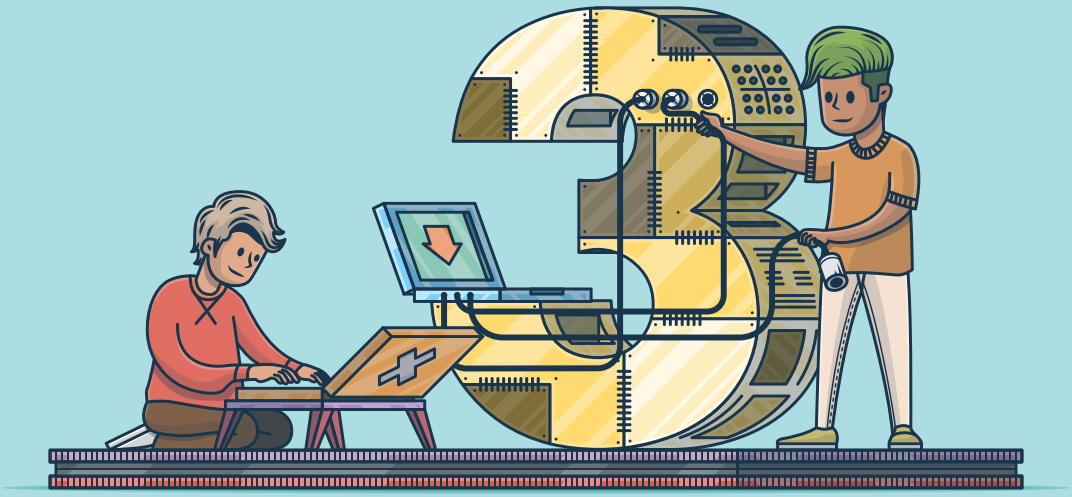
Real Python

PYTHON BASICS



A PRACTICAL INTRODUCTION TO PYTHON 3

FOURTH EDITION

BY THE REALPYTHON.COM TUTORIAL TEAM
FLETCHER HEISLER, DAVID AMOS, DAN BADER, JOANNA JABLONSKI

Python Basics: A Practical Introduction to Python 3

Real Python

Python Basics

Fletcher Heisler, David Amos, Dan Bader, Joanna Jablonski

For online information and ordering of this and other books by Real Python, please visit realpython.com. For more information, please contact us at info@realpython.com.

ISBN: 9781775093329 (paperback)

ISBN: 9781775093336 (electronic)

Cover design by Aldren Santos

This is a sample from "Python Basics: A Practical Introduction to Python 3"

With the full version of the book you get a complete Python curriculum to go all the way from beginner to intermediate-level. Every step along the way is explained and illustrated with short & clear code samples.

Coding exercises within each chapter and our interactive quizzes help fast-track your progress and ensure you always know what to focus on next.

Become a fluent Pythonista and gain programming knowledge you can apply in the real-world, today:

If you enjoyed the sample chapters you can purchase a full version of the book at realpython.com/pybasics-book.

What Pythonistas Say About *Python Basics: A Practical Introduction to Python 3*

"I love [the book]! The wording is casual, easy to understand, and makes the information flow well. I never feel lost in the material, and it's not too dense so it's easy for me to review older chapters over and over.

I've looked at over 10 different Python tutorials/books/online courses, and I've probably learned the most from Real Python!"

— Thomas Wong

"Three years later and I still return to my Real Python books when I need a quick refresher on usage of vital Python commands."

— Rob Fowler

"I floundered for a long time trying to teach myself. I slogged through dozens of incomplete online tutorials. I snoozed through hours of boring screencasts. I gave up on countless crufty books from big-time publishers. And then I found Real Python.

The easy-to-follow, step-by-step instructions break the big concepts down into bite-sized chunks written in plain English. The authors never forget their audience and are consistently thorough and detailed in their explanations. I'm up and running now, but I constantly refer to the material for guidance."

— Jared Nielsen

"I love the book because at the end of each particular lesson there are real world and interesting challenges. I just built a savings estimator that actually reflects my savings account – neat!"

— Drew Prescott

"As a practice of what you taught I started building simple scripts for people on my team to help them in their everyday duties. When my managers noticed that, I was offered a new position as a developer.

I know there is heaps of things to learn and there will be huge challenges, but I finally started doing what I really came to like.

Once again: MANY THANKS!"

— Kamil

"What I found great about the Real Python courses compared to others is how they explain things in the simplest way possible.

A lot of courses, in any discipline really, require the learning of a lot of jargon when in fact what is being taught could be taught quickly and succinctly without too much of it. The courses do a very good job of keeping the examples interesting."

— Stephen Grady

"After reading the first Real Python course I wrote a script to automate a mundane task at work. What used to take me three to five hours now takes less than ten minutes!"

— Brandon Youngdale

"Honestly, throughout this whole process what I found was just me looking really hard for things that could maybe be added or improved, but this tutorial is amazing! You do a wonderful job of explaining and teaching Python in a way that people like me, a complete novice, could really grasp.

The flow of the lessons works perfectly throughout. The exercises truly helped along the way and you feel very accomplished when you finish up the book. I think you have a gift for making Python seem more attainable to people outside the programming world.

This is something I never thought I would be doing or learning and with a little push from you I am learning it and I can see that it will be nothing but beneficial to me in the future!"

— Shea Klusewicz

"The authors of the courses have NOT forgotten what it is like to be a beginner – something that many authors do – and assume nothing about their readers, which makes the courses fantastic reads. The courses are also accompanied by some great videos as well as plenty of references for extra learning, homework assignments and example code that you can experiment with and extend.

I really liked that there was always full code examples and each line of code had good comments so you can see what is doing what.

I now have a number of books on Python and the Real Python ones are the only ones I have actually finished cover to cover, and they are hands down the best on the market. If like me, you're not a programmer (I work in online marketing) you'll find these courses to be like a mentor due to the clear, fluff-free explanations! Highly recommended!"

— Craig Addyman

About the Authors

At Real Python you'll learn real-world programming skills from a community of professional Pythonistas from all around the world.

The realpython.com website launched in 2012 and currently helps more than a million Python developers each month with free programming tutorials and in-depth learning resources.

Everyone who worked on this book is a *practitioner* with several years of professional experience in the software industry. Here are the members of the Real Python Tutorial Team who worked on *Python Basics*:

Fletcher Heisler is the founder of Hunter2, where he teaches developers how to hack and secure modern web apps. As one of the founding members of Real Python, Fletcher wrote the original version of this book in 2012.

David Amos is a mathematician by training, a data scientist/Python developer by profession, and a coffee junkie by choice. He is a member of the Real Python tutorial team and rewrote large parts of this book to update it to Python 3.

Dan Bader is the owner and Editor in Chief of Real Python and a complete Python nut. When he's not busy working on the Real Python learning platform he helps Python developers take their coding skills to the next level with tutorials, books, and online training.

Joanna Jablonski is the Executive Editor of Real Python. She loves natural languages just as much as she loves programming languages. When she's not producing educational materials to help Python developers level up, she's finding new ways to optimize various aspects of her life.

Contents

Foreword

Hello and welcome to **Python Basics: A Practical Introduction to Python 3**. I hope you are ready to learn why so many professional and hobbyist developers are drawn to Python and how you can begin using it on your projects, small and large, right away.

This book is targeted at beginners who either know a little programming but not the Python language and ecosystem, as well as complete beginners.

If you don't have a Computer Science degree, don't worry. Fletcher, David, Dan, and Joanna will guide you through the important computing concepts while teaching you the Python basics, and just as importantly, skipping the unnecessary details at first.

Python Is a Full-Spectrum Language

When learning a new programming language, you don't yet have the experience to judge how well it will serve you in the long run. If you are considering Python, let me assure you that this is a good choice. One key reason is that Python is a **full-spectrum** language.

What do I mean by this? Some languages are very good for beginners. They hold your hand and make programming super easy. We can go to the extreme and look at visual languages such as Scratch.

Here you get blocks that represent programming concepts, like variables, loops, method calls, and so on, and you drag and drop them on a visual surface. Scratch may be easy to get started with for sim-

ple programs. But you cannot build professional applications with it. Name one Fortune 500 company that powers its core business logic with Scratch.

Came up empty? Me too—because that would be insanity.

Other languages are incredibly powerful for expert developers. The most popular one in this category is likely C++ and its close relative C. Whatever web browser you used today was likely written in C or C++. Your operating system running that browser was also very likely built with C/C++. Your favorite first-person shooter or strategy video game? You nailed it: C/C++.

You can do amazing things with these languages. But they are wholly unwelcoming to newcomers looking for a gentle introduction.

You might not have read a lot of C++ code. It can almost make your eyes burn. Here's an example, a real albeit complex one:

```
template <typename T>
_Defer<void(*(PID<T>, void (T::*)(void)))
    (const PID<T>&, void (T::*)(void))>
defer(const PID<T>& pid, void (T::*method)(void))
{
  void (*dispatch)(const PID<T>&, void (T::*)(void)) =
    &process::template dispatch<T>;
  return std::tr1::bind(dispatch, pid, method);
}
```

Please, just no.

Both Scratch and C++ are decidedly not what I would call full-spectrum languages. In the Scratch level, it's easy to start but you have to switch to a "real" language to build real applications. Conversely, you can build real apps with C++, yet there is no gentle on-ramp. You dive head first into all the complexity of that language which exists to support these rich applications.

Python, on the other hand, is special. It is a full-spectrum language. We often judge the simplicity of a language based on the "hello world" test. That is, what syntax and actions are necessary to get that language to output "hello world" to the user? In Python, it couldn't be simpler:

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

That's it. However, I find this an unsatisfying test.

The "hello world" test is useful but really not enough to show the power or complexity of a language. Let's try another example. Not everything here needs to make total sense, just follow along to get the Zen of it. The book covers these concepts and more as you go through. The next example is certainly something you could write near the end.

Here's the new test: What would it take to write a program that accesses an external website, downloads the content to your app in memory, then displays a subsection of that content to the user? Let's try that experiment with Python 3 with the help of the `requests` package (which needs to be installed—more on that in chapter 12):

```
import requests
resp = requests.get("https://realpython.com")
html = resp.text
print(html[205:294])
```

Incredibly, that's it. When run, the output is (something like):

```
<title>Python Tutorials – Real Python</title>
<meta name="author" content="Real Python">
```

This is the easy, getting started side of the spectrum of Python. A few trivial lines and incredible power is unleashed. Because Python has access to so many powerful but well-packaged libraries, such as `requests`, it is often described as *having batteries included*.

So there you have a simple powerful starter example. On the real apps

side of things, we have many incredible applications written in Python as well.

YouTube, the world's most popular video streaming site, is written in Python and processes more than 1,000,000 requests per second. Instagram is another example of a Python application. More close to home, we even have realpython.com and my sites such as talkpython.fm.

This full-spectrum aspect of Python means you can start easy and adopt more advanced features as you need them when your application demands grow.

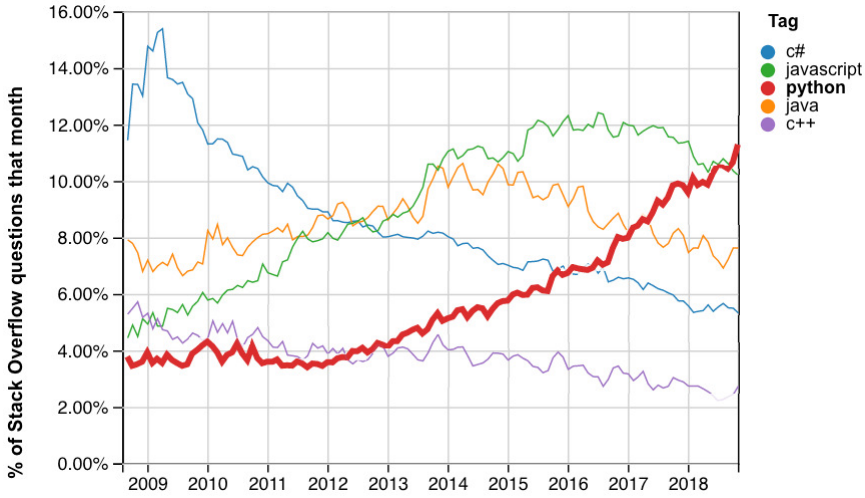
Python Is Popular

You might have heard that Python is popular. On one hand, it may seem that it doesn't really matter how popular a language is if you can build the app you want to build with it.

For better or worse, in software development popularity is a strong indicator of the quality of libraries you will have available as well the number of job openings there are. In short, you should tend to gravitate towards more popular technologies as there will be more choices and integrations available.

So, is Python actually that popular? Yes it is. You'll of course find a lot of hype and hyperbole. But there are plenty of stats to back this one. Let's look at some analytics available and presented by StackOverflow.com.

They run a site called **StackOverflow Trends**. Here you can look at the trends for various technologies by tag. When we compare Python to the other likely candidates you could pick to learn programming, you'll see one is unlike the others:



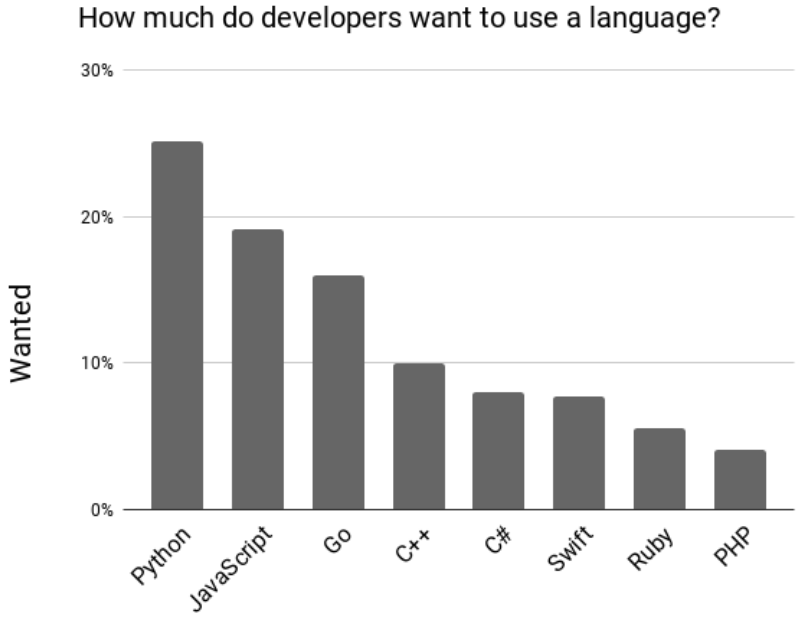
You can explore this chart and create similar charts to this one over at insights.stackoverflow.com/trends.

Notice the incredible growth of Python compared to the flatline or even downward trend of the other usual candidates! If you are betting your future on the success of a given technology, which one would you choose from this list?

That's just one chart, what does it really tell us? Well, let's look at another. StackOverflow does a yearly survey of developers. It's comprehensive and very well done. You can find the full 2018 results at insights.stackoverflow.com/survey/2018/. From that writeup, I'd like to call your attention to a section entitled Most Loved, Dreaded, and Wanted Languages. In the most wanted section, you'll find responses for:

> *Developers who are not developing with the language or technology but have expressed interest in developing with it.*

Again, in the graph below, you'll see that Python is topping the charts and well above even second place:



So if you agree with me that the relative popularity of a programming language matters. Python is clearly a good choice.

We Don't Need You to Be a Computer Scientist

One other point I do want to emphasis as you start this journey of learning Python is that we don't need you to be a computer scientist. If that's your goal, great. Learning Python is a powerful step in that direction. But learning programming is often framed in the shape of "we have all these developer jobs going unfilled, we need software developers!"

That may or may not be true. But more importantly for you, programming (even a little programming) can be a superpower for you personally.

To illustrate this idea, suppose you are a biologist. Should you drop out of biology and get a front-end web developer job? Probably not. But having skills such as the one I opened this foreword with, using requests to get data from the web, will be incredible powerful for you as you do biology.

Rather than manually exporting and scraping data from the web or spreadsheets, with Python you can scrape 1,000's of data sources or spreadsheets in the time it takes you to do just one manually. Python skills can be what takes your *biology power* and amplifies it well beyond your colleagues' and makes it your *superpower*.

Dan and Real Python

Finally, let me leave you with a comment on your authors. Dan Bader along with the other Real Python authors work day in and out to bring clear and powerful explanations of Python concepts to all of us via realpython.com.

They have a unique view into the Python ecosystem and are keyed into what beginners need to know.

I'm confident leaving you in their hands on this Python journey. Go forth and learn this amazing language using this great book. Most importantly, remember to have fun!

— **Michael Kennedy**, Founder of Talk Python (@mkennedy)

Chapter 1

Introduction

Welcome to Real Python's *Python Basics* book, fully updated for Python 3.8! In this book you'll learn real-world Python programming techniques, illustrated with useful and interesting examples.

Whether you're new to programming or a professional software developer looking to dive into a new language, this book will teach you all of the practical Python that you need to get started on projects on your own.

No matter what your ultimate goals may be, if you work with a computer at all, you will soon be finding endless ways to improve your life by automating tasks and solving problems through Python programs that you create.

But what's so great about Python as a programming language? Python is open-source freeware, meaning you can download it for free and use it for any purpose, commercial or not.

Python also has an amazing community that has built a number of additional useful tools you can use in your own programs. Need to work with PDF documents? There's a comprehensive tool for that. Want to collect data from web pages? No need to start from scratch!

Python was built to be easier to use than other programming lan-

guages. It's usually much easier to read Python code and much faster to write code in Python than in other languages.

For instance, here's some simple code written in C, another commonly used programming language:

```
#include <stdio.h>

int main(void)
{
    printf("Hello, world\n");
}
```

All the program does is show the text `Hello, world` on the screen. That was a lot of work to output one phrase! Here's the same program, written in Python:

```
print("Hello, world")
```

That's pretty simple, right? The Python code is faster to write and easier to read. We find that it looks friendlier and more approachable, too!

At the same time, Python has all the functionality of other languages and more. You might be surprised how many professional products are built on Python code: Instagram, YouTube, Reddit, Spotify, to name just a few.

Not only is Python a friendly and fun language to learn—it also powers the technology behind multiple world-class companies and offers fantastic career opportunities for any programmer who masters it.

1.1 Why This Book?

Let's face it, there's an overwhelming amount of information about Python on the internet.

But many beginners who are studying on their own have trouble fig-

uring out *what* to learn and *in what order* to learn it.

You may be asking yourself, "What should I learn about Python in the beginning to get a strong foundation?" If so, this book is for you—whether you're a complete beginner or already dabbled in Python or other languages before.

Python Basics is written in plain English and breaks down the core concepts you really need to know into bite-sized chunks. This means you'll know "enough to be dangerous" with Python, fast.

Instead of just going through a boring list of language features, you'll see exactly how the different building blocks fit together and what's involved in building real applications and scripts with Python.

Step by step you'll master fundamental Python concepts that will help you get started on your journey to learn Python.

Many programming books try to cover every last possible variation of every command which makes it easy for readers to get lost in the details. This approach is great if you're looking for a reference manual, but it's a horrible way to learn a programming language. Not only do you spend most of your time cramming things into your head you'll never use, it also isn't any fun!

This book is built on the 80/20 principle. We will cover the commands and techniques used in the vast majority of cases and focus on how to program real-world solutions to problems that will help make your life easier.

This way, we guarantee that you will:

- Learn useful programming techniques quickly
- Spend less time struggling with unimportant complications
- Find more practical uses for Python in your own life
- Have more fun in the process

Once you've mastered the material in this book, you will have gained

a strong enough foundation that venturing out into more advanced territory on your own will be a breeze.

So dive in! Learn to program in a widely used, free language that can do more than you ever thought was possible.

1.2 About Real Python

At Real Python, you'll learn real-world programming skills from a community of professional Pythonistas from all around the world.

The realpython.com website launched in 2012 and currently helps more than a million Python developers each month with books, programming tutorials, and other in-depth learning resources.

Everyone who worked on this book is a *Python practitioner* recruited from the Real Python team with several years of professional experience in the software industry.

Here's where you can find Real Python on the web:

- realpython.com
- @realpython on Twitter
- The Real Python Email Newsletter

1.3 How to Use This Book

The first half of this book is a quick but thorough overview of all the Python fundamentals. You do not need any prior experience with programming to get started. The second half is focused on finding practical solutions to interesting, real-world coding problems.

As a beginner, we recommend that you go through the first half of this book from start to end. The second half covers topics that don't overlap as much so you can jump around more easily, but the chapters do increase in difficulty as you go along.

If you are a more experienced programmer, then you may find yourself heading toward the second part of the book right away. But don't neglect getting a strong foundation in the basics first and be sure to fill in any knowledge gaps along the way.

Most sections within a chapter are followed by **review exercises** to help you make sure that you've mastered all the topics covered. There are also a number of **code challenges**, which are more involved and usually require you to tie together a number of different concepts from previous chapters.

The practice files that accompany this book also include full solutions to the challenges as well as some of the trickier exercises. But to get the most out of the material, you should try your best to solve the challenge problems on your own before looking at the example solutions.

If you're completely new to programming, you may want to supplement the first few chapters with additional practice. We recommend working through the *Python Fundamentals* tutorials available for free at realpython.com to make sure you are on solid footing.

If you have any questions or feedback about the book, you're always welcome to contact us directly.

Learning by Doing

This book is all about learning by doing, so be sure to *actually type in* the code snippets you encounter in the book. For best results, we recommend that you avoid copying and pasting the code examples.

You will learn the concepts better and pick up the syntax faster if you type out each line of code yourself. Plus, if you screw up—which is totally normal and happens to all developers on a daily basis—the simple act of correcting typos will help you learn how to debug your code.

Try to complete the review exercises and code challenges on your own before getting help from outside resources. With enough practice, you will master this material—and have fun along the way!

How Long Will It Take to Finish This Book?

If you're already familiar with a programming language you could finish the book in as little as 35 to 40 hours. If you're new to programming you may need to spend up to 100 hours or more. Take your time and don't feel like you have to rush. Programming is a super rewarding, but complex skill to learn. Good luck on your Python journey, we're rooting for you!

1.4 Bonus Material & Learning Resources

Online Resources

This book comes with a number of free bonus resources that you can access at realpython.com/python-basics/resources. On this web page you can also find an errata list with corrections maintained by the Real Python team.

Interactive Quizzes

Most chapters in this book come with a free online quiz to check your learning progress. You can access the quizzes using the links provided at the end of the chapter. The quizzes are hosted on the Real Python website and can be viewed on your phone or computer.

Each quiz takes you through a series of questions related to a particular chapter in the book. Some of them are multiple choice, some will ask you to type in an answer, and some will require you to write actual Python code. As you make your way through each quiz, it keeps score of which questions you answered correctly.

At the end of the quiz you receive a grade based on your result. If you don't score 100% on your first try—don't fret! These quizzes are meant to challenge you and it's expected that you go through them several times, improving your score with each run.

Exercises Code Repository

This book has an accompanying code repository on the web containing example source code as well as the answers to exercises and code challenges. The repository is broken up by chapter so you can check your code against the solutions provided by us after you finish each chapter. Here's the link:

realpython.com/python-basics/exercises

Example Code License

Note

The code found in this book has been tested with Python 3.8 on Windows, macOS, and Linux.

Formatting Conventions

Code blocks will be used to present example code:

```
# This is Python code:
print("Hello world!")
```

Terminal commands follow the Unix format:

```
$ # This is a terminal command:
$ python hello-world.py
```

(Dollar signs are not part of the command.)

Italic text will be used to denote a file name: *hello-world.py*.

Bold text will be used to denote a new or important term.

Keyboard shortcuts will be formatted as follows: Ctrl + S.

Menu shortcuts will be formatted as follows: File > New File

Notes and Warning boxes appear as follows:

Note

This is a note filled in with placeholder text. The quick brown fox jumps over the lazy dog. The quick brown Python slithers over the lazy hog.

Important

This is a warning also filled in with placeholder text. The quick brown fox jumps over the lazy dog. The quick brown Python slithers over the lazy hog.

Feedback & Errata

We welcome ideas, suggestions, feedback, and the occasional rant. Did you find a topic confusing? Did you find an error in the text or code? Did we leave out a topic you would love to know more about?

We're always looking to improve our teaching materials. Whatever the reason, please send in your feedback at the link below:

realpython.com/python-basics/feedback

Chapter 2

Setting Up Python

This book is about programming computers with Python. You could read this book cover-to-cover and absorb the information without ever touching a keyboard, but you'd miss out on the fun part—coding.

To get the most out of this book, you need to have a computer with Python installed on it and a way to create, edit, and save Python code files.

In this chapter, you will learn how to:

- Install the latest version of Python 3 on your computer
- Open **IDLE**, Python's built-in **I**ntegrated **D**evelopment and **L**earning **E**nvironment

Let's get started!

2.1 A Note On Python Versions

Many operating systems, such as macOS and Linux, come with Python pre-installed. The version of Python that comes with your operating system is called your **system Python**.

The system Python is almost always out-of-date and may not even be a full Python installation. It's essential that you have the most recent version of Python so that you can follow along successfully with the examples in this book.

It's possible to have multiple versions of Python installed on your computer. In this chapter, you'll install the latest version of Python 3 alongside any system Python that may already exist on your machine.

Note

Even if you already have Python 3.8 installed, it is still a good idea to skim this chapter to double check that your environment is set-up for following along with this book.

This chapter is split into three sections: Windows, macOS, and Ubuntu Linux. Find the section for your operating system and follow the steps to get set-up, then skip ahead to the next chapter.

If you have a different operating system, check out Real Python's Python 3 Installation & Setup Guide to see if your OS is covered.

2.2 Windows

Follow these steps to install Python 3 and open IDLE on Windows.

Important

The code in this book is only tested against Python installed as described in this section.

Be aware that if you have installed Python through some other means, such as Anaconda Python, you may encounter problems when running the code examples.

Install Python

Windows systems do not typically ship with Python pre-installed. Fortunately, installation does not involve much more than downloading the Python installer from the python.org website and running it.

Step 1: Download the Python 3 Installer

Open a browser window and navigate to the download page for Windows at python.org.

Underneath the heading at the top that says *Python Releases for Windows*, click on the link for the *Latest Python 3 Release - Python 3.x.x*. As of this writing, the latest version is Python 3.8. Then scroll to the bottom and select *Windows x86-64 executable installer*.

Note

If your system has a 32-bit processor, then you should choose the 32-bit installer. If you aren't sure if your computer is 32-bit or 64-bit, stick with the 64-bit installer mentioned above.

Step 2: Run the Installer

Run the installer by double-clicking on the downloaded file. You should see the following window:

Important

Make sure you check the box that says *Add Python 3.x to PATH* as shown to ensure that the install places the interpreter in your execution path.

If you install Python without checking this box, you can run the installer again and select it.

Click Install Now to install Python 3. Wait for the installation to finish, and then continue to open IDLE.

Open IDLE

You can open IDLE in two steps:

1. Click on the start menu and locate the *Python 3.8* folder.
2. Open the folder and select *IDLE (Python 3.8)*.

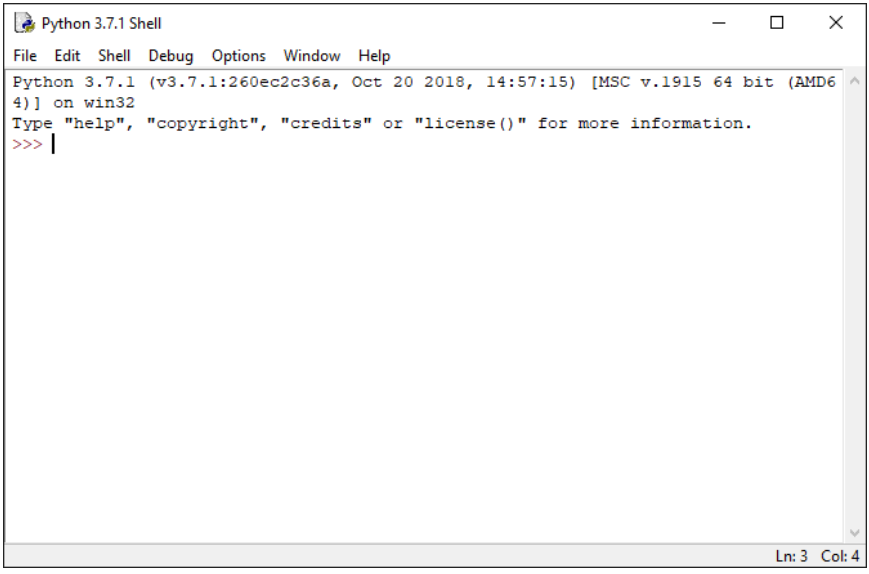
Note

We recommend using IDLE to follow along with this book.

You may use a different code editor if you prefer. However, some chapters, such as Chapter 7: *Finding And Fixing Code Bugs*, contain material specific to IDLE.

IDLE opens a **Python shell** in a new window. The Python shell is an interactive environment that allows you to type in some Python code and execute it immediately. It is a great way to get started with Python!

The Python shell window looks like this:



At the top of the window, you can see the version of Python that is running and some information about the operating system. If you

see a version less than 3.7, you may need to revisit the installation instructions in the previous section.

The >>> symbol that you see is called a **prompt**. Whenever you see this, it means that Python is waiting for you to give it some instructions.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/python-basics-2

Now that you have Python installed, let's get straight into writing your first Python program! Go ahead and move on to Chapter 3.

2.3 macOS

Follow these steps to install Python 3 and open IDLE on macOS.

Important

The code in this book is only tested against Python installed as described in this section.

Be aware that if you have installed Python through some other means, such as Homebrew or Anaconda Python, you may encounter problems when running the code examples.

Install Python

Most macOS machines come with Python 2 installed. You'll want to install the latest version of Python 3. You can do this by downloading an installer from the python.org website.

Step 1: Download the Python 3 Installer

Open a browser window and navigate to the download page for macOS at python.org.

Underneath the heading at the top that says *Python Releases for macOS*, click on the link for the *Latest Python 3 Release - Python 3.x.x*. As of this writing, the latest version is Python 3.8. Then scroll to the bottom of the page and select *macOS 64-bit/32-bit installer*. This starts the download.

Step 2: Run the Installer

Run the installer by double-clicking on the downloaded file. You should see the following window:

1. Press the Continue button a few times until you are asked to agree to the software license agreement. Then click Agree. You are shown a window that tells you where Python will be installed and how much space it will take.

2. You most likely don't want to change the default location, so go ahead and click Install to start the installation. The Python installer will tell you when it is finished copying files.
3. Click Close to close the installer window. Now that Python is installed, you can open up IDLE and get ready to write your first Python program.

Open IDLE

You can open IDLE in three steps:

1. Open Finder and click on *Applications*.
2. Locate the *Python 3.8* folder and double-click on it.
3. Double-click on the IDLE icon.

You may also open IDLE using the Spotlight search feature. Press Cmd + Spacebar to open the Spotlight search, type the word `idle`, then press Return to open IDLE.

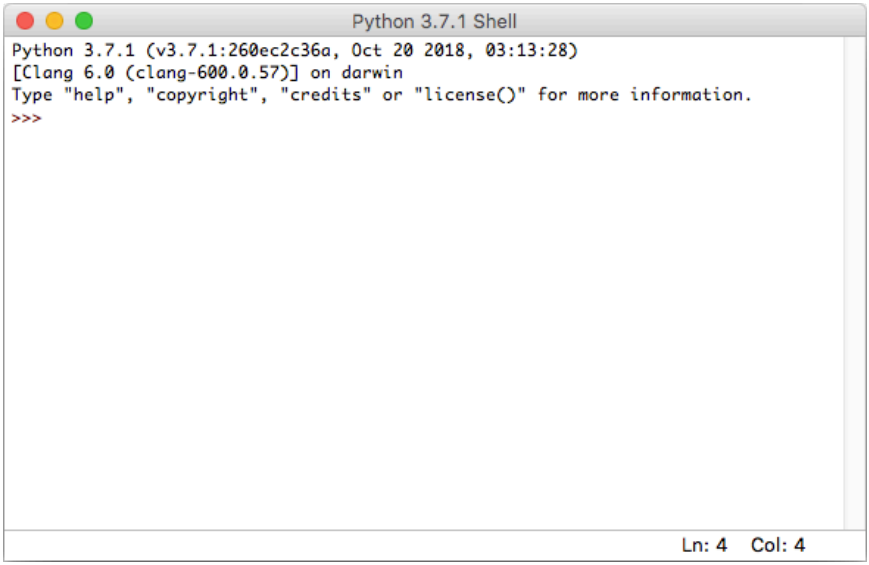
Note

We recommend using IDLE to follow along with this book.

You may use a different code editor if you prefer. However, some chapters, such as Chapter 7: *Finding And Fixing Code Bugs*, contain material specific to IDLE.

IDLE opens a **Python shell** in a new window. The Python shell is an interactive environment that allows you to type in some Python code and execute it immediately. It is a great way to get started with Python!

The Python shell window looks like this:



At the top of the window, you can see the version of Python that is running and some information about the operating system. If you see a version less than 3.7, you may need to revisit the installation instructions in the previous section.

The >>> symbol that you see is called a **prompt**. Whenever you see this, it means that Python is waiting for you to give it some instructions.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/python-basics-2

Now that you have Python installed, let's get straight into writing your first Python program! Go ahead and move on to Chapter 3.

2.4 Ubuntu Linux

Follow these steps to install Python 3 and open IDLE on Ubuntu Linux.

Important

The code in this book is only tested against Python installed as described in this section.

Be aware that if you have installed Python through some other means, such as Anaconda Python, you may encounter problems when running the code examples.

Install Python

There is a good chance your Ubuntu distribution has Python installed already, but it probably won't be the latest version, and it may be Python 2 instead of Python 3.

To find out what version(s) you have, open a terminal window and try the following commands:

```
$ python --version
$ python3 --version
```

One or more of these commands should respond with a version, as below (your version number may vary):

```
$ python3 --version
Python 3.8.1
```

If the version shown is Python 2.x or a version of Python 3 that is less than 3.8, then you want to install the latest version. How you install Python on Ubuntu depends on which version of Ubuntu you are running. You can determine your local Ubuntu version by running the following command:

```
$ lsb_release -a
No LSB modules are available.
Distributor ID: Ubuntu
Description:    Ubuntu 18.04.1 LTS
Release:        18.04
Codename:       bionic
```

Look at the version number next to `Release` in the console output, and follow the corresponding instructions below:

- **Ubuntu 18.04+** does not come with Python 3.8 by default, but it is in the Universe repository. You should be able to install it with the following commands:

  ```
  $ sudo apt-get update
  $ sudo apt-get install python3.8 idle-python3.8
  ```

- If you are using **Ubuntu 17 and lower**, Python 3.8 is not in the Universe repository, and you need to get it from a Personal Package Archive (PPA). To install Python from the "deadsnakes" PPA, do the following:

  ```
  $ sudo add-apt-repository ppa:deadsnakes/ppa
  $ sudo apt-get update
  $ sudo apt-get install python3.8 idle-python3.8
  ```

You can check that the correct version of Python was installed by running `python3 --version`. If you see a version number less than `3.7`, you may need to type `python3.8 --version`. Now you are ready to open IDLE and get ready to write your first Python program.

Open IDLE

You can open IDLE from the command line by typing the following:

```
$ idle-python3.8
```

On some Linux installations, you can open IDLE with the following shortened command:

```
$ idle3
```

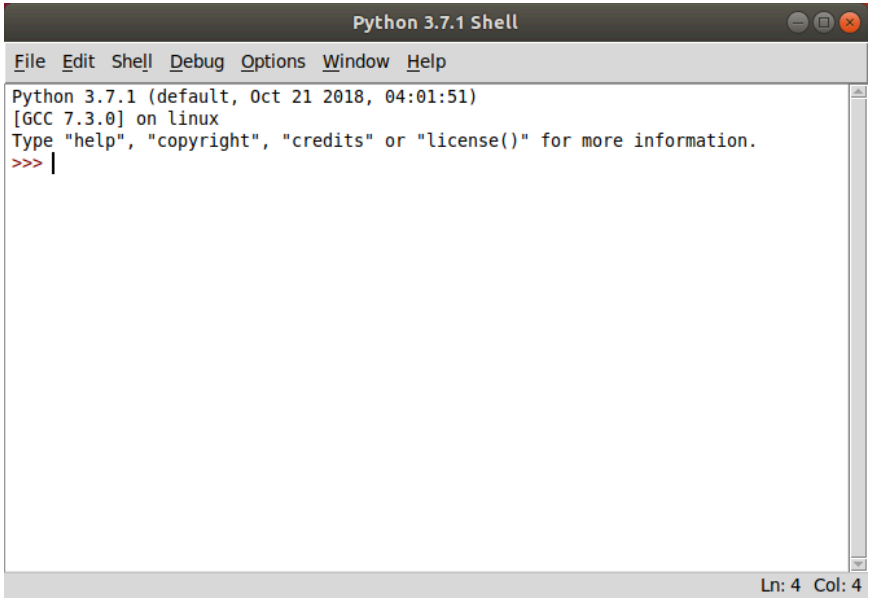
Note

We recommend using IDLE to follow along with this book.

You may use a different code editor if you prefer. However, some chapters, such as Chapter 7: *Finding And Fixing Code Bugs*, contain material specific to IDLE.

IDLE opens a **Python shell** in a new window. The Python shell is an interactive environment that allows you to type in some Python code and execute it immediately. It is a great way to get started with Python!

The Python shell window looks like this:



At the top of the window, you can see the version of Python that is running and some information about the operating system. If you

see a version less than 3.7, you may need to revisit the installation instructions in the previous section.

Important

If you opened IDLE with the `idle3` command and see a version less than 3.7 displayed in the Python shell window, then you will need to open IDLE with the `idle-python3.7` command.

The `>>>` symbol that you see in the IDLE window is called a **prompt**. Whenever you see this, it means that Python is waiting for you to give it some instructions.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/python-basics-2

Now that you have Python installed, let's get straight into writing your first Python program! Go ahead and move on to Chapter 3.

Chapter 3

Your First Python Program

Now that you have the latest version of Python installed on your computer, it's time to start coding!

In this chapter, you will:

- Write your first Python script
- Learn what happens when you run a script with an error
- Learn how to declare a variable and inspect its value
- Learn how to write comments

Ready to begin your Python journey? Let's go!

3.1 Write a Python Script

If you don't have IDLE open already, go ahead and open it. There are two main windows that you will work with in IDLE: the **interactive window**, which is the one that opens when you start IDLE, and the **script window**.

You can type code into both the interactive and script windows. The difference between the two is how the code is executed. In this section, you will write your first Python program and learn how to run it in

both windows.

The Interactive Window

The interactive window contains a **Python shell**, which is a textual user interface used to interact with the Python language. Hence the name "interactive window."

When you first open IDLE, the text displayed looks something like this:

```
Python 3.8.1 (tags/v3.8.1:1b293b6, Dec 18 2019, 22:39:24)
[MSC v.1916 32 bit (Intel)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>>
```

The first line tells you what version of Python is running. In this case, IDLE is running Python 3.8.1. The second and third lines give some information about the operating system and some commands you can use to get more information about Python.

The `>>>` symbol in the last line is called the **prompt**. This is where you will type in your code. Go ahead and type `1 + 1` at the prompt and press `Enter`.

When you hit Enter, Python evaluates the expression, displays the result `2`, and then prompts you for more input:

```
>>> 1 + 1
2
>>>
```

Notice that the Python prompt `>>>` appears again after your result. Python is ready for more instructions! Every time you run some code, a new prompt appears directly below the output.

The sequence of events in the interactive window can be described as a loop with three steps:

1. First, Python reads the code entered at the prompt.
2. Then the code is evaluated.
3. Finally, the output is printed in the window and a new prompt is displayed.

This loop is commonly referred to as a **R**ead-**E**valuate-**P**rint **L**oop, or **REPL**. Python programmers sometimes refer the Python shell as a "Python REPL", or just "the REPL" for short.

Note

From this point on, the final `>>>` prompt displayed after executing code in the interactive window is excluded from code examples.

Let's try something a little more interesting than adding two numbers. A rite of passage for every programmer is writing their first "Hello, world" program that prints the phrase "Hello, world" on the screen.

To print text to the screen in Python, you use the `print()` function. A **function** is a bit of code that typically takes some input, called an **argument**, does something with that input, and produces some output, called the **return value**.

Loosely speaking, functions in code work like mathematical functions. For example, the mathematical function $A(r)=\pi r^2$ takes the radius `r` of a circle as input and produces the area of the circle as output.

Important

The analogy to mathematical functions has some problems, though, because code functions can have **side effects**. A side effect occurs anytime a function performs some operation that changes something about the program or the computer running the program.

For example, you can write a function in Python that takes someone's name as input, stores the name in a file on the computer, and then outputs the path to the file with the name in it. The operation of saving the name to a file is a side effect of the function.

You'll learn more about functions, including how to write your own, in Chapter 6.

Python's `print()` function takes some text as input and then displays that text on the screen. To use `print()`, type the word `print` at the prompt in the interactive window, followed by the text `"Hello, world"` inside of parentheses:

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

Here `"Hello, world"` is the argument that is being **passed** to `print()`. `"Hello, world"` must be written with quotation marks so that Python interprets it as text and not something else.

Note

As you type code into the interactive window, you may notice that the font color changes for certain parts of the code. IDLE **highlights** parts of your code in different colors to help make it easier for you to identify what the different parts are.

By default, built-in functions, such as `print()` are displayed in purple, and text is displayed in green.

The interactive window can execute only a single line of code at a time. This is useful for trying out small code examples and exploring the Python language, but it has a major limitation. Code must be entered in by a person one line at a time!

Alternatively, you can store some Python code in a text file and then execute all of the code in the file with a single command. The code in the file is called a **script**, and files containing Python scripts are called **script files**.

Script files are nice not only because they make it easier to run a program, but also because they can be shared with other people so that they can run your program, too.

The Script Window

Scripts are written using IDLE's script window. You can open the script window by selecting File > New File from the menu at the top of the interactive window.

Notice that when the script window opens, the interactive window stays open. Any output generated by code run in the script window is displayed in the interactive window, so you may want to rearrange the two windows so that you can see both of them at the same time.

In the script window, type in the same code you used to print `"Hello, world"` in the interactive window:

```
print("Hello, world")
```

Just like the interactive window, code typed into the script window is highlighted.

Important

When you write code in a script, you do not need to include the `>>>` prompt that you see in IDLE's interactive window. Keep this in mind if you copy and paste code from examples that show the REPL prompt.

Remember, though, that it's not recommended that you copy and paste examples from the book. Typing each example in yourself really pays off!

Before you can run your script, you must save it. From the menu at the top of the window, select File > Save and save the script as `hello_world.py`. The `.py` file extension is the conventional extension used to indicate that a file contains Python code.

In fact, if you save your script with any extension other than `.py`, the code highlighting will disappear and all the text in the file will be displayed in black. IDLE will only highlight Python code when it is stored in a `.py` file.

Once the script is saved, all you have to do to run the program is select Run > Run Module from the script window and you'll see `Hello, world` appear in the interactive window:

```
Hello, world
```

Note

You can also press F5 to run a script from the script window.

Every time you run a script you will see something like the following output in the interactive window:

```
>>> ========== RESTART ==========
```

This is IDLE's way of separating output from distinct runs of a script. Otherwise, if you run one script after another, it may not be clear what

output belongs to which script.

To open an existing script in IDLE, select File » Open... from the menu in either the script window or the interactive window. Then browse for and select the script file you want to open. IDLE opens scripts in a new script window, so you can have several scripts open at a time.

Note

Double-clicking on a `.py` file from a file manager, such as Windows Explorer, does execute the script in a new window. However, the window is closed immediately when the script is done running—often before you can even see what happened.

To open the file in IDLE so that you can run it and see the output, you can right-click on the file icon (Ctrl + Click on macOS) and choose to Edit with IDLE.

3.2 Mess Things Up

Everybody makes mistakes—especially while programming! In case you haven't made any mistakes yet, let's get a head start on that and mess something up on purpose to see what happens.

Mistakes made in a program are called **errors**, and there are two main types of errors you'll experience:

1. Syntax errors
2. Run-time errors

In this section you'll see some examples of code errors and learn how to use the output Python displays when an error occurs to understand what error occurred and which piece of code caused it.

Syntax Errors

In loose terms, a **syntax error** occurs when you write some code that isn't allowed in the Python language. You can create a syntax error by changing the contents of the `hello_world.py` script from the last section to the following:

```
print("Hello, world)
```

In this example, the double quotation mark at the end of `"Hello, world"` has been removed. Python won't be able to tell where the string of text ends. Save the altered script and then try to run it. What happens?

The code won't run! IDLE displays an alert box with the following message:

```
EOL while scanning string literal.
```

EOL stands for **E**nd **O**f **L**ine, so this message tells you that Python read all the way to the end of the line without finding the end of something called a string literal.

A **string literal** is text contained in-between two double quotation marks. The text `"Hello, world"` is an example of a string literal.

Note

For brevity, string literals are often referred to as **strings**, although the term "string" technically has a more general meaning in Python. You will learn more about strings in Chapter 4.

Back in the script window, notice that the line containing with `"Hello, world` is highlighted in red. This handy features helps you quickly find which line of code caused the syntax error.

Run-time Errors

IDLE catches syntax errors before a program starts running, but some errors can't be caught until a program is executed. These errors are

known as **run-time errors** because they only occur at the time that a program is run.

To generate a run-time error, change the code in `hello_world.py` to the following:

```
print(Hello, world)
```

Now both quotation marks from the phrase `"Hello, world"` have been removed. Did you notice how the text color changes to black when you removed the quotation marks? IDLE no longer recognizes `Hello, world` as a string.

What do you think happens when you run the script? Try it out and see!

Some red text is displayed in the interactive window:

```
Traceback (most recent call last):
  File "/home/hello_world.py", line 1, in <module>
    print(Hello, world)
NameError: name 'Hello' is not defined
```

What happened? While trying to execute the program Python **raised** an error. Whenever an error occurs, Python stops executing the program and displays the error in IDLE's interactive window.

The text that gets displayed for an error is called a **traceback**. Tracebacks give you some useful information about the error. The traceback above tells us all of the following:

- The error happened on line `1` of the `hello_world.py`.
- The line that generated the error was: `print(Hello, world)`.
- A `NameError` occurred.
- The specific error was `name 'Hello' is not defined`

The quotation marks around `Hello, world` are missing, so Python doesn't understand that it is a string of text. Instead, Python thinks

that `Hello` and `world` are the names of something else in the code. Since names `Hello` and `world` haven't been defined anywhere, the program crashes.

In the next section, you'll see how to define names for values in your code. Before you move on though, you can get some practice with syntax errors and run-time errors by working on the review exercises.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Write a script that IDLE won't let you run because it has a syntax error.
2. Write a script that only crashes your program once it is already running because it has a run-time error.

3.3 Create a Variable

In Python, **variables** are names that can be assigned a value and used to reference that value throughout your code. Variables are fundamental to programming for two reasons:

1. **Variables keep values accessible:** For example, the result of some time-consuming operation can be assigned to a variable so that the operation does not need to be performed each time you need to use the result.
2. **Variables give values context:** The number `28` could mean lots of different things, such as the number of students in a class, or the number of times a user has accessed a website, and so on. Naming the value `28` something like `num_students` makes the meaning of the value clear.

In this section, you'll learn how to use variables in your code, as well as some of the conventions Python programmers follow when choosing

names for variables.

The Assignment Operator

Values are assigned to a variable using a special symbol `=` called the **assignment operator**. An **operator** is a symbol, like `=` or `+`, that performs some operation on one or more values.

For example, the `+` operator takes two numbers, one to the left of the operator and one to the right, and adds them together. Likewise, the `=` operator takes a value to the right of the operator and assigns it to the name on the left of the operator.

To see the assignment operator in action, let's modify the "Hello, world" program you saw in the last section. This time, we'll use a variable to store some text before printing it to the screen:

```
>>> phrase = "Hello, world"
>>> print(phrase)
Hello, world
```

In the first line, a variable named `phrase` is created and assigned the value `"Hello, world"` using the `=` operator. The string `"Hello, world"` that was originally used inside of the parentheses in the `print()` function is replaced with the variable `phrase`.

The output `Hello, world` is displayed when you execute `print(phrase)` because Python looks up the name `phrase` and finds it has been assigned the value `"Hello, world"`.

If you hadn't executed `phrase = "Hello, world"` before executing `print(phrase)`, you would have seen a `NameError` like you did when trying to execute `print(Hello, world)` in the previous section.

Note

Although = looks like the equals sign from mathematics, it has a different meaning in Python. Distinguishing the = operator from the equals sign is important, and can be a source of frustration for beginner programmers.

Just remember, whenever you see the = operator, whatever is to the right of it is being assigned to a variable on the left.

Variable names are **case-sensitive**, so a variable named `phrase` is distinct from a variable named `Phrase` (note the capital `P`). For instance, the following code produces a `NameError`:

```
>>> phrase = "Hello, world"
>>> print(Phrase)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'Phrase' is not defined
```

When you run into trouble with the code examples in this book, be sure to double-check that every character in your code—including spaces—exactly matches the examples. Computers can't use common sense to interpret what you meant to say, so being *almost* correct won't get a computer to do the right thing!

Rules for Valid Variable Names

Variable names can be as long or as short as you like, but there are a couple of rules that you must follow. Variable names can only contain uppercase and lowercase letters (A–Z, a–z), digits (0–9), and underscores (_). However, variable names cannot begin with a digit.

For example, `phrase`, `string1`, `_a1p4a`, and `list_of_names` are all valid variable names, but `9lives` is not.

Note

Python variable names can contain many different valid Unicode characters. **Unicode** is a standard for digitally representing text used in most of the world's writing systems.

That means variable names can contain letters from non-English alphabets, such as decorated letters like é and ü, and even Chinese, Japanese, and Arabic symbols.

However, not every system can display decorated characters, so it is a good idea to avoid them if your code is going to be shared with people in many different regions.

You can learn more about Unicode on Wikipedia. Python's support for Unicode is covered in the official Python documentation.

Just because a variable name is valid doesn't necessarily mean that it is a good name. Choosing a good name for a variable can be surprisingly difficult. However, there are some guidelines that you can follow to help you choose better names.

Descriptive Names Are Better Than Short Names

Descriptive variable names are essential, especially for complex programs. Often, descriptive names require using multiple words. Don't be afraid to use long variable names.

In the following example, the value `3600` is assigned to the variable `s`:

```
s = 3600
```

The name `s` is totally ambiguous. Using a full word makes it a lot easier to understand what the code means:

```
seconds = 3600
```

`seconds` is a better name than `s` because it provides more context. But

it still doesn't convey the full meaning of the code. Is `3600` the number of seconds it takes for some process to finish, or the length of a movie? There's no way to tell.

The following name leaves no doubt about what the code means:

```
seconds_per_hour = 3600
```

When you read the above code, there is no question that `3600` is the number of seconds in one hour. Although `seconds_per_hour` takes longer to type than both the single letter `s` and the word `seconds`, the pay-off in clarity is massive.

Although naming variables descriptively means using longer variable names, you should avoid names that are excessively long. What "excessively long" really means is subjective, but a good rule of thumb is to keep variable names to fewer than three or four words.

Python Variable Naming Conventions

In many programming languages, it is common to write variable names in **camelCase** like `numStudents` and `listOfNames`. The first letter of every word, except the first, is capitalized, and all other letters are lowercase. The juxtaposition of lower-case and upper-case letters look like humps on a camel.

In Python, however, it is more common to write variable names in **snake case** like `num_students` and `list_of_names`. Every letter is lowercase, and each word is separated by an underscore.

While there is no hard-and-fast rule mandating that you write your variable names in snake case, the practice is codified in a document called PEP 8, which is widely regarded as the official style guide for writing Python.

Following the standards outlined in PEP 8 ensures that your Python code is readable by a large number of Python programmers. This makes sharing and collaborating on code easier for everyone involved.

Note

All of the code examples in this course follow PEP 8 guidelines, so you will get a lot of exposure to what Python code that follows standard formatting guidelines looks like.

In this section you learned how to create a variable, rules for valid variable names, and some guidelines for choosing good variable names. Next, you will learn how to inspect a variable's value in IDLE's interactive window.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Using the interactive window, display some text on the screen by using the `print()` function.
2. Using the interactive window, display a string of text by saving the string to a variable, then reference the string in a `print()` function using the variable name.
3. Do each of the first two exercises again by first saving your code in a script and running it.

3.4 Inspect Values in the Interactive Window

You have already seen how to use `print()` to display a string that has been assigned to a variable. There is another way to display the value of a variable when you are working in the Python shell.

Type the following into IDLE's interactive window:

```
>>> phrase = "Hello, world"
>>> phrase
```

When you press `Enter` after typing `phrase` a second time, the following output is displayed:

```
'Hello, world'
```

Python prints the string `"Hello, world"`, and you didn't have to type `print(phrase)`!

Now type the following:

```
>>> print(phrase)
```

This time, when you hit `Enter` you see:

```
Hello, world
```

Do you see the difference between this output and the output of simply typing `phrase`? It doesn't have any single quotes surrounding it. What's going on here?

When you type `phrase` and press `Enter`, you are telling Python to **inspect** the variable `phrase`. The output displayed is a useful representation of the value assigned to the variable.

In this case, `phrase` is assigned the string `"Hello, world"`, so the output is surrounded with single quotes to indicate that `phrase` is a string.

On the other hand, when you `print()` a variable, Python displays a more human-readable representation of the variable's value. For strings, both ways of being displayed are human-readable, but this is not the case for every type of value.

Sometimes, both printing and inspecting a variable produces the same output:

```
>>> x = 2
>>> x
2
>>> print(x)
```

```
2
```

Here, `x` is assigned to the number `2`. Both the output of `print(x)` and inspecting `x` is not surrounded with quotes, because `2` is a number and not a string.

Inspecting a variable, instead of printing it, is useful for a couple of reasons. You can use it to display the value of a variable without typing `print()`. More importantly, though, inspecting a variable usually gives you more useful information than `print()` does.

Suppose you have two variables: `x = 2` and `y = "2"`. In this case, `print(x)` and `print(y)` both display the same thing. However, inspecting `x` and `y` shows the difference between the each variable's value:

```
>>> x = 2
>>> y = "2"
>>> print(x)
2
>>> print(y)
2
>>> x
2
>>> y
'2'
```

The key takeaway here is that `print()` displays a readable representation of a variable's value, while inspection provides additional information about the type of the value.

You can inspect more than just variables in the Python shell. Check out what happens when you type `print` and hit `Enter`:

```
>>> print
<built-in function print>
```

Keep in mind that you can only inspect variables in a Python shell. For example, save and run the following script:

```
phrase = "Hello, world"
phrase
```

The script executes without any errors, but no output is displayed! Throughout this book, you will see examples that use the interactive window to inspect variables.

3.5 Leave Yourself Helpful Notes

Programmers often read code they wrote several months ago and wonder "What the heck does this do?" Even with descriptive variable names, it can be difficult to remember why you wrote something the way you did when you haven't looked at it for a long time.

To help avoid this problem, you can leave comments in your code. **Comments** are lines of text that don't affect the way the script runs. They help to document what's supposed to be happening.

In this section, you will learn three ways to leave comments in your code. You will also learn some conventions for formatting comments, as well as some pet peeves regarding their over-use.

How to Write a Comment

The most common way to write a comment is to begin a new line in your code with the `#` character. When your code is run, any lines starting with `#` are ignored. Comments that start on a new line are called **block comments**.

You can also write **in-line comments**, which are comments that appear on the same line as some code. Just put a `#` at the end of the line of code, followed by the text in your comment.

Here is an example of the `hello_world.py` script with both kinds of comments added in:

```
# This is a block comment.

phrase = "Hello, world."
print(phrase)  # This is an in-line comment.
```

The first line doesn't do anything, because it starts with a `#`. Likewise, `print(phrase)` is executed on the last line, but everything after the `#` is ignored.

Of course, you can still use the `#` symbol inside of a string. For instance, Python won't mistake the following for the start of a comment:

```
print("#1")
```

In general, it's a good idea to keep comments as short as possible, but sometimes you need to write more than will reasonably fit on a single line. In that case, you can continue your comment on a new line that also begins with a `#` symbol:

```
# This is my first script.
# It prints the phrase "Hello, world."
# The comments are longer than the script!

phrase = "Hello, world."
print(phrase)
```

Besides leaving yourself notes, comments can also be used to **comment out** code while you're testing a program. In other words, adding a `#` at the beginning of a line of code lets you run your program as if that line of code didn't exist without having to delete any code.

To comment out a section of code in IDLE highlight one ore more lines to be commented and press:

- **Windows:** Alt + 3
- **macOS**: Ctrl + 3
- **Ubuntu Linux:**: Ctrl + D

Two # symbols are inserted at the beginning of each line. This doesn't follow PEP 8 comment formatting conventions, but it gets the job done!

To un-comment out your code and remove the # symbols from the beginning of each line, highlight the code that is commented out and press:

- **Windows:** Alt + 4
- **macOS:** Ctrl + 4
- **Ubuntu Linux:** Ctrl + Shift + D

Now let's look at some common conventions regarded code comments.

Conventions and Pet Peeves

According to PEP 8, comments should always be written in complete sentences with a single space between the # and the first word of the comment:

```
# This comment is formatted to PEP 8.

#don't do this
```

For in-line comments, PEP 8 recommends at least two spaces between the code and the # symbol:

```
phrase = "Hello, world"  # This comment is PEP 8 compliant.
print(phrase)# This comment isn't.
```

A major pet peeve among programmers are comments that describe what is already obvious from reading the code. For example, the following comment is unnecessary:

```
# Print "Hello, world"
print("Hello, world")
```

No comment is needed in this example because the code itself explicitly describes what is being done. Comments are best used to clarify code that may not be easy to understand, or to explain why something is done a certain way.

In general, PEP 8 recommends that comments be used sparingly. Use comments only when they add value to your code by making it easier to understand *why* something is done a certain way. Comments that describe *what* something does can often be avoided by using more descriptive variable names.

3.6 Summary and Additional Resources

In this chapter, you wrote and executed your first Python program! You wrote a small program that displays the text `"Hello, world"` using the `print()` function.

You were introduced to three concepts:

1. **Variables** give names to values in your code using the assignment operator (`=`)
2. **Errors**, such as syntax errors and run-time errors, are raised whenever Python can't execute your code. They are displayed in IDLE's interactive window in the form of a traceback.
3. **Comments** are lines of code that don't get executed and serve as documentation for yourself and other programmers that need to read your code.

Interactive Quiz

This chapter comes with a free online quiz to check your learning progress. You can access the quiz using your phone or computer at the following web address:

realpython.com/quizzes/python-basics-3

Additional Resources

To learn more, check out the following resources:

- 11 Beginner Tips for Learning Python Programming
- Writing Comments in Python (Guide)
- Recommended resources on realpython.com

Chapter 4

Strings and String Methods

Many programmers, regardless of their specialty, deal with text on a daily basis. For example, web developers work with text that gets input from web forms. Data scientists process text to extract data and perform things like sentiment analysis, which can help identify and classify opinions in a body of text.

Collections of text in Python are called **strings**. Special functions called **string methods** are used to manipulate strings. There are string methods for changing a string from lowercase to uppercase, removing whitespace from the beginning or end of a string, or replacing parts of a string with different text, and many more.

In this chapter, you will learn how to:

- Manipulate strings with string methods
- Work with user input
- Deal with strings of numbers
- Format strings for printing

Let's get started!

4.1 What is a String?

In Chapter 3, you created the string `"Hello, world"` and printed it in IDLE's interactive window using the `print()` function. In this section, you'll get a deeper look into what exactly a string is and the various ways you can create them in Python.

The String Data Type

Strings are one of the fundamental Python data types. The term **data type** refers to what kind of data a value represents. Strings are used to represent text.

Note

There are several other data types built-in to Python. For example, you'll learn about numerical data types in Chapter 5, and Boolean types in Chapter 8.

We say that strings are a **fundamental** data type because they can't be broken down into smaller values of a different type. Not all data types are fundamental. You'll learn about compound data types, also known as **data structures**, in Chapter 9.

The string data type has a special abbreviated name in Python: `str`. You can see this by using the `type()` function, which is used to determine the data type of a given value.

Type the following into IDLE's interactive window:

```
>>> type("Hello, world")
<class 'str'>
```

The output `<class 'str'>` indicates that the value `"Hello, world"` is an instance of the `str` data type. That is, `"Hello, world"` is a string.

Note

For now, you can think of the word "class" as a synonym for "data type," although it actually refers to something more specific. You'll see just what a class is in Chapter 10.

`type()` also works for values that have been assigned to a variable:

```
>>> phrase = "Hello, world"
>>> type(phrase)
<class 'str'>
```

Strings have three properties that you'll explore in the coming sections:

1. Strings contains **characters**, which are individual letters or symbols.
2. Strings have a **length**, which is the number of characters contained in the string.
3. Characters in a string appear in a **sequence**, meaning each character has a numbered position in the string.

Let's take a closer look at how strings are created.

String Literals

As you've already seen, you can create a string by surrounding some text with quotation marks:

```
string1 = 'Hello, world'
string2 = "1234"
```

Either single quotes (`string1`) or double quotes (`string2`) can be used to create a string, as long as both quotation marks are the same type.

Whenever you create a string by surrounding text with quotation marks, the string is called a **string literal**. The name indicates that the string is literally written out in your code. All of the strings you

have seen thus far are string literals.

Note

Not every string is a string literal. For example, a string captured as user input isn't a string literal because it isn't explicitly written out in the program's code.

You'll learn how to work with user input in section 4 of this chapter.

The quotes surrounding a string are called **delimiters** because they tell Python where a string begins and where it ends. When one type of quotes is used as the delimiter, the other type of quote can be used inside of the string:

```
string3 = "We're #1!"
string4 = 'I said, "Put it over by the llama."'
```

After Python reads the first delimiter, all of the characters after it are considered a part of the string until a second matching delimiter is read. This is why you can use a single quote in a string delimited by double quotes and vice versa.

If you try to use double quotes inside of a string that is delimited by double quotes, you will get an error:

```
>>> text = "She said, "What time is it?""
  File "<stdin>", line 1
    text = "She said, "What time is it?""
                       ^
SyntaxError: invalid syntax
```

Python throws a `SyntaxError` because it thinks that the string ends after the second `"` and doesn't know how to interpret the rest of the line.

Note

A common pet peeve among programmers is the use of mixed quotes as delimiters. When you work on a project, it's a good idea to use only single quotes or only double quotes to delimit every string.

Keep in mind that there isn't really a right or wrong choice! The goal is to be consistent, because consistency helps make your code easier to read and understand.

Strings can contain any valid Unicode character. For example, the string `"We're #1!"` contains the pound sign (`#`) and `"1234"` contains numbers. `"×Pýthøn×"` is also a valid Python string!

Determine the Length of a String

The number of characters contained in a string, including spaces, is called the **length** of the string. For example, the string `"abc"` has a length of `3`, and the string `"Don't Panic"` has a length of `11`.

To determine a string's length, you use Python's built-in `len()` function. To see how it works, type the following into IDLE's interactive window:

```
>>> len("abc")
3
```

You can also use `len()` to get the length of a string that's assigned to a variable:

```
>>> letters = "abc"
>>> num_letters = len(letters)
>>> num_letters
3
```

First, the string `"abc"` is assigned to the variable `letters`. Then `len()` is used to get the length of `letters` and this value is assigned to the `num_letters` variable. Finally, the value of `num_letters`, which is `3`, is

displayed.

Multiline Strings

The PEP 8 style guide recommends that each line of Python code contain no more than 79 characters—including spaces.

Note

PEP 8's 79-character line-length is recommended because, among other things, it makes it easier to read two files side-by-side. However, many Python programmers believe forcing each line to be at most 79 characters sometimes makes code harder to read.

In this book we will strictly follow PEP 8's recommended line-length. Just know that you will encounter lots of code in in the real world with longer lines.

Whether you decide to follow PEP 8, or choose a larger number of characters for your line-length, you will sometimes need to create string literals with more characters than your chosen limit.

To deal with long strings, you can break the string up across multiple lines into a **multiline string**. For example, suppose you need to fit the following text into a string literal:

> "This planet has—or rather had—a problem, which was this: most of the people living on it were unhappy for pretty much of the time. Many solutions were suggested for this problem, but most of these were largely concerned with the movements of small green pieces of paper, which is odd because on the whole it wasn't the small green pieces of paper that were unhappy."
>
> — Douglas Adams, *The Hitchhiker's Guide to the Galaxy*

This paragraph contains far more than 79 characters, so any line of code containing the paragraph as a string literal violates PEP 8. So, what do you do?

There are a couple of ways to tackle this. One way is to break the string up across multiple lines and put a backslash (\) at the end of all but the last line. To be PEP 8 compliant, the total length of the line, including the backslash, must be 79 characters or less.

Here's how you could write the paragraph as a multiline string using the backslash method:

```
paragraph = "This planet has - or rather had - a problem, which was \
this: most of the people living on it were unhappy for pretty much \
of the time. Many solutions were suggested for this problem, but \
most of these were largely concerned with the movements of small \
green pieces of paper, which is odd because on the whole it wasn't \
the small green pieces of paper that were unhappy."
```

Notice that you don't have to close each line with a quotation mark. Normally, Python would get to the end of the first line and complain that you didn't close the string with a matching double quote. With a backslash at the end, however, you can keep writing the same string on the next line.

When you `print()` a multiline string that is broken up by backslashes, the output displayed on a single line:

```
>>> long_string = "This multiline string is \
displayed on one line"
>>> print(long_string)
This multiline string is displayed on one line
```

Multiline strings can also be created using triple quotes as delimiters (`"""` or `'''`). Here is how you might write a long paragraph using this approach:

```
paragraph = """This planet has - or rather had - a problem, which was
this: most of the people living on it were unhappy for pretty much
of the time. Many solutions were suggested for this problem, but
most of these were largely concerned with the movements of small
green pieces of paper, which is odd because on the whole it wasn't
the small green pieces of paper that were unhappy."""
```

Triple-quoted strings preserve whitespace. This means that running `print(paragraph)` displays the string on multiple lines just like it is in the string literal, including newlines. This may or may not be what you want, so you'll need to think about the desired output before you choose how to write a multiline string.

To see how whitespace is preserved in a triple-quoted string, type the following into IDLE's interactive window:

```
>>> print("""This is a
...     string that spans across multiple lines
...         and also preserves whitespace.""")
This is a
    string that spans across multiple lines
        and also preserves whitespace.
```

Notice how the second and third lines in the output are indented exactly the same way they are in the string literal.

Note

Triple-quoted strings have a special purpose in Python. They are used to document code. You'll often find them at the top of a `.py` with a description of the code's purpose. They are also used to document custom functions.

When used to document code, triple-quoted strings are called **docstrings**. You'll learn more about docstrings in Chapter 6.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Print a string that uses double quotation marks inside the string.
2. Print a string that uses an apostrophe inside the string.
3. Print a string that spans multiple lines, with whitespace preserved.
4. Print a string that is coded on multiple lines but displays on a single line.

4.2 Concatenation, Indexing, and Slicing

Now that you know what a string is and how to declare string literals in your code, let's explore some of the things you can do with strings.

In this section, you'll learn about three basic string operations:

1. Concatenation, which joins two strings together
2. Indexing, which gets a single character from a string
3. Slicing, which gets several characters from a string at once

Let's dive in!

String Concatenation

Two strings can be combined, or **concatenated**, using the `+` operator:

```
>>> string1 = "abra"
>>> string2 = "cadabra"
>>> magic_string = string1 + string2
>>> magic_string
'abracadabra'
```

In this example, string concatenation occurs on the third line. `string1` and `string2` are concatenated using `+` and the result is assigned to the variable `magic_string`. Notice that the two strings are joined without any whitespace between them.

You can use string concatenation to join two related strings, such as joining a first and last name into a full name:

```
>>> first_name = "Arthur"
>>> last_name = "Dent"
>>> full_name = first_name + " " + last_name
>>> full_name
'Arthur Dent'
```

Here string concatenation occurs twice on the same line. `first_name` is concatenated with `" "`, resulting in the string `"Arthur "`. Then this result is concatenated with `last_name` to produce the full name `"Arthur Dent"`.

String Indexing

Each character in a string has a numbered position called an **index**. You can access the character at the *Nth* position by putting the number *N* in between two square brackets (`[` and `]`) immediately after the string:

```
>>> flavor = "apple pie"
>>> flavor[1]
'p'
```

`flavor[1]` returns the character at position `1` in `"apple pie"`, which is `p`. Wait, isn't `a` the first character of `"apple pie"`?

In Python—and most other programming languages—counting always starts at zero. To get the character at the beginning of a string, you need to access the character at position `0`:

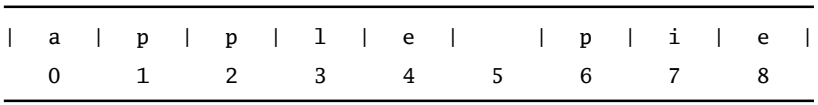
```
>>> flavor[0]
'a'
```

Note

Forgetting that counting starts with zero and trying to access the first character in a string with the index `1` results in an **off-by-one error**.

Off-by-one errors are a common source of frustration for both beginning and experienced programmers alike!

The following figure shows the index for each character of the string `"apple pie"`:



If you try to access an index beyond the end of a string, Python raises an `IndexError`:

```
>>> flavor[9]
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    flavor[9]
IndexError: string index out of range
```

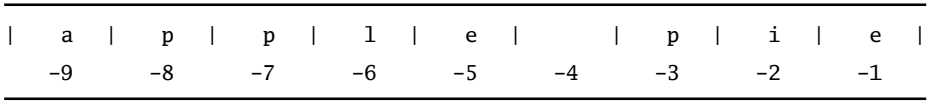
The largest index in a string is always one less than the string's length. Since `"apple pie"` has a length of nine, the largest index allowed is `8`.

Strings also support negative indices:

```
>>> flavor[-1]
'e'
```

The last character in a string has index `-1`, which for `"apple pie"` is the letter `e`. The second-to-last character `i` has index `-2`, and so on.

The following figure shows the negative index for each character in the string `"apple pie"`:



Just like positive indices, Python raises an `IndexError` if you try to access a negative index less than the index of the first character in the string:

```
>>> flavor[-10]
Traceback (most recent call last):
  File "<pyshell#5>", line 1, in <module>
    flavor[-10]
IndexError: string index out of range
```

Negative indices may not seem useful at first, but sometimes they are a better choice than a positive index.

For example, suppose a string input by a user is assigned to the variable `user_input`. If you need to get the last character of the string, how do you know what index to use?

One way to get the last character of a string is to calculate the final index using `len()`:

```
final_index = len(user_input) - 1
last_character = user_input[final_index]
```

Getting the final character with the index `-1` takes less typing and doesn't require an intermediate step to calculate the final index:

```
last_character = user_input[-1]
```

String Slicing

Suppose you need the string containing just the first three letters of the string `"apple pie"`. You could access each character by index and concatenate them, like this:

```
>>> first_three_letters = flavor[0] + flavor[1] + flavor[2]
>>> first_three_letters
'app'
```

If you need more than just the first few letters of a string, getting each character individually and concatenating them together is clumsy and long-winded. Fortunately, Python provides a way to do this with much less typing.

You can extract a portion of a string, called a **substring**, by inserting a colon between two index numbers inside of square brackets, like this:

```
>>> flavor = "apple pie"
>>> flavor[0:3]
'app'
```

`flavor[0:3]` returns the first three characters of the string assigned to `flavor`, starting with the character with index `0` and going up to, but not including, the character with index `3`. The `[0:3]` part of `flavor[0:3]` is called a **slice**. In this case, it returns a slice of `"apple pie"`. Yum!

String slices can be confusing because the substring returned by the slice includes the character whose index is the first number, but doesn't include the character whose index is the second number.

To remember how slicing works, you can think of a string as a sequence of square slots. The left and right boundary of each slot is numbered from zero up to the length of the string, and each slot is filled with a character in the string.

Here's what this looks like for the string `"apple pie"`:

```
 ---------------------------------------------
| a | p | p | l | e |   | p | i | e |
```

0	1	2	3	4	5	6	7	8	9

The slice `[x:y]` returns the substring between the boundaries `x` and `y`. So, for `"apple pie"`, the slice `[0:3]` returns the string `"app"`, and the slice `[3:9]` returns the string `"le pie"`.

If you omit the first index in a slice, Python assumes you want to start at index `0`:

```
>>> flavor[:5]
'apple'
```

The slice `[:5]` is equivalent to the slice `[0:5]`, so `flavor[:5]` returns the first five characters in the string `"apple pie`.

Similarly, if you omit the second index in the slice, Python assumes you want to return the substring that begins with the character whose index is the first number in the slice and ends with the last character in the string:

```
>>> flavor[5:]
' pie'
```

For `"apple pie"`, the slice `[5:]` is equivalent to the slice `[5:9]`. Since the character with index `5` is a space, `flavor[5:9]` returns the substring that starts with the space and ends with the last letter, which is `" pie"`.

If you omit both the first and second numbers in a slice, you get a string that starts with the character with index `0` and ends with the last character. In other words, omitting both numbers in a slice returns the entire string:

```
>>> flavor[:]
'apple pie'
```

It's important to note that, unlike string indexing, Python won't raise an `IndexError` when you try to slice between boundaries before or after

the beginning and ending boundaries of a string:

```
>>> flavor[:14]
'apple pie'
>>> flavor[13:15]
''
```

In this example, the first line gets the slice from the beginning of the string up to but not including the fourteenth character. The string assigned to `flavor` has length nine, so you might expect Python to throw an error. Instead, any non-existent indices are ignored and the entire string `"apple pie"` is returned.

The second shows what happens when you try to get a slice where the entire range is out of bounds. `flavor[13:15]` attempts to get the thirteenth and fourteenth characters, which don't exist. Instead of raising an error, the **empty string** `""` is returned.

You can use negative numbers in slices. The rules for slices with negative numbers are exactly the same as slices with positive numbers. It helps to visualize the string as slots with the boundaries labeled by negative numbers:

```
 |  a  |  p  |  p  |  l  |  e  |     |  p  |  i  |  e  |
-9    -8    -7    -6    -5    -4    -3    -2    -1
```

Just like before, the slice `[x:y]` returns the substring between the boundaries `x` and `y`. For instance, the slice `[-9:-6]` returns the first three letters of the string `"apple pie"`:

```
>>> flavor[-9:-6]
'app'
```

Notice, however, that the right-most boundary does not have a negative index. The logical choice for that boundary would seem to be the number `0`, but that doesn't work:

```
>>> flavor[-9:0]
''
```

Instead of returning the entire string, `[-9:0]` returns the **empty string** `""`. This is because the second number in a slice must correspond to a boundary that comes after the boundary corresponding to the first number, but both `-9` and `0` correspond to the left-most boundary in the figure.

If you need to include the final character of a string in your slice, you can omit the second number:

```
>>> flavor[-9:]
'apple pie'
```

Strings Are Immutable

To wrap this section up, let's discuss an important property of string objects. Strings are **immutable**, which means that you can't change them once you've created them. For instance, see what happens when you try to assign a new letter to one particular character of a string:

```
>>> word = "goal"
>>> word[0] = "f"
Traceback (most recent call last):
  File "<pyshell#16>", line 1, in <module>
    word[0] = "f"
TypeError: 'str' object does not support item assignment
```

Python throws a `TypeError` and tells you that `str` objects don't support item assignment.

Note

The term `str` is Python's internal name for the string data type.

If you want to alter a string, you must create an entirely new string. To change the string `"goal"` to the string `"foal"`, you can use a string

slice to concatenate the letter `"f"` with everything but the first letter of the word `"goal"`:

```
>>> word = "goal"
>>> word = "f" + word[1:]
>>> word
'foal'
```

First assign the string `"goal"` to the variable `word`. Then concatenate the slice `word[1:]`, which is the string `"oal"`, with the letter `"f"` to get the string `"foal"`. If you're getting a different result here, make sure you're including the `:` colon character as part of the string slice.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Create a string and print its length using the `len()` function.
2. Create two strings, concatenate them, and print the resulting string.
3. Create two strings and use concatenation to add a space in-between them. Then print the result.
4. Print the string `"zing"` by using slice notation on the string `"bazinga"` to specify the correct range of characters.

4.3 Manipulate Strings With Methods

Strings come bundled with special functions called **string methods** that can be used to work with and manipulate strings. There are numerous string methods available, but we'll focus on some of the most commonly used ones.

In this section, you will learn how to:

- Convert a string to upper or lower case

- Remove whitespace from string
- Determine if a string begins and ends with certain characters

Let's go!

Converting String Case

To convert a string to all lower case letters, you use the string's `.lower()` method. This is done by tacking `.lower()` on to the end of the string itself:

```
>>> "Jean-luc Picard".lower()
'jean-luc picard'
```

The dot (`.`) tells Python that what follows is the name of a method—the `lower()` method in this case.

Note

We will refer to the names of string methods with a dot at the beginning of them. So, for example, the `.lower()` method is written with a dot, instead of `lower()`.

The reason we do this is to make it easy to spot functions that are string methods, as opposed to built-in functions like `print()` and `type()`.

String methods don't just work on string literals. You can also use the `.lower()` method on a string assigned to a variable:

```
>>> name = "Jean-luc Picard"
>>> name.lower()
'jean-luc picard'
```

The opposite of the `.lower()` method is the `.upper()` method, which converts every character in a string to upper case:

```
>>> loud_voice = "Can you hear me yet?"
>>> loud_voice.upper()
'CAN YOU HEAR ME YET?'
```

Compare the `.upper()` and `.lower()` string methods to the general-purpose `len()` function you saw in the last section. Aside from the different results of these functions, the important distinction here is how they are used.

The `len()` function is a stand-alone function. If you want to determine the length of the `loud_voice` string, you call the `len()` function directly, like this:

```
>>> len(loud_voice)
20
```

On the other hand, `.upper()` and `.lower()` must be used in conjunction with a string. They do not exist independently.

Removing Whitespace From a String

Whitespace is any character that is printed as blank space. This includes things like spaces and **line feeds**, which are special characters that move output to a new line.

Sometimes you need to remove whitespace from the beginning or end of a string. This is especially useful when working with strings that come from user input, where extra whitespace characters may have been introduced by accident.

There are three string methods that you can use to remove whitespace from a string:

1. `.rstrip()`
2. `.lstrip()`
3. `.strip()`

`.rstrip()` removes whitespace from the right side of a string:

```
>>> name = "Jean-luc Picard     "
>>> name
'Jean-luc Picard     '
>>> name.rstrip()
'Jean-luc Picard'
```

In this example, the string `"Jean-luc Picard     "` has five trailing spaces. Python doesn't remove any trailing spaces in a string automatically when the string is assigned to a variable. The `.rstrip()` method removes trailing spaces from the right-hand side of the string and returns a new string `"Jean-luc Picard"`, which no longer has the spaces at the end.

The `.lstrip()` method works just like `.rstrip()`, except that it removes whitespace from the left-hand side of the string:

```
>>> name = "     Jean-luc Picard"
>>> name
'     Jean-luc Picard'
>>> name.lstrip()
'Jean-luc Picard'
```

To remove whitespace from both the left and the right sides of the string at the same time, use the `.strip()` method:

```
>>> name = "     Jean-luc Picard     "
>>> name
'     Jean-luc Picard     '
>>> name.strip()
'Jean-luc Picard'
```

Note

None of the `.rstrip()`, `.lstrip()`, and `.strip()` methods remove whitespace from the middle of the string. In each of the previous examples the space between "Jean-luc" and "Picard" is always preserved.

Determine if a String Starts or Ends With a Particular String

When you work with text, sometimes you need to determine if a given string starts with or ends with certain characters. You can use two string methods to solve this problem: `.startswith()` and `.endswith()`.

Let's look at an example. Consider the string `"Enterprise"`. Here's how you use `.startswith()` to determine if the string starts with the letters `e` and `n`:

```
>>> starship = "Enterprise"
>>> starship.startswith("en")
False
```

You must tell `.startswith()` what characters to search for by providing a string containing those characters. So, to determine if `"Enterprise"` starts with the letters `e` and `n`, you call `.startswith("en")`. This returns `False`. Why do you think that is?

If you guessed that `.startswith("en")` returns `False` because `"Enterprise"` starts with a capital `E`, you're absolutely right! The `.startswith()` method is **case-sensitive**. To get `.startswith()` to return `True`, you need to provide it with the string `"En"`:

```
>>> starship.startswith("En")
True
```

The `.endswith()` method is used to determine if a string ends with certain characters:

```
>>> starship.endswith("rise")
True
```

Just like `.startswith()`, the `.endswith()` method is case-sensitive:

```
>>> starship.endswith("risE")
False
```

Note

The `True` and `False` values are not strings. They are a special kind of data type called a **Boolean value**. You will learn more about Boolean values in Chapter 8.

String Methods and Immutability

Recall from the previous section that strings are immutable—they can't be changed once they have been created. Most string methods that alter a string, like `.upper()` and `.lower()`, actually return copies of the original string with the appropriate modifications.

If you aren't careful, this can introduce subtle bugs into your program. Try this out in IDLE's interactive window:

```
>>> name = "Picard"
>>> name.upper()
'PICARD'
>>> name
'Picard'
```

When you call `name.upper()`, nothing about `name` actually changes. If you need to keep the result, you need to assign it to a variable:

```
>>> name = "Picard"
>>> name = name.upper()
>>> name
'PICARD'
```

`name.upper()` returns a new string `"PICARD"`, which is re-assigned to the `name` variable. This **overrides** the original string `"Picard"` assigned to `"name"`.

Use IDLE to Discover Additional String Methods

Strings have lots of methods associated to them. The methods introduced in this section barely scratch the surface. IDLE can help you

find new string methods. To see how, first assign a string literal to a variable in the interactive window:

```
>>> starship = "Enterprise"
```

Next, type `starship` followed by a period, but do not hit `Enter`. You should see the following in the interactive window:

```
>>> starship.
```

Now wait for a couple of seconds. IDLE displays a list of every string method that you can scroll through with the arrow keys.

A related shortcut in IDLE is the ability to fill in text automatically without having to type in long names by hitting `Tab`. For instance, if you only type in `starship.u` and then hit the `Tab` key, IDLE automatically fills in `starship.upper` because there is only one method belonging to `starship` that begins with a `u`.

This even works with variable names. Try typing in just the first few letters of `starship` and, assuming you don't have any other names already defined that share those first letters, IDLE completes the name `starship` for you when you hit the `Tab` key.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Write a script that converts the following strings to lowercase: `"Animals"`, `"Badger"`, `"Honey Bee"`, `"Honeybadger"`. Print each lowercase string on a separate line.
2. Repeat Exercise 1, but convert each string to uppercase instead of lowercase.
3. Write a script that removes whitespace from the following strings:

```
string1 = "    Filet Mignon"
string2 = "Brisket    "
```

```
string3 = "  Cheeseburger   "
```

Print out the strings with the whitespace removed.

4. Write a script that prints out the result of `.startswith("be")` on each of the following strings:

```
string1 = "Becomes"
string2 = "becomes"
string3 = "BEAR"
string4 = "  bEautiful"
```

5. Using the same four strings from Exercise 4, write a script that uses string methods to alter each string so that `.startswith("be")` returns `True` for all of them.

4.4 Interact With User Input

Now that you've seen how to work with string methods, let's make things interactive. In this section, you will learn how to get some input from a user with the `input()` function. You'll write a program that asks a user to input some text and then display that text back to them in uppercase.

Enter the following into IDLE's interactive window:

```
>>> input()
```

When you press Enter, it looks like nothing happens. The cursor moves to a new line, but a new >>> doesn't appear. Python is waiting for you to enter something!

Go ahead and type some text and press Enter:

```
>>> input()
Hello there!
'Hello there!'
>>>
```

The text you entered is repeated on a new line with single quotes.

That's because `input()` returns any text entered by the user as a string.

To make `input()` a bit more user friendly, you can give it a **prompt** to display to the user. The prompt is just a string that you put in between the parentheses of `input()`. It can be anything you want: a word, a symbol, a phrase—anything that is a valid Python string.

The `input()` function displays the prompt and waits for the user to type something on their keyboard. When the user hits `Enter`, `input()` returns their input as a string that can be assigned to a variable and used to do something in your program.

To see how `input()` works, save and run the following script:

```
prompt = "Hey, what's up? "
user_input = input(prompt)
print("You said:", user_input)
```

When you run this script, you'll see `Hey, what's up?` displayed in the interactive window with a blinking cursor. Because there is a single space at the end of this string, any text entered by the user will be separated from the prompt by a space.

The single space at the end of the string `"Hey, what's up? "` makes sure that when the user starts to type, the text is separated from the prompt with a space. When you type a response and press `Enter`, your response is assigned to the `user_input` variable.

Here's a sample run of the program:

```
Hey, what's up? Mind your own business.


You said: Mind your own business.
```

Once you have input from a user, you can do something with it. For example, the following script takes user input and converts it to uppercase with `.upper()` and prints the result:

```
response = input("What should I shout? ")
shouted_response = response.upper()
print("Well, if you insist...", shouted_response)
```

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Write a script that takes input from the user and displays that input back.
2. Write a script that takes input from the user and displays the input in lowercase.
3. Write a script that takes input from the user and displays the number of characters inputted.

4.5 Challenge: Pick Apart Your User's Input

Write a script named `first_letter.py` that first prompts the user for input by using the string `"Tell me your password:"` The script should then determine the first letter of the user's input, convert that letter to upper-case, and display it back.

For example, if the user input is `"no"` then the program should respond like this:

```
The first letter you entered was: N
```

For now, it's okay if your program crashes when the user enters nothing as input—that is, they just hit `Enter` instead of typing something in. You'll learn about a couple of ways you can deal with this situation in an upcoming chapter.

You can find the solutions to this code challenge and many other bonus resources online at realpython.com/python-basics/resources.

4.6 Working With Strings and Numbers

When you get user input using the `input()` function, the result is always a string. There are many other times when input is given to a program as a string. Sometimes those strings contain numbers that need to be fed into calculations.

In this section you will learn how to deal with strings of numbers. You will see how arithmetic operations work on strings, and how they often lead to surprising results. You will also learn how to convert between strings and number types.

Strings and Arithmetic Operators

You've seen that string objects can hold many types of characters, including numbers. However, don't confuse numerals in a string with actual numbers. For instance, try this bit of code out in IDLE's interactive window:

```
>>> num = "2"
>>> num + num
'22'
```

The `+` operator concatenates two string together. So, the result of `"2" + "2"` is `"22"`, not `"4"`.

Strings can be "multiplied" by a number as long as that number is an integer, or whole number. Type the following into the interactive window:

```
>>> num = "12"
>>> num * 3
'121212'
```

`num * 3` concatenates the string `"12"` with itself three times and returns the string `"121212"`. To compare this operation to arithmetic with numbers, notice that `"12" * 3 = "12" + "12" + "12"`. In other words, multiplying a string by an integer `n` concatenates that string with itself `n` times.

The number on the right-hand side of the expression `num * 3` can be moved to the left, and the result is unchanged:

```
>>> 3 * num
'121212'
```

What do you think happens if you use the `*` operator between two strings? Type `"12" * "3"` in in the interactive window and press `Enter`:

```
>>> "12" * "3"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't multiply sequence by non-int of type 'str'
```

Python raises a `TypeError` and tells you that you can't multiply a sequence by a non-integer. When the `*` operator is used with a string on either the left or the right side, it always expects an integer on the other side.

Note

A **sequence** is any Python object that supports accessing elements by index. Strings are sequences. You will learn about other sequence types in Chapter 9.

What do you think happens when you try to add a string and a number?

```
>>> "3" + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Again, Python throws a `TypeError` because the `+` operator expects both things on either side of it to be of the same type. If any one of the objects on either side of `+` is a string, Python tries to perform string concatenation. Addition will only be performed if both objects are numbers. So, to add `"3" + 3` and get `6`, you must first convert the

string `"3"` to a number.

Converting Strings to Numbers

The `TypeError` errors you saw in the previous section highlight a common problem encountered when working with user input: type mismatches when trying to use the input in an operation that requires a number and not a string.

Let's look at an example. Save and run the following script.

```
num = input("Enter a number to be doubled: ")
doubled_num = num * 2
print(doubled_num)
```

When you enter a number, such as `2`, you expect the output to be `4`, but in this case, you get `22`. Remember, `input()` always returns a string, so if you input `2`, then `num` is assigned the string `"2"`, not the integer `2`. Therefore, the expression `num * 2` returns the string `"2"` concatenated with itself, which is `"22"`.

To perform arithmetic on numbers that are contained in a string, you must first convert them from a string type to a number type. There are two ways to do this: `int()` and `float()`.

`int()` stands for **integer** and converts objects into whole numbers, while `float()` stands for **floating-point number** and converts objects into numbers with decimal points. Here's what using them looks like in the interactive window:

```
>>> int("12")
12


>>> float("12")
12.0
```

Notice how `float()` adds a decimal point to the number. Floating-point numbers always have at least one decimal place of precision. For

this reason, you can't change a string that looks like a floating-point number into an integer because you would lose everything after the decimal point:

```
>>> int("12.0")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: invalid literal for int() with base 10: '12.0'
```

Even though the extra `0` after the decimal place doesn't add any value to the number, Python won't change `12.0` into `12` because it would result in the loss of precision.

Let's revisit the script from the beginning of this section and see how to fix it. Here's the script again:

```
num = input("Enter a number to be doubled: ")
doubled_num = num * 2
print(doubled_num)
```

The issue lies in the line `doubled_num = num * 2` because `num` references a string and `2` is an integer. You can fix the problem by wrapping `num` with either `int()` or `float()`. Since the prompts asks the user to input a number, and not specifically an integer, let's convert `num` to a floating-point number:

```
num = input("Enter a number to be doubled: ")
doubled_num = float(num) * 2
print(doubled_num)
```

Now when you run this script and input `2`, you get `4.0` as expected. Try it out!

Converting Numbers to Strings

Sometimes you need to convert a number to a string. You might do this, for example, if you need to build a string from some pre-existing variables that are assigned to numeric values.

As you've already seen, the following produces a `TypeError`:

```
>>> num_pancakes = 10
>>> "I am going to eat " + num_pancakes + " pancakes."
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can only concatenate str (not "int") to str
```

Since `num_pancakes` is a number, Python can't concatenate it with the string `"I'm going to eat"`. To build the string, you need to convert `num_pancakes` to a string using `str()`:

```
>>> num_pancakes = 10
>>> "I am going to eat " + str(num_pancakes) + " pancakes."
'I am going to eat 10 pancakes.'
```

You can also call `str()` on a number literal:

```
>>> "I am going to eat " + str(10) + " pancakes."
'I am going to eat 10 pancakes.'
```

`str()` can even handle arithmetic expressions:

```
>>> total_pancakes = 10
>>> pancakes_eaten = 5
>>> "Only " + str(total_pancakes - pancakes_eaten) + " pancakes left."
"Only 5 pancakes left."
```

You're not limited to numbers when using `str()`. You can pass it all sorts of objects to get their string representations:

```
>>> str(print)
'<built-in function print>'

>>> str(int)
"<class 'int'>"

>>> str(float)
```

```
"<class 'float'>"
```

These examples may not seem very useful, but they illustrate how flexible `str()` is.

In the next section, you'll learn how to format strings neatly to display values in a nice, readable manner. Before you move on, though, check your understanding with the following review exercises.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Create a string containing an integer, then convert that string into an actual integer object using `int()`. Test that your new object is a number by multiplying it by another number and displaying the result.
2. Repeat the previous exercise, but use a floating-point number and `float()`.
3. Create a string object and an integer object, then display them side-by-side with a single print statement by using the `str()` function.
4. Write a script that gets two numbers from the user using the `input()` function twice, multiplies the numbers together, and displays the result. If the user enters `2` and `4`, your program should print the following text:

   ```
   The product of 2 and 4 is 8.0.
   ```

4.7 Streamline Your Print Statements

Suppose you have a string `name = "Zaphod"` and two integers `heads = 2` and `arms = 3`. You want to display them in the following line: `Zaphod has 2 heads and 3 arms`. This is called **string interpolation**, which is just a fancy way of saying that you want to insert some variables into specific locations in a string.

You've already seen two ways of doing this. The first involves using commas to insert spaces between each part of the string inside of a `print()` function:

```
print(name, "has", str(heads), "heads and", str(arms), "arms")
```

Another way to do this is by concatenating the strings with the `+` operator:

```
print(name + " has " + str(heads) + " heads and " + str(arms) + " arms")
```

Both techniques produce code that can be hard to read. Trying to keep track of what goes inside or outside of the quotes can be tough. Fortunately, there's a third way of combining strings: formatted string literals, more commonly known as f-strings.

The easiest way to understand f-strings is to see them in action. Here's what the above string looks like when written as an f-string:

```
>>> f"{name} has {heads} heads and {arms} arms"
'Zaphod has 2 heads and 3 arms'
```

There are two important things to notice about the above examples:

1. The string literal starts with the letter `f` before the opening quotation mark
2. Variable names surrounded by curly braces (`{` and `}`) are replaced with their corresponding values without using `str()`

You can also insert Python expressions in between the curly braces. The expressions are replaced with their result in the string:

```
>>> n = 3
>>> m = 4
>>> f"{n} times {m} is {n*m}"
'3 times 4 is 12'
```

It is a good idea to keep any expressions used in an f-string as simple as possible. Packing in a bunch of complicated expressions into a

string literal can result in code that is difficult to read and difficult to maintain.

f-strings are only available in Python version 3.6 and above. In earlier versions of Python, the `.format()` method can be used to get the same results. Returning to the Zaphod example, you can use `.format()` method to format the string like this:

```
>>> "{} has {} heads and {} arms".format(name, heads, arms)
'Zaphod has 2 heads and 3 arms'
```

f-strings are shorter, and sometimes more readable, than using `.format()`. You will see f-strings used throughout this book.

For an in-depth guide to f-strings and comparisons to other string formatting techniques, check out the Python 3's f-Strings: An Improved String Formatting Syntax (Guide) on realpython.com

Note

There is also another way to print formatted strings: using the `%` operator. You might see this in code that you find elsewhere, and you can read about how it works here if you're curious.

Keep in mind that this style has been phased out entirely in Python 3. Just be aware that it exists and you may see it in legacy Python code bases.

Review Exercises

You can find the solutions to these exercises and many other bonus resources online at realpython.com/python-basics/resources.

1. Create a `float` object named `weight` with the value 0.2, and create a string object named `animal` with the value `"newt"`. Then use these objects to print the following string using only string concatenation:

This is a sample from "Python Basics: A Practical Introduction to Python 3"

With the full version of the book you get a complete Python curriculum to go all the way from beginner to intermediate-level. Every step along the way is explained and illustrated with short & clear code samples.

Coding exercises within each chapter and our interactive quizzes help fast-track your progress and ensure you always know what to focus on next.

Become a fluent Pythonista and gain programming knowledge you can apply in the real-world, today:

If you enjoyed the sample chapters you can purchase a full version of the book at realpython.com/pybasics-book.
