and displays myList2
myList2 = myList [1: 5] print (myList2)
# You will receive [2, 3, 4, 5]
#modify the second element of myList and display the updated list myList
[1] = 20
print (myList)
# You will receive [1, 20, 3, 4, 5, "Hello"]
# add a new item in myList and print the updated myList.append list ("How
are you")
print (myList)
# You will receive [1, 20, 3, 4, 5, "Hi", "How are you"]
#remove the sixth element of myList and display the updated list of
myList [5] print (myList)
# You will receive [1, 20, 3, 4, 5, "How are you"]

```

Tuples

Tuples are like lists, but you can't change their values. The initial values are those that will remain for the rest of the program. An example where tuples are useful is when the program needs to remember the names of the months of the year.

To declare a tuple, type tupleName = [initial values] . Note that we use square brackets [] when declaring a tuple. Multiple values are separated by a comma.

Example:

```
monthOfYear = ["Jan", "Feb", "Mar", "Apr", "May",  
               "Jun", "Jul", "Aug", "Set", "Out", "Nov", "Dec"]
```

You can access the individual values of a tuple using its indexes, just like in a list.

So, monthOfYear [0] = "Jan", monthOfYear [-1] = "Dec".

Dictionary

The dictionary is a collection of related data PAIRS. For example, if we want to store the username and age of 5 users, we can store them in a dictionary.

To declare a dictionary, type dictionaryName = {dictionary key:data}, with the requirement that the dictionary keys are unique (in a dictionary). In other words, you cannot declare a dictionary like this myDictionary = {"Tessy": 38, "Steve": 51, "Mary": 13}.

Indeed, "Peter" is used twice as a dictionary key. Note that we use curly braces {} when declaring a dictionary. Several pairs are separated by commas.

Example:

```
userNameAndAge = {"Tessy": 38, "Steve": 51, "Mary": 13,  
                  "Justin": "Not available"}
```

You can also declare a dictionary using the dict () method. To declare the dictionary userNameAndAge above, enter

```
userNameAndAge = dict (Peter = 38, Steve = 51, Alex = 13, Alvin  
                       = "Not available")
```

By using this method to declare a dictionary, the brackets () are used instead of curly brackets {} and no quotation marks are put for the dictionary keys.

To access the individual elements of the dictionary, we use the dictionary key, which is the first value of the {dictionary key: data} pair. For example, to get Steve's age, enter userNameAndAge ["Steve"].

You will receive 51.

To change items in a dictionary, we write DictionaryName [dictionary key of object to change] = new data. For example, to change the pair "Steve": 51, we would write userNameAndAge ["Steve"] = 21 . Our dictionary now

becomes `userNameAndAge = {"Peter": 38, "Steve": 21, "Alex": 13, "Alvin": "Not available"}`.

We can also declare a dictionary without giving it an initial value. We simply write `DictionaryName = {}`. What we have now is an empty dictionary with no items.

To add items to a dictionary, we write `DictionaryName [dictionary key] = data`. For example, if we want to add "Joe": 40 to our dictionary, we write `userNameAndAge ["Joe"] = 40`. Our dictionary now becomes `userNameAndAge = {"Peter": 38, "Steve": 21, "Alex": 13, "Alvin": "Not available", "Joe": 40}`

To remove items from a dictionary, write `del the dictionaryName [dictionary key]`. For example, to remove the pair "Alex": 13, we write `del userNameAndAge ["Alex"]`. Our dictionary now becomes `userNameAndAge = {"Peter": 38, "Steve": 21, "Alvin": "Not available", "Joe": 40}`

Run the program below to see it all in action.

#declaring the dictionary, dictionary data and keys can be of different data types

```
myDict = {"One":1.25, 3.2:"Two Point Five", 3:"+", 7.9:2}
```

#print the whole dictionary

```
print(myDict)
```

#You will get {2.5: three Point two, 3: '+', 'One': 1.25, 7.9: 2}

#Note that entries in a dictionary are not saved in the same order as the way we declare them.

#print the entry with key = "One". print(myDict["One"])

#You will get 1.25

#print the entry with key = 7.9. print(myDict[7.9])

#You will get 2

#modify the entry with key = 3.5 and print the updated dictionary

```
myDict[3.5] = "three and a Half" print(myDict)
```

#You will get {3.5: three and a Half, 3: '+', 'One': 1.35, 7.9: 2}

*#add a new entry and print the updated dictionary myDict["New entry"] =
"I'm new"*

print(myDict)

*#You will get {'New entry: 'I'm new', 3.5: Three and a Half', 3: '+', 'One':
1.35, 7.9: 2}*

#remove the entry with key = "One" and print the updated dictionary

del myDict["One"] print(myDict)

#You will get {'New entry: 'I'm new', 3.5: Three and a Half', 3: '+', 7.9: 2}

CHAPTER SIX: MAKING CHOICES AND DECISIONS

The ability to make a decision, to follow a path or another, is an essential part of useful work. Mathematics gives the computer the ability to obtain useful information. Decision making makes it possible to do something with the information once obtained. A computer system would be useless without the capability to make decisions. Therefore, any language you use will include the ability to make decisions in one way or another. This chapter explores the techniques used by Python to make decisions.

Think about the process you use to make a decision. You get the real value of something, compare it to the desired value, and act accordingly. For example, when you see the light and see that it is red, you compare the red light to the green light that you want, decide that the light is not green, and then to. Many individuals do not review the process they use because they use it so often every day. Decision making is natural for humans, but computers must perform the following tasks each time:

- Get the current or actual value of something.
- Compare the current or actual value to the desired value.
- Perform an action corresponding to the desired comparison result.

MAKING SIMPLE DECISIONS USING THE IF STATEMENT

The if statement is the easiest way to decide Python. It simply states that if a condition is satisfied, Python must perform the following steps. The following parts explain how to make use of the if statement to make decisions under different types in Python. You may be surprised to see what this simple statement can bring you.

Understanding the statement if

You use if statements regularly in everyday life. For example, you might say, "If it's Wednesday, I'll eat a tuna salad for lunch." The Python instruction itself is a little less detailed but follows the same pattern. Suppose you create a variable, `TestMe`, and set it to a value of 7, like this:

```
TestMe = 7
```

You can then tell the computer to check a value of 7 in `TestMe` , like this:

```
if TestMe == 7:  
    print ("TestMe equals 7!")
```

Each if statement python begins, curiously, with the word `if`. When Python sees `if`, it knows you want him to make a decision. After the word `if` comes a condition. A condition indicates the type of comparison you want Python to do. In a situation like this, you want Python to check if `TestMe` contains the value 7.

Note that the condition uses the relational equality operator, `==` and not the assignment operator, `=`. A common error encountered by developers is to use the assignment operator instead of the equality operator.

The condition always ends with a colon (`:`). If you do not provide a colon, Python does not know that the condition is complete and will continue to search for other conditions on which to base its decision. After the colon is all the tasks you want to perform in Python. In this case, Python displays a statement that `TestMe` equals 6.

Working with relational operators

How an operand on the left side of an expression compares to the operand on the right side of an expression is determined by the relational operator. Once the determination is made, it displays a true or false value that reflects the true value of the expression. For instance, `7 == 7` is true, while `7 == 8` is false. The steps below explain how to make and use an if statement. This example also appears with the download source code as `SimpleIf1.py`.

Launch a Python shell window.

The familiar Python prompt will be shown.

Type `TestNum = 7` and press Enter.

This step assigns a value of 7 to `TestMe`. Note that it makes use of the assignment operator (not the equality operator).

Type *if TestNum == 7 :* and press Enter.

This step generates an if statement that tests the value of TestNum using the equality operator. At this point, notice two features of the Python shell:

The word itself is highlighted in a different color from the rest of the statement.

The next line is automatically indented.

Type *print ("TestMe is 6!")* And press Enter.

Note that Python still does not execute the if statement. That goes back to the next line. The word 'print' is displayed in a special color because it is a function name. Also, the text appears in another color to indicate that it is a string value. Color coding let us easily understand how Python works.

Press Enter.

Shell Python overcomes this next line and executes the if statement. Note that the output is still in another color. Because TestNum contains a value of 7, the if statement works as expected.

Making multiple comparisons using logical operators

So far, the examples have shown a single comparison. In real life, you usually have to make several comparisons to meet various requirements. For instance, when baking cookies, if the timer has been turned off and the edges are brown, it is time to remove the cookies from the oven.

To perform multiple comparisons, you create different conditions using relational operators and combine them using logical operators. A logical operator describes how conditions can be combined. For instance, you can set *x == 6* and *y == 7* as two conditions to perform one or more tasks. The keyword *and* is a logical operator that indicates that the two conditions must be true.

Interval checking, which consists of determining whether the data falls between two values, is an important element in making your application safe and user-friendly. The following steps help you to see how to perform this task. Here, you create a file so that you can run the application multiple times. This example also appears with the download source code as SimpleIf3.py.

Launch a Python file window.

An editor where you can type the sample code is shown.

Enter the following code in the window - press Enter after each line:

```
Value = int (input ("choose any number which is greater than 1 but less than 20:"))
```

```
if (value> 1) and (value <= 20):
```

```
print ("You typed:", value)
```

The example starts with getting an input value. You have no clue what the user has typed other than that it's a value of some sort. Using the `int ()` function means that the user must enter an integer (one without a decimal part). Otherwise, the application throws an exception (an error indication, previous chapter describes the exceptions). This first check ensures that the entry is at least the correct type.

The `if` statement includes two conditions. The first implies that `Value` must be greater than 0. You can also display this condition as `Value> = 1`. The second condition tells you that `Value` must be equal to or less than 10. Only when `Value` satisfies both conditions will the statement succeed and print the value entered by the user.

Choose Run ⇌ Run Module.

You see an open Python environment window with a prompt to enter a number between 1 and 10.

Enter 5 and press Enter on your keyboard.

The application concludes that the number is in the correct range and generates a message.

Repeat steps 3 and 4, but type 33 instead of 5.

The application produces nothing because the number is in the wrong range. When you enter a value outside the programmed range, the instructions in the `"if"` block will not be executed.

Repeat steps 3 and 4, but type 6.5 instead of 5.

Python displays an error message. Although you think that 5.5 and 5 are numbers, Python considers the first number as a floating-point number and the second as an integer.

Repeat steps 3 and 4, but type Hi instead of 5.

Python displays the same error message as before. Python does not differentiate between wrong input types. You only know that the input type is incorrect and therefore unusable.

CHOOSE ALTERNATIVES USING THE IF ... ELSE STATEMENT

Many of the decisions made in an application fall into a category of choice of one of two options depending on conditions. For example, when you look at a traffic light, you choose one of two options: press the accelerator to continue or press the brake to stop. The chosen option depends on the conditions. A green light indicates that you can continue through the light; A red light tells you to stop. The following sections describe how Python allows you to choose between two alternatives.

Understanding the if...else statement

With Python, you select one of two alternatives using the else clause of the if statement. A clause is an addition to a block of code that modifies its operation. Most code blocks support multiple clauses. In this case, the else clause allows you to perform an alternative task, which maximizes the relevance of the if statement. Most developers make reference to the form of the if statement that contains the else clause as the if ... else statement, with the ellipses indicating that something is happening between if and else.

Sometimes, developers encounter challenges with the if ... else statement because they didn't bear in mind that the else clause always runs when the conditions of the if statement is not met. It is important to think about the consequences of always performing a set of tasks when conditions are wrong. Sometimes this can have unintended consequences.

Launch a Python file window.

An editor in which you can type your sample code is provided.

Enter the following code in the window - press Enter after each line:

```
Value = int (input ("choose any number which is greater than 1 but  
less than 20:"))  
if (value> 1) and (value <= 20):  
    print ("You typed:", value)
```

```
other:
```

```
print ("The value you entered is incorrect!")
```

As before, the sample receives an input from the user and then determines whether this entry is in the correct range. But, in this case, the else clause gives an alternative output message when the user types data outside the desired range.

Note that the else clause ends in a colon, just like the if statement. Most clauses that you use with Python instructions have two points associated with them so that Python knows when the clause is complete. If you get a coding error for your software, be sure to check for the presence of both dots if necessary.

Run module.

You see an open Python environment window with a prompt to enter a number between 1 and 10.

Type 5 and then press Enter.

The software will check if the number is in the correct range and generates a message. Repeat steps 3 and 4, but type 50 instead of 5.

This time, the application sends an error message. The user now knows that the entry is outside the desired range and that he must try to enter it again.

USING THE IF ... ELIF STATEMENT IN AN APPLICATION

You go to the restaurant and look at the menu. The restaurant offers eggs, cookies, waffles, and oatmeal for breakfast. After choosing one of the elements, the server brings it to you. Creating a menu selection requires something like a statement if ... else, but with a little more force. In this case, the elif clause is used to generate another set of conditions. The elif clause is a combination of a separate if statement and the else clause. The following steps describe how to make use of the if ... elif statement to create a menu. This example also appears with the download source code as IfElif.py.

Launch a Python file window.

An editor in which you can write the sample code is shown.

Enter the following code in the window - press Enter after each line:

```
print ("1. red")
print ("2. Orange")
print ("3. yellow")
print ("4. Green")
print ("5. blue")
print ("6. Violet")
Choose = int (input ("Choose your favorite color:"))
if (choose == 1):
    print ("You have chosen red!")
elif (choice == 2):
    print ("You have chosen Orange!")
elif (choice == 3):
    print ("You have chosen yellow!")
elif (choice == 4):
    print ("You have chosen green!")
elif (choice == 5):
    print ("You chose blue!")
elif (choice == 6):
    print ("You have chosen purple!")
other:
    print ("You made an invalid choice!")
```

The example begins by displaying a menu. The user sees a list of options for the application. Then he asks the user to make a selection that he places in Choice. Using the int () function ensures that the user can not enter anything other than a number.

Once the programmer has made his choice, the application searches for it in the list of possible values. In each case, Choice is compared to a specific value to create a condition for that value. When the user types 1, the application displays the message "You have chosen red!". If none of the

options are correct, the else clause is executed by default to inform the user that the input option is not valid.

Run module.

You see an open Python Shell window with the menu displayed. The application prompts you to select your favorite color.

Type 1 and press Enter.

The application displays the appropriate output message.

Repeat the 3rd and 4th steps, but type 5 instead of 1.

The application shows a different output message - the one connected with the requested color.

Repeat the 3rd and 4th steps, but type 8 instead of 1.

The application shows that you made the wrong choice.

Repeat the 3rd and 4th steps, but type red instead of 1.

The application shows the expected error message. Any application you create must be able to detect errors and incorrect entries. Chapter 9 shows how to handle errors so that they are friendly.

USING NESTED DECISION STATEMENTS

The decision-making process usually takes place by levels. For example, when you go to a restaurant and choose eggs for breakfast, you make a first level decision. Now the waiter asks you what type of bread you want with your eggs. The waiter would not ask that question if you asked for pancakes. The choice of toast would, therefore, become a second-level decision. When breakfast arrives, you decide if you want to use jelly on your toast. It is a third level decision. If you have selected a type of toast that does not work well with jelly, you may not have had to make that decision. This level decision-making process, each level depending on the decision taken at the previous level, is called nesting. Developers often use grouping techniques to create applications that can make complex, multi-entry decisions. The following sections describe various types of nesting that you can use in Python to make complex decisions.

USING MULTIPLE IF OR ELSE STATEMENTS

The most commonly used multiple selection techniques is a combination of if statement and if ... else statements. This form of selection is often referred to as a selection tree because it resembles the branches of a tree. But here, you follow a particular path to get the desired result. The sample program in this section also appears with the source code to download as `MultipleIfElse.py`.

Open a Python file window.

You see an editor where you can enter the sample code.

Enter the following code in the window - press Enter after each line:

```
x= float (input ("Input a number within the range of 100 and 200:"))
z = float (input ("Input a number within the range of 100 and
200:"))
if (x> = 100) and (x<=200):
if (z> = 100) and (z <= 200):
print ("Your secret number is:", x * z)
other:
print ("second incorrect value!")
other:
print ("First incorrect value!")
```

It's just an extension of the `IfElse.py` example that you see in the "Using the if ... else in an application" section of the chapter. Note, however, that the indentation is different. The second if ... else statement is indented in the first if ... else statement. Indentation let Python know that this is a second-level instruction.

Choose the Run Run module.

You see an open Python environment window with a prompt to enter a number between 1 and 10.

Enter 5 and press Enter.

The shell requests another number between 1 and 10.

Enter 2 and press Enter.

The combination of the two digits is displayed in the output.

PERFORMING REPETITIVE TASKS

So far, all the examples in the book have taken a series of measurements once and then stopped. Many individual tasks are repetitive. For instance, your doctor may state that you need to exercise more and ask you to do 100 pumps a day. If you get up, the exercise will not bring you many benefits, and you will certainly not follow the doctor's requests. Of course, as you know exactly how many pumps you do, you can perform the task a specific number of times. Python allows the same type of repetition with the `for` statement.

Unfortunately, you do not always know how often to complete a task. For example, consider the need to check a stack of coins for an extremely rare coin. Take only the first piece from the top, examine it and determine if it is a rare piece that does not complete the task. Instead, you should examine each piece, looking for the rare piece. Your stack may contain more than one. Only after examining all the parts of the stack can you see that the task is complete. However, since you cannot predict how many pieces are in the stack, you do not know how often to perform the task at the beginning. You know that the task is over when the stack is gone. Python performs this type of repetition with the `while` statement.

Most programming languages call any repetitive sequence of events a loop. The idea is to imagine the repetition as a circle, with the code rotating and rotating the tasks until the end of the loop. Loops are an essential part of the application's elements, such as menus. Writing more modern applications without using loops would be impossible.

In some cases, you must create loops in loops. For instance, to build a multiplication table, you make use of a loop in a loop. The inner loop calculates the values of the columns and the outer loop moves between the rows. You will see this example later in the chapter, so do not worry too much to understand exactly how these things work now.

Processing Data Using the `for` Statement

The first loop code block that most developers encounter is the `for` statement. It is not easy to imagine the creation of a common programming language without this assertion. In this case, the loop runs a fixed number of times, and you know how many times it will run before the loop starts.

Because everything concerning a for loop is known at first because loops tend to be the easiest type of loop to use. However, to use one, you need to know how many times to loop. The following parts describe the for loop in more detail.

Understanding the ‘for’ statement

A for loop begins with a 'to' statement. The statement describes how to loop. The Python loop works through any sequence. It does not matter if the sequence is a series of letters in a chain or elements of a collection. You can even be specific about the range of values to use by specifying the range () function. Here is a simple statement.

```
for Letter in "Howdy!":
```

The declaration begins with the keyword for. The next element is a variable that contains a single element in a sequence. In this case, the name of the variable is a letter. The keyword in informs Python that the sequence follows. In this case, the channel is the "Howdy" channel. The for statement must always terminate with a colon, as well as the statements of decision making.

Retracted under the for statement are the tasks that you want to perform in the for loop. Python considers each part of the next indented statement of the block of code that constitutes the for loop. Again, the for loop works exactly like the decision statements.

Creating a basic ‘for’ loop

The best way to see how a loop works are to create one. In this case, the below example makes use of a string for the string. The for loop processes each character of the string, in turn, until the characters are exhausted. This example also appears with the source code to download as SimpleFor.py.

Launch a Python file window.

An editor in which you can write the sample code is provided.

Enter the following code in the window - press Enter after each line:

```
LetterN = 1
for the letter in "Hello!":
    print ("Letter", LetterN, "is", Letter)
```

LetterN += 1

The example begins by initializing a variable, LetterN, to track the number of letters processed. Whenever the loop ends, LetterNum is updated to 1.

The for statement works through the string of letters in the "Howdy!" String. Put each letter, in turn, in the letter. The following code displays the current value of LetterNum and its associated character found in the letter.

Choose the Run Run module.

A Python shell window opens. The application displays the chain of letters with the letter number.

Running control with the break command

Life is often about exception to the rule. For example, you may want an assembly line to produce multiple clocks. However, at some point, the assembly line is left without any required parts. If the part is not available, the assembly line must stop at the middle of the treatment cycle. The account has not been completed, but the line must be stopped until the missing part is replenished.

Interruptions also occur on computers. You may be transmitting data from an online source in the event of a network failure and terminating the connection; the stream is temporarily dry, so the application runs out of tasks, even if the number of tasks defined is not completed.

The interrupt clause is used to interrupt a loop. However, you do not just put the pause clause in your code; you surround it with an if statement that sets the condition to pause. The statement could say something like this: If the flow is dry, leave the loop.

In this example, you discover what happens when the number reaches a certain level when processing a string. The example is somewhat invented in the interest of keeping things simple but reflects what could happen in the real world when a piece of data is too long to process (which may indicate an error condition). This example also appears with the source code to download as ForBreak.py.

Open a Python file window.

You see an editor where you can enter the sample code.

Enter the following code in the window - press Enter after each line:

```
Value = entry ("Enter less than 6 characters:")
LetterN = 1
for letter in value:
    print ("Letter", LetterN, "is", Letter)
    LetterN += 1
if LetterN > 7:
    print ("The string is too long!")
    pause
```

This example is based on the one discovered in the previous section. However, it allows the user to provide a variable length string. When the string has more than six characters, the application terminates.

The if statement carries the conditional code. When LetterN is greater than 7, it means that the string is too long. Note the second level of indentation which was used for the if statement. In this case, the user encounters an error message indicating that the string is too long, then pauses to end the loop.

Choose the Run Run module.

A Window of Python application Shell with a prompt requesting an entry is opened.

Type Hello and press Enter.

The application lists each character in the chain.

Repeat steps 3 and 4, but type I am too tall, instead of hello.

The application shows the expected error message and terminates the string at character 6

Controlling execution with the continue statement

Sometimes you want to check all the elements of a sequence, but you do not want to process certain elements. For example, you may decide to process all information from all cars in a database, except for brown cars. Maybe you do not need information about this particular color of the car. The interrupt clause terminates the cycle, so you can not use it in this situation. Otherwise, you will not see the remaining elements of the sequence.

The continuous clause is the alternative used by many developers. As with the break clause, the continuous clause appears in an if statement. However, rather than ending completely, processing continues with the next element in the sequence.

The following steps allow you to see how the continuation clause differs from the pause clause. In this case, the code declines to process the letter w but will process all other letters of the alphabet. This example also appears with the source code to download as ForContinue.py.

Launch a Python file window.

An editor in which you can write the sample code is shown.

Enter the following code in the window - press Enter after each line:

```
LetterN = 1
for the letter in "Howdy!":
    if the letter = "w":
        Carry on
    print ("Found w, untreated.") print ("Letter", LetterN, "is", Letter)
    LetterN += 1
```

This sample program is based on the one found in the "Creating a simple loop to another" section earlier in this book. However, this sample program adds an if statement with the continue clause in the if code block. Note the print () function that is part of the if code block. You never see this printed sequence because the iteration of the current loop ends immediately.

Choose Run ⇌ Run Module.

You see an open Python Shell window. The application displays the sequence of letters with the letter number. However, note the effect of the continuous clause: the letter w is not processed.

Execution control with the else statement

Python has another loopback clause that you will not find in other languages: else. The else clause makes the execution code possible even if you do not have elements to process in a sequence. For example, you may need to tell the user that there is simply nothing to do. This is what the

following example does. This example also appears with the source code to download as ForElse.py.

Launch a Python file window.

An editor in which you can write the sample code is shown.

Enter the following code in the window - press Enter after each line:

```
Value = entry ("Enter less than 6 characters:")
LetterNum = 1
for letter in value:
    print ("Letter", LetterNum, "is", Letter)
    LetterNum + = 1
other:
    print ("The string is empty".)
```

Processing data with the while statement

You use the while statement for situations where you are not sure how much data the application will need to process. Instead of asking Python to process a fixed number of items, you make use of the while statement to inform Python to continue processing items until items are exhausted. This type of loop is called upon when you need to carry out tasks like streaming data from a source, like a radio station or downloading files of unknown size. Any situation in which you can not define the amount of data the application will handle from the start will be a good candidate for the while statement described in more detail in the following sections.

Understand the while statement

The while statement works with a condition and not with a string. The condition indicates that the while statement must execute a task until the condition is no longer true. For example, imagine a multi-client deli in front of the counter. The seller continues to serve customers until there are no more customers in the queue. The line can (and probably will) grow as other customers are managed. It is therefore impossible to know from the start how many customers will be served. All the seller understands is that it is important to continue serving customers until there is nothing left. Here's what a while statement might look like:

```
while total <510:
```

The declaration begins with the while keyword. He then adds a condition. In this case, a variable, total, must be less than 510 for the loop to continue. Nothing specifies the Current Total value, and the code does not define how the Total value will change. The only thing is known when Python executes the statement is that Total must be less than 510 for the loop to continue with preformatting tasks. The declaration ends with two points, and the tasks are indented under the declaration.

Because the while statement does not perform a series of tasks multiple times, it is possible to create an infinite loop, which means that the loop never ends. For example, suppose that Total is set to 0 when the loop starts and the last condition is that Total should be less than 5. If the Total value never increases, the loop will continue to run indefinitely (or at least until the end of the reading, the computer is off). Endless loops can cause all sorts of weird problems in systems, such as sluggishness and even computer hang-ups, so it's best not to encounter them. You must always generate a method for the loop to complete when you use a while loop (as opposed to the for loop, in which the end of the sequence determines the end of the loop). Therefore, when you use the while statement, you must perform three tasks:

Create the environment for the condition (for example, set Sum to 0).

Specify the condition in the while statement (such as Sum <5).

Update the condition as needed to make sure the loop ends (for example, adding Sum += 1 to the code block while).

As with the for statement, you can change the default behavior of the while statement. You have access to modify the behavior of the while statement:

Break: Ends the current loop.

Continue: immediately finish processing the current item.

Pass: finishes processing the current element after completing the states of the if block.

Else: provides another processing technique when the conditions are not met for the loop.

Making use of the while statement in an application

You can make use of the while statement in several ways, but this first example is simple. It simply displays a number based on the initial and final conditions of a variable named Sum. The steps below allow you to create and test the sample code. This example also appears with the download source code as SimpleWhile.py.

Launch a Python file window.

An editor in which you can write the sample code is shown.

Enter the following code in the window - press Enter after each line:

```
Sum = 0
while Sum <5:
    print (Sum)
    Sum += 1
```

Nesting Loop Instructions

A For or While loop, in most cases, can be used to achieve the same effect. The effect is the same, but the ways work differently. In this sample code, you create a multiplication table generator by nesting a while loop in a for loop. Because you want the output to be beautiful, you also use some formatting. This example also appears with the source code to download as ForElse.py.

Launch a Python file window.

An editor in which you can write the sample code is shown.

Enter the following code in the window - press Enter after each line:

```
Y = 1
Z = 1
print ('{:> 4}'.format ( ' '), end = ' ')
for Y in the meantime (1, 11):
    print ('{:> 4}'.format (Y), end = ' ')
    impression ()
for Y in the meantime (1.11):
    print ('{:> 4}'.format (Y), end = ' ')
```

```
while Y <= 10:  
    print ('{:> 4}'.format (Y * Z), end = ' ')  
    Z += 1  
    impression ()  
    Z = 1
```

This example starts by creating two variables, Y and Z, to keep the row and column values of the table. Y is the row variable, and Z is the column variable.

For the table to be readable, this example must create a title at the top and another at the side. When users discover a 1 at the top and 1 at a side and follow these values as long as they intersect in the table, they can see the value of the two numbers as they are multiplied.

The first print () statement will add a space (because nothing appears in the corner of the table. The formatting declaration only creates a space of 4 characters wide and includes a space. The {:> 4 part of the code defines the size of the column. The formatting function (") determines what appears in this space. The final attribute of the print () statement changes the last character of a carriage return to a single space.

The first for loop displays numbers from 1 to 10 at the beginning of the table. The range () function builds the sequence of numbers for you. When you use the range () function, you specify the initial value, which is 1 in this case, and one more value than the final value, which is 11 in this case.

At this stage, the cursor is at the end of the title line. To transfer it to the next line, the code issues a print () call without further information.

Even though the following code seems quite complex, you can find out if you look at it one line at a time. The multiplication table shows values from 1 * 1 to 10 * 10, so you need ten rows and ten columns to display the information. The instruction tells Python to create ten lines.

Look again at the header of the line. The first print call () displays the value of the line header. Of course, you must format this information, and the code uses a four-character space that ends with space, rather than a newline, to continue printing information on that line.

The while loop comes next. This loop prints the columns on an individual line. The column values are the final values of $Y * Z$. Again, the output is formatted to occupy four spaces. The while loop ends when Z is updated to the next value using $Z + = 1$.

You are now back in the loop. The `print ()` statement ends the current line. Also, Z must be reset to 1 to be ready for the beginning of the next line, which starts at 1.

CHAPTER SEVEN: ERROR MANAGEMENT

It should not be shocking that errors occur - applications are written by humans, and humans make mistakes. Most developers call application error exceptions, which means they are the exception to the rule. Because exceptions occur in applications, you need to detect them and fix them as much as you can. Detecting and dealing with an exception is called error handling or exception handling. To correctly detect errors, you must know the sources of error and know why the errors occur first. When you detect the error, you must handle it by capturing the exception. Capturing an exception means looking at it and possibly doing something about it. Therefore, another part of this chapter deals with the discovery of exception handling in your application.

Sometimes your code detects an application error. When this happens, you must throw or raise an exception. You discover that both terms are used for the same thing, which means that your code has encountered an error that can not be manipulated, and then passed the error information to another piece of code to manipulate (interpret, process, and, with a bit of luck correct exception). In some cases, you make use of custom error message objects to convey information. Although Python has a large number of generic message objects that cover most situations, some are special. For instance, you may want to provide special support for a database application, and Python will not normally cover this event with a generic message object. It's essential to know when to control exceptions locally, when to give them to the code that requested for your code, and when to create unique exceptions so that each part of the application understands how to handle the exception - all topics covered in this chapter.

You may also need to make sure that your application normally handles an exception, even if it means terminating the application. Fortunately, Python provides the final clause, which is always executed even in case of exception. You can put code to close files or perform other essential tasks in the code block associated with this clause. Even if you do not do this task all the time, this is the last topic in the chapter.

KNOWING WHY PYTHON DOES NOT UNDERSTAND YOU

Developers are often frustrated with programming languages and computers because they seem to do everything to cause communication problems. Of course, programming languages and computers are inanimate - we do not want anything from them. Programming languages and computers do not think either; they accept what the developer has to say literally. There is a problem.

Neither the computer nor Python "will know what you mean" when entering instructions as a code. Both follow the instructions you provide to the letter and as you provide them. You may not want to tell Python to delete a data file unless an absurd condition occurs. However, if you do not clarify the conditions, Python will delete the file, whether the condition exists or not. When such an error occurs, people often say that the application contains a bug. The bugs are program errors that can be removed using a debugger. (A debugger is a unique kind of tool that allows you to pause or pause running applications, examine the contents of variables, and often dissect the application to see what makes it work.)

Errors occur in several cases when the developer makes assumptions that are not true. Of course, this includes assumptions about the user of the application, who probably does not care about the extreme level of attention you received when creating your application. The user will enter incorrect data. Again, Python will not know if the data is incorrect and will care and treat it even if its purpose is to prevent an incorrect entry. Python does not understand good or bad data concepts; It simply processes the received data according to the defined rules, which means that you need to define rules to protect users against themselves.

Python is neither proactive nor creative - these qualities only exist at the developer. When the user does something unexpected, or a network error occurs, Python does not create a solution to the problem. It only deals with the code. If you do not provide the code to handle the error, it is likely that the application will fail and fail incorrectly, possibly resulting in the transfer of all user data. Of course, the developer can not anticipate all possible error situations. This is why the most complex applications contain errors - omission errors in this case.

Some developers think they can create a code foolproof, despite the absurdity of thinking that such a code is possible. Smart developers assume that several bugs will go through the process of sorting the code, that users will continue to carry out unexpected actions, and that even the smartest developer cannot predict all error conditions possible. Always assume that your application is prone to errors that may cause exceptions. This way, you will have the necessary state of mind to make your application more reliable.

CONSIDERING THE SOURCES OF ERRORS

You can guess the possible sources of error in your application by reading tea leaves, but this is not the best way to do things. Mistakes fall into distinct categories that help you predict (to some extent) when and where they will occur. By thinking about these kinds as you work with your application, you are much more likely to discover potential sources of error before they occur and cause potential damage. The two main categories are

- Errors that occur at a given time
- Errors of a specific type

The following sections describe these two categories in more detail. The general concept is that you need to think about classifying errors to be able to locate and fix any errors in your application before they become a problem.

SORTING WHEN ERRORS OCCUR

Errors occur at specific times. The two main deadlines are

- Compilation time
- Execution time

No matter when an error occurs, your application behaves badly.

The following sections describe each period.

Compilation time

A compilation error occurs when you tell Python to run the application. Python must interpret the code and put it in a format that is understandable to the computer before it can run the application. A computer depends on the machine code specific to that processor and architecture. If the instructions you have written are poorly formed or do not contain the necessary information, Python will not be able to perform the required conversion. There is an error that you must correct before the application can run.

Fortunately, compilation errors are the easiest to detect and correct. Because the application does not run with a compilation error, the user never sees this category of error. You correct this type of error when writing your code.

The occurrence of an error during compilation should inform you that other typos or omissions may exist in the code. It is always helpful to check the code to make sure there are no other potential issues that might not appear in the build cycle.

Execution time

A runtime error occurs after Python has compiled the code you are writing and the computer has started running it. Runtime errors come in different types, and some are more difficult to detect than others. You know that you have a runtime error when the user complains of an incorrect (or at least unstable) output or when the application stops working and shows an exception dialog box.

All runtime errors do not generate an exception. Some runtime errors cause instability (the application crashes), erroneous output, or data corruption. Runtime errors may affect other applications or create unexpected damage to the platform on which the application is running. In short, run-time errors can cause problems depending on the type of error you are currently using.

Many runtime errors are caused by wrong codes. For example, you can incorrectly type the name of a variable, preventing Python from placing information in the correct variable at run time. Leaving an argument optional but necessary when calling a method can also cause problems. Here are some examples of commission errors, which are specific errors associated with your code. In general, you can find these types of errors by simply reading your code line by line to check for errors or by using a debugger.

Runtime errors can also come from external sources that are not associated with your code. For example, the user may enter incorrect information not expected by the application, causing an exception. A network error can lead to access denial to a required resource. Sometimes even the hardware has a fault that causes a non-repeatable application error. These are all examples of omission errors, from which the application can recover if your application has an interrupt code. It is vital to consider both types of runtime errors (commission errors and omissions) when creating your application.

CONCLUSION

Learning to start with computer programming can seem like a challenge. There are many different programming options to choose from, but many are difficult to learn, it will take time to figure out, and they won't always do everything you need to know. A lot of people worry about being really smart or having a lot of experience in education and programming before they reach the level of programming they want. But with Python, even a beginner can start programming.

Python has made it so easy to start writing code whether you are a beginner or an expert. The language is based on English, so it's easy to read and gets rid of a lot of other symbols that make coding difficult for others. And since this is a user domain, anyone can make changes and view other code to make it easier.

This guide has taken the time to talk about the different functions you can perform in Python and how easily a beginner can get started. You will see that this process is straightforward and that you can learn with a little practice. It's easy to use, works on many platforms, and even the latest Mac systems come with this one already downloaded.

When you're ready to start programming, or want to find a program that will do a lot of things without all the hassle, check out Python. It's one of the most widely-used options when it comes to programming and you'll find it easy to read and study, even if you don't know where to start. Use this guide to explore some of the basic functions and learn more about the Python program.