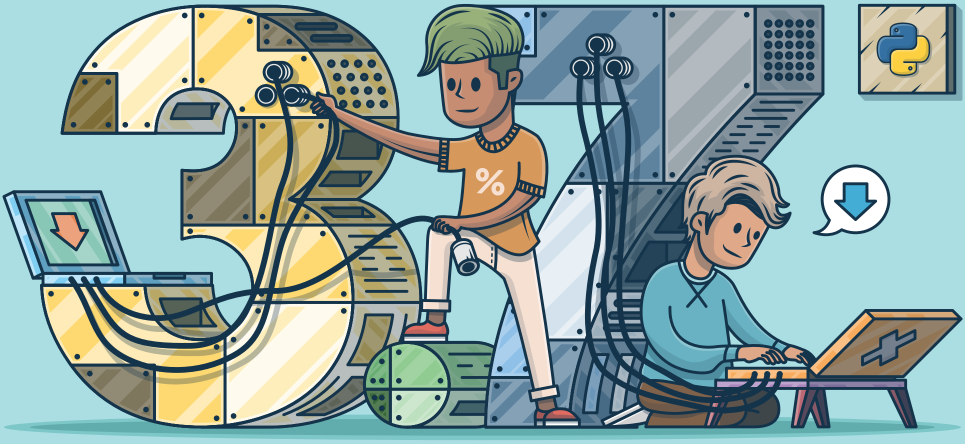
Real Python

UP TO DATE FOR PYTHON 3.7

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

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

Updated 2019-09-09 We would like to thank our early access readers for their excellent feedback: Zoheb Ainapore, Marc Poulin, Meir Guttman, Robert Livingston, Ricky, Joanna Jablonski, Mursalin Simpson, Xu Chunyang, W., Vlad, JIm Anderson, Vivek, Srinivasan Samuel, marp, Jorge Alberch, Edythe, Miguel Galán, Tom Carnevale, Florent, Patrick Starrenburg, Gavin, Karen H Calhoun MD, Roman, Robert Robb Livingston, Terence Phillips, Nico, Daniel, Jacob Andersen, Mario, Alejandro Ramos, Beni_begin, AJ, CJ Hwang, Guido, Evan, Hans van Nielen, Dave, Miguel Galan, Han Qi, Jim Bremner, Matt Chang, Daniel Drazan, Cole, Bob, Reed Howald, Youri Torchalski, Edward Duarte, Mike Parker, Aart Kleinendorst, rock, Johnny, Rock Lee, Dusan Ranisavljev, Grant, Jack, Reinhard, Ceejay Cervantes, David, Vivek Vashist, Mark, Robert, Dan, Garett, Peter, Jun Lee, James Silk, Nik Singhal, Charles, Allard Schmidt, Jeff Desalle, Miguel, Steve Poe, Jonathan Seubert, Rohit Bharti, Phillip Douglas, John Stephenson, Jeff Jones, George Mast, Allards, Palak, Nikola N., Palak Kalsi, Annekathrin, Tsung-Ju Yang, Nick Huntington, Sai, Jordan, Wim Alsemgeest, DJ, Bob Harris, Martin, Andrew, Reggie Smith, Steve Santy, Mohee Jarada, Mark Arzaga, Larry Eisenberg, Poulose Matthen, Brent Gordon, Gary Butler, Bryant, Dana, Koajck, Reggie, Luis Bravo, Elijah, Nikolay, Eric Lietsch, Fred Janssen, Don Stillwell, Gaurav Sharma, Mike, Mike McKenna, karthik babu, bulat, Bulat Mansurov, August Trillanes, Darren Saw, Jagadish, Nathan Eger, Kyle, Tejas Shetty, Baba Sariffodeen, Don, Ian, Ian Barbour, Redhouane, Wayne Rosing, Emanuel, Toigongonbai, Jason Castillo, krishna chaitanya swamy kesavarapu, Corey Huguley, Nick, xuchunyang, Daniel BUIS, kenneth, Leodanis Pozo Ramos, John Phenix, Linda Moran, W Laleau, Troy Flynn, Heber Nielsen, ROCK, Mike LeRoy, Thomas Davis, Jacob, Szabolcs Sinka, kalaiselvan, Leanne Kuss, Andrey, omar, Jason Woden, David Cebalo, John Miller, David Bui, Nico Zanferrari, Ariel, Boris, Boris Ender, Charlie3, Ossy, Matthias Kuehl, Scott Koch, Jesus Avina, charlie3, Awadhesh, Andie, Chris Johnson, Malan, Ciro, Thamizhselvan, Neha, Christian Langpap, Ivan, Dr. Craig Levy, H B Robinson, Stéphane, Steve McIlree, Yves, Teresa, Allard, tom cone jr, Dirk, Joachim van der Weijden, Jim Woodward, Christoph Lipka, John Vergelli, Gerry, Lu, Robert R., vlad, Richard Heatwole, Gabriel, Krzysztof Surowiecki, Alexandra Davis, Jason Voll, and Dwayne Dever. Thank you all!

This is an Early Access version of “Python Basics: A Practical Introduction to Python 3”

With your help we can make this book even better:

At the end of each section of the book you’ll find a “magical” feedback link. Clicking the link takes you to an **online feedback form** where you can share your thoughts with us.

We welcome any and all feedback or suggestions for improvement you may have.

Please feel free to be as terse or detailed as you see fit. All feedback is stored anonymously, but you can choose to leave your name and contact information so we can follow up or mention you on our “Thank You” page.

We use a different feedback link for each section, so we’ll always know which part of the book your notes refer to.

Thank you for helping us make this book an even more valuable learning resource for the Python community.

— Dan Bader, Editor-in-Chief at Real Python

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 simple 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.

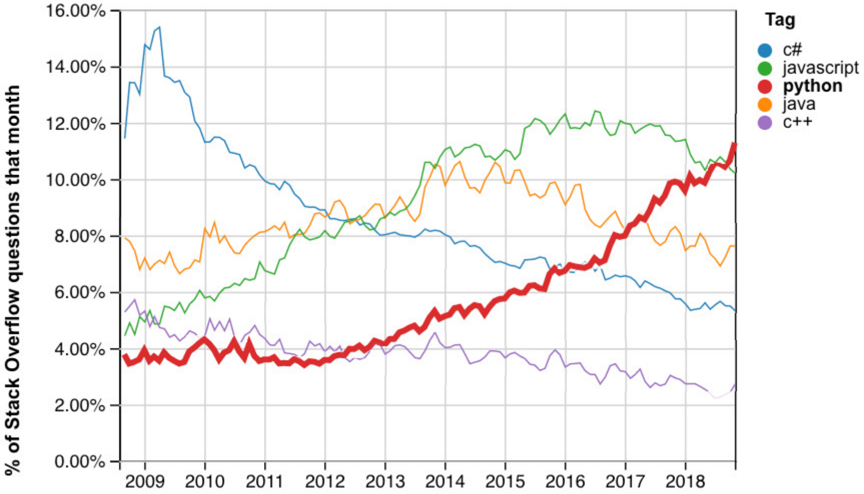
Python Is Popular

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:



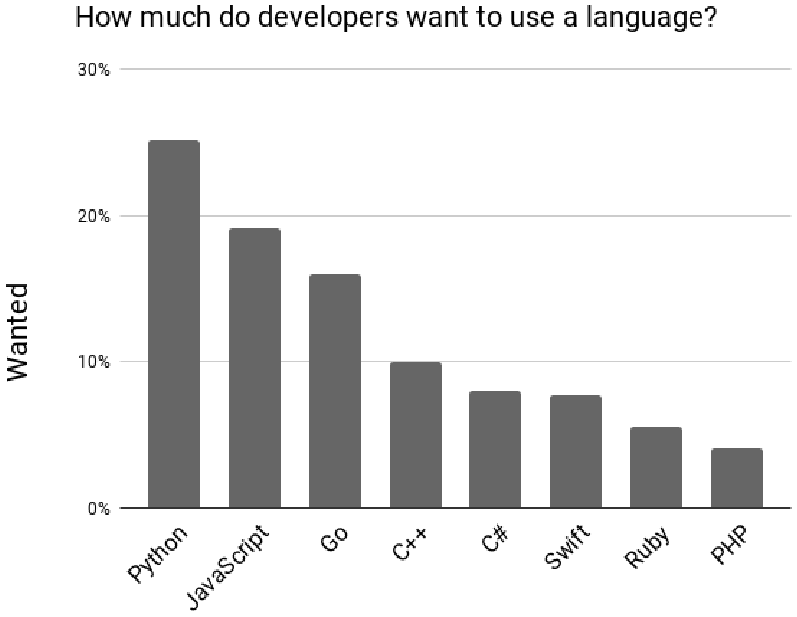
You can explore this chart and create similar charts to this one over at insights.stackoverflow.com/trends.

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 3.7! 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 languages. 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 figuring 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

The example Python scripts associated with this book are licensed under a Creative Commons Public Domain (CC0) License. This means that you're welcome to use any portion of the code for any purpose in your own programs.

> **Note**
>
> The code found in this book has been tested with Python 3.6 and Python 3.7 on Windows, macOS, and Linux.

Formatting Conventions

Code blocks will be used to present example code:

```python
# 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.

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

Leave feedback on this section »

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

Note

Even if you already have Python 3.7 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.

Throughout this book, IDLE will be used to create and modify Python code files. If you have a different preferred code editor, then you may follow along with the examples using that editor.

Just know that some sections, particularly the material covered in Chapter 7, will not apply to code editors other than IDLE.

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.

There are two major versions of Python available: Python 2, also known as legacy Python, and Python 3. Python 2 was released in the year 2000 and will reach its end-of-life on January 1, 2020. This book focuses exclusively on Python 3.

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

If you have a different operating system, check out the Python 3 Installation & Setup Guide maintained on realpython.com to see if your OS is covered.

Let's dig in!

Leave feedback on this section »

2.1 Windows

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

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

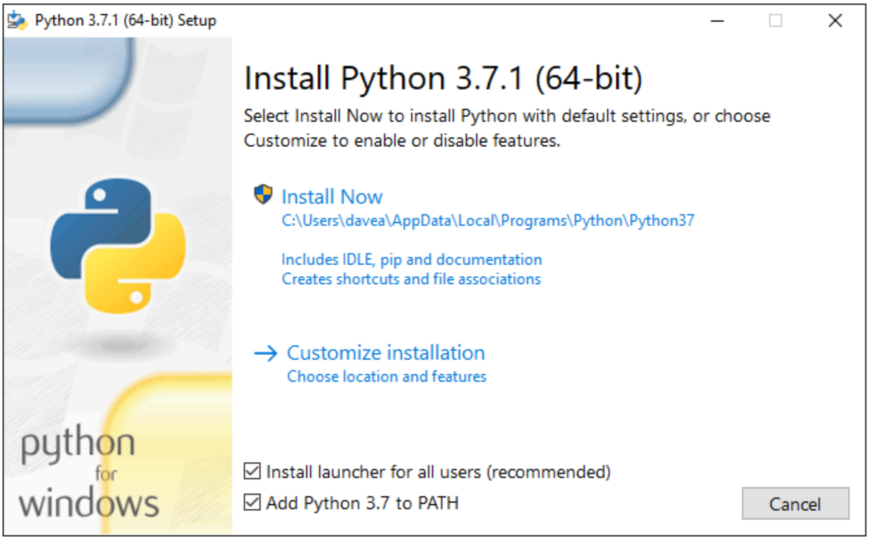
Open a browser window and navigate to the download page for Windows at python.org.

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 3.7. 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:


Python 3.7.1 (64-bit) Setup

Install Python 3.7.1 (64-bit)

Select Install Now to install Python with default settings, or choose Customize to enable or disable features.

Install Now
C:\Users\davea\AppData\Local\Programs\Python\Python37

Includes IDLE, pip and documentation
Creates shortcuts and file associations

→ Customize installation
Choose location and features

☑ Install launcher for all users (recommended)
☑ Add Python 3.7 to PATH

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.

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

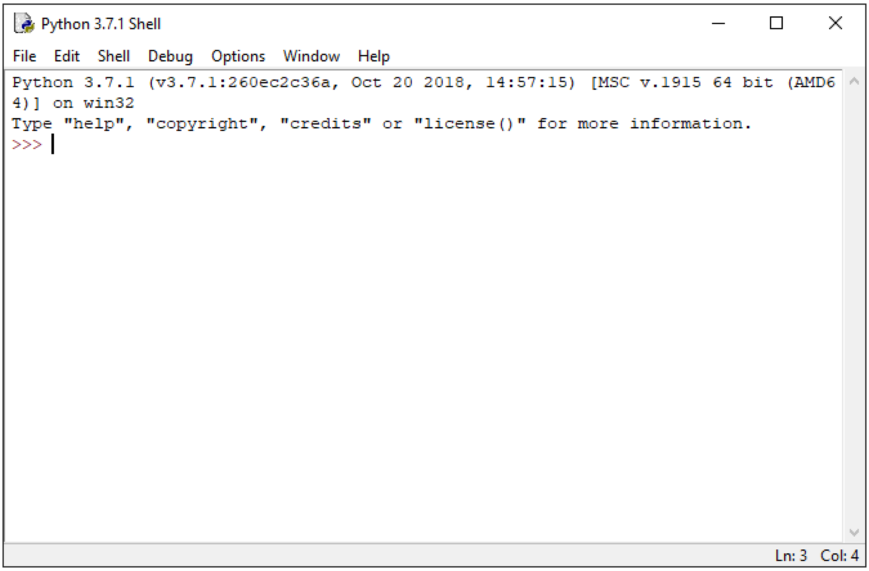
Open IDLE

You can open IDLE in two steps:

1. Click on the start menu and locate the *Python 3.7* folder.
2. Open the folder and select *IDLE (Python 3.7)*.

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:



```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MSC v.1915 64 bit (AMD6
4)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> 
```

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.

Leave feedback on this section »

2.2 macOS

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

Note

Many resources recommend installing Python 3 on macOS with the Homebrew package manager. Community guides like The Hitchhiker's Guide to Python also recommend this approach, as does Real Python's Python 3 Installation & Setup Guide.

Homebrew is useful for installing packages for macOS, including Python, from the terminal. While Homebrew is something you may want to learn to use, the process of getting Homebrew installed and using it to install Python can be daunting for a beginner.

If you are interested in using Homebrew, check out the Python 3 Installation & Setup Guide for step-by-step instructions.

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.

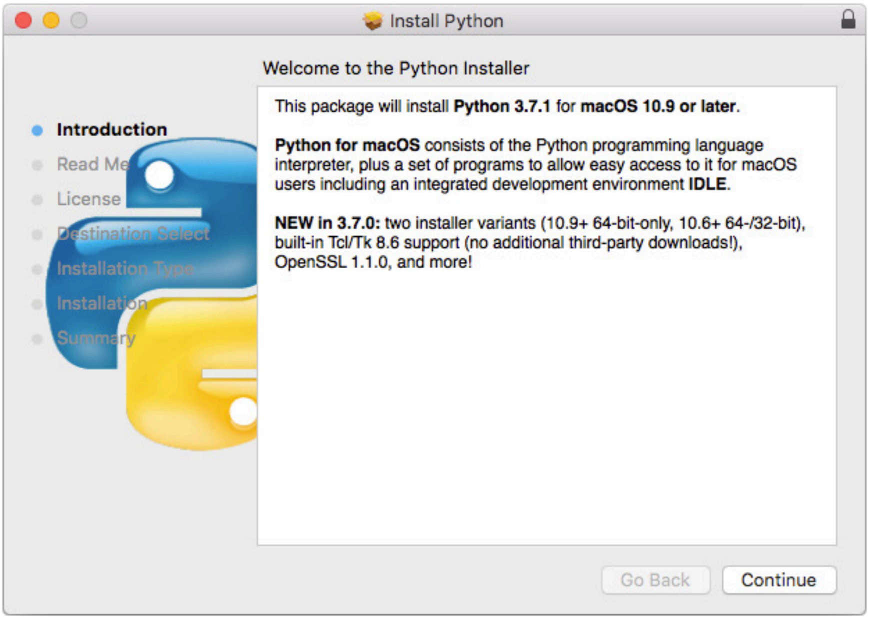
Step 1: Download the Python 3 Installer

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 3.7. 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:



Welcome to the Python Installer

This package will install **Python 3.7.1** for **macOS 10.9 or later**.

Python for macOS consists of the Python programming language interpreter, plus a set of programs to allow easy access to it for macOS users including an integrated development environment **IDLE**.

NEW in 3.7.0: two installer variants (10.9+ 64-bit-only, 10.6+ 64-/32-bit), built-in Tcl/Tk 8.6 support (no additional third-party downloads!), OpenSSL 1.1.0, and more!

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

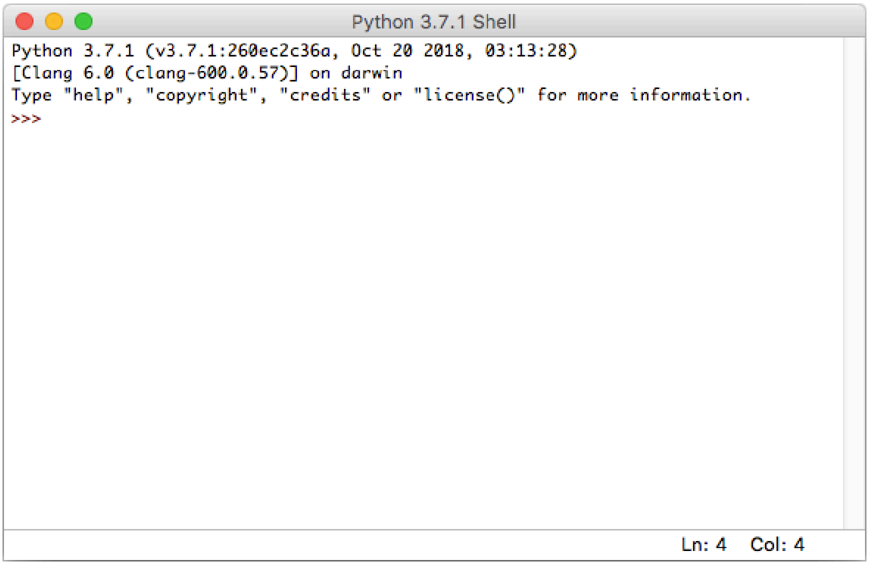
You can open IDLE in three steps:

1. Open Finder and click on *Applications*.
2. Locate the *Python 3.7* folder and double-click on it.
3. Double-click on the IDLE icon.

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!

> **Note**
>
> To open IDLE even more quickly, press `Cmd+Spacebar` to open the Spotlight search, type the word `idle`, and press `Return`.

The Python shell window looks like this:



```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 03:13:28)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license()" for more information.
>>>
```

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.

Leave feedback on this section »

2.3 Ubuntu Linux

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

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 3.7.2
```

If the version shown is Python 2.x or a version of Python 3 that is less than 3.7, 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 3.7 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 python3.7 idle-python3.7
  ```

- If you are using **Ubuntu 17 and lower**, Python 3.7 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 python3.7 idle-python3.7
  ```

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 `python3.7 --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-python3.7
```

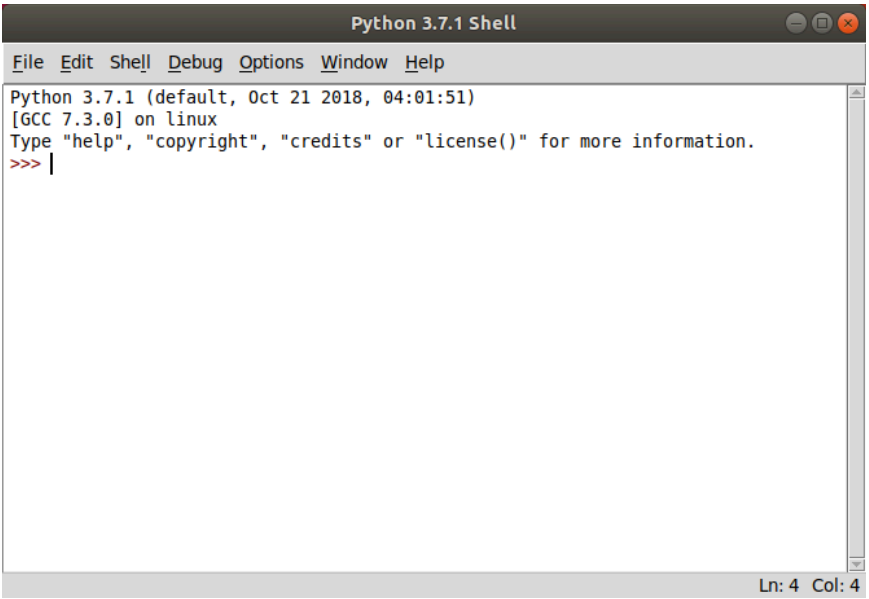
Note

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

```
$ idle3
```

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:



```
Python 3.7.1 (default, Oct 21 2018, 04:01:51)
[GCC 7.3.0] on linux
Type "help", "copyright", "credits" or "license()" for more information.
>>> 
```

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.

Leave feedback on this section »

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!

Leave feedback on this section »

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 3.7.2 (default, Dec 25 2018, 03:50:46)
[GCC 7.3.0] on linux
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 3.7.2. 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, or re-run, a script, you may see the following output in 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*.

Leave feedback on this section »

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. That above traceback tells us 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.

Leave feedback on this section »

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 you're 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 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:

```python
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 every other letter is 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, what the rules for valid variable names, and some guidelines for choosing a 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.

Leave feedback on this section »

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.

Leave feedback on this section »

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 let's 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, you can highlight the lines to be comments and press `Ctrl+D` (`Cmd+D` on macOS). This puts two `#` symbols at the beginning of each line, which 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 `Ctrl+Shift+D` (`Cmd+Shift+D` on macOS).

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.

Some programmers strive to write **self-documenting** code, which is explicit enough to understand without inserting comments. This is not always possible, though!

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.

Leave feedback on this section »

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 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

Leave feedback on this section »

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, such as changing a string from lowercase to uppercase, removing whitespace from the beginning or end of string, or replacing parts of a string with different text.

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!

Leave feedback on this section »

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. `"×Pýŧħøŋ×"` 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 one-line string that you have written out on multiple lines.

Leave feedback on this section »

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 an 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` the 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":

```
|  a  |  p  |  p  |  l  |  e  |     |  p  |  i  |  e  |
   0     1     2     3     4     5     6     7     8
```

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"`:

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

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 string variables, then print one of them after the other (with a space added in between) using a comma in your print statement.
4. Repeat exercise 3, but instead of using commas in print(), use concatenation to add a space between the two strings.
5. Print the string "zing" by using slice notation on the string "bazinga" to specify the correct range of characters.

Leave feedback on this section »

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 hit `Ctrl+Space` (`Cmd+Space` on macOS). 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.

Leave feedback on this section »

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.

To use the `input()` function, you must specify a **prompt**. 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 that 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 "shouts" it back by converting the input to uppercase with `.upper()` and printing the result:

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

Leave feedback on this section »

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.

Leave feedback on this section »

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 an integer. 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 throws 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 to 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.
   ```

Leave feedback on this section »

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. This method also works in Python 3.6 and 3.7. 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. That said, there are still some cases when the `.format()` method is preferable to an f-string, but those cases are beyond the scope of 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:

```
0.2 kg is the weight of the newt.
```

2. Display the same string by using the `.format()` method and empty `{}` place-holders.

3. Display the same string using an f-string.

Leave feedback on this section »

4.8 Find a String in a String

One of the most useful string methods is `.find()`. As its name implies, you can use this method to find the location of one string in another string—commonly referred to as a **substring**.

To use `.find()`, tack it to the end of a variable or a string literal and pass the string you want to find in between the parentheses:

```
>>> phrase = "the surprise is in here somewhere"
>>> phrase.find("surprise")
4
```

The value that `.find()` returns is the index of the first occurrence of the string you pass to it. In this case, `"surprise"` starts at the fifth character of the string `"the surprise is in here somewhere"` which has index `4` because counting starts at `0`.

If `.find()` doesn't find the desired substring, it will return `-1` instead:

```
>>> phrase = "the surprise is in here somewhere"
>>> phrase.find("eyjafjallajökull")
-1
```

You can call string methods on a string literal directly, so in this case, you don't need to create a new string:

```
>>> "the surprise is in here somewhere".find("surprise")
4
```

Keep in mind that this matching is done exactly, character by character, and is case-sensitive. For example, if you try to find `"SURPRISE"`, the `.find()` method returns `-1`:

```
>>> "the surprise is in here somewhere".find("SURPRISE")
-1
```

If a substring appears more than once in a string, `.find()` only returns the index of the first appearance, starting from the beginning of the string:

```
>>> "I put a string in your string".find("string")
8
```

There are two instances of the `"string"` in `"I put a string in your string"`. The first starts at index `8`, and the second at index `23`. `.find()` returns `8`, which is the index of the first instance of `"string"`.

The `.find()` method only accepts a string as its input. If you want to find an integer in a string, you need to pass the integer to `.find()` as a string. If you do pass something other than a string to `.find()`, Python raises a `TypeError`:

```
>>> "My number is 555-555-5555".find(5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: must be str, not int

>>> "My number is 555-555-5555".find("5")
13
```

Sometimes you need to find all occurrences of a particular substring and replace them with a different string. Since `.find()` only returns the index of the first occurrence of a substring, you can't easily use it to perform this operation. Fortunately, string objects have a `.replace()` method that replaces each instance of a substring with another string.

Just like .find(), you tack .replace() on to the end of a variable or string literal. In this case, though, you need to put two strings inside of the parentheses in .replace() and separate them with a comma. The first string is the substring to find, and the second string is the string to replace each occurrence of the substring with.

For example, the following code shows how to replace each occurrence of "the truth" in the string "I'm telling you the truth; nothing but the truth" with the string "lies":

```
>>> my_story = "I'm telling you the truth; nothing but the truth!"
>>> my_story.replace("the truth", "lies")
"I'm telling you lies; nothing but lies!"
```

Since strings are immutable objects, .replace() doesn't alter my_story. If you immediately type my_story into the interactive window after running the above example, you'll see the original string, unaltered:

```
>>> my_story
"I'm telling you the truth; nothing but the truth!"
```

To change the value of my_story, you need to reassign to it the new value returned by .replace():

```
>>> my_story = my_story.replace("the truth", "lies")
>>> my_story
"I'm telling you lies; nothing but lies!"
```

.replace() can only replace one substring at a time, so if you want to replace multiple substrings in a string you need to use .replace() multiple times.

For example, the following code converts the string "all" to "bee" by using .replace() twice:

```
>>> text = "all"
>>> new_text = text.replace("a", "b")
>>> new_text = new_text.replace("l", "e")
>>> new_text
'bee'
```

The `.replace()` method is a powerful string method. You'll get a taste for how powerful it is in the assignment in the next section. Before you attempt that assignment, 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. In one line of code, display the result of trying to `.find()` the substring `"a"` in the string `"AAA"`. The result should be `-1`.
2. Create the variable `n = 2.0`. Then use `input()` to get a string from the user. Finally, display the index of the first occurrence of `n` in the input string using `.find()`.

   The output should look something like this:

   ```
   Enter a string: version 2.0
   8
   ```
3. Write and test a script that accepts user input using the `input()` function and displays the result of trying to `.find()` a particular letter in that input.

Leave feedback on this section »

4.9 Challenge: Turn Your User Into a L33t H4x0r

Write a script called `translate.py` that asks the user for some input with the following prompt: `Enter some text:`. Then use the `.replace()` method to convert the text entered by the user into "leetspeak" by making the following changes to lower-case letters:

- The letter `a` becomes `4`
- The letter `b` becomes `8`
- The letter `e` becomes `3`
- The letter `l` becomes `1`
- The letter `o` becomes `0`
- The letter `s` becomes `5`
- The letter `t` becomes `7`

Your program should then display the resulting string as output. Below is a sample run of the program:

```
Enter some text: I like to eat eggs and spam.
I 1ik3 70 347 3gg5 4nd 5p4m.
```

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

Leave feedback on this section »

4.10 Summary and Additional Resources

In this chapter, you learned the ins and outs of Python string objects. You learned how to access different characters in a string using subscripts and slices, as well as how to determine the length of a string with `len()`.

Strings come with numerous methods. The `.upper()` and `.lower()` methods convert all characters of a string to upper or lower case, respectively. The `.rstrip()`, `.lstrip()`, and `strip()` methods remove whitespace from strings, and the `.startswith()` and `.endswith()` methods will tell you if a string starts or ends with a given substring.

You also saw how to capture input from a user as a string using the `input()` function, and how to convert that input to a number using `int()`

and `float()`. To convert numbers, and other objects, to strings, you use `str()`.

Finally, you saw how the `.find()` and `.replace()` methods are used to find the location of a substring and replace a substring with a new string.

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-4

Additional Resources

To learn more, check out the following resources:

- Python String Formatting Best Practices
- Splitting, Concatenating, and Joining Strings in Python
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 5

Numbers and Math

You don’t need to be a math whiz to program well. The truth is, few programmers need to know more than basic algebra.

Of course, how much math you need to know depends on the application you are working on. In general, the level of math required to successfully work as a programmer is less than you might expect.

Although math and computer programming aren’t as correlated as some people might believe, numbers are an integral part of any programming language and Python is no exception.

In this chapter, you will learn how to:

- Work with Python’s three built-in number types: integer, floating-point, and complex numbers
- Round numbers to a given number of decimal places
- Format and display numbers in strings

Let’s get started!

Leave feedback on this section »

5.1 Integers and Floating-Point Numbers

Python has three built-in number data types: integers, floating-point numbers, and complex numbers. In this section, you'll learn about integers and floating-point numbers, which are the two most commonly used number types. You'll learn about complex numbers in section 5.6.

Integers

An **integer** is a whole number with no decimal places. For example, `1` is an integer, but `1.0` isn't. The name for the integer data type is `int`, which you can see with the `type()` function:

```
>>> type(1)
<class 'int'>
```

You can create an integer by simply typing the number explicitly or using the `int()` function. In Chapter 4, you learned how to convert a string containing an integer to a number using `int()`. For example, the following converts the string `"25"` to the integer `25`:

```
>>> int("25")
25
```

An **integer literal** is an integer value that is written explicitly in your code, just like a string literal is a string that is written explicitly in your code. For example, `1` is an integer literal, but `int("1")` isn't.

Integer literals can be written in two different ways:

```
>>> 1000000
1000000

>>> 1_000_000
1000000
```

The first example is straightforward. Just type a `1` followed by six zeros. The downside to this notation is that large numbers can be difficult to read.

When you write large numbers by hand, you probably group digits into groups of three, separated by a comma. 1,000,000 is a lot easier to read than 1000000.

In Python, you can't use commas to group digits in integer literals, but you can use an underscore (_). The value `1_000_000` expresses one million in a more readable manner.

There is no limit to how large an integer can be, which might be surprising considering computers have finite memory. Try typing the largest number you can think of into IDLE's interactive window. Python can handle it with no problem!

Floating-Point Numbers

A **floating-point number**, or **float** for short, is a number with a decimal place. `1.0` is a floating-point number, as is `-2.75`. The name of a floating-point data type is `float`:

```
>>> type(1.0)
<class 'float'>
```

Floats can be created by typing a number directly into your code, or by using the `float()` function. Like `int()`, `float()` can be used to convert a string containing a number to a floating-point number:

```
>>> float("1.25")
1.25
```

A **floating-point literal** is a floating-point value that is written explicitly in your code. `1.25` is a floating-point literal, while `float("1.25")` is not.

Floating-point literals can be created in three different ways. Each of the following creates a floating-point literal with a value of one million:

```
>>> 1000000.0
1000000.0

>>> 1_000_000.0
1000000.0

>>> 1e6
1000000.0
```

The first two ways are similar to the two methods for creating integer literals that you saw early. The second method, which uses underscores to separate digits into groups of three, is useful for creating larger float literals.

For really large numbers, you can use **exponential notation**. The third method in the previous example uses exponential notation to create a float literal.

To write a float literal in exponential notation, type a number followed by the letter `e` and then another number. Python takes the number to the left of the `e` and multiplies by `10` raised to the power of the number after the `e`. So `1e6` is equivalent to 1×10^6.

Python also uses scientific notation to display large floating point numbers:

```
>>> 200000000000000000.0
2e+17
```

The float `200000000000000000.0` gets displayed as `2e+17`. The `+` sign indicates that the exponent `17` is a positive number. You can also use negative numbers as the exponent:

```
>>> 1e-4
0.0001
```

The literal `1e-4` is interpreted as `10` raised to the power `-4`, which is

1/1000 or, equivalently, `0.0001`.

Unlike integers, floats do have a maximum size. The maximum floating-point number depends on your system, but something like `2e400` ought to be well beyond most machines' capabilities. `2e400` is 2×10^{400}, which is far more than the total number of atoms in the universe!

When you reach the maximum floating-point number, Python returns a special float value `inf`:

```
>>> 2e400
inf
```

`inf` stand for infinity, and it just means that the number you've tried to create is beyond the maximum floating-point value allowed on your computer. The type of `inf` is still float:

```
>>> n = 2e400
>>> n
inf
>>> type(n)
<class 'float'>
```

There is also `-inf` which stands for negative infinity, and represents a negative floating-point number that is beyond the minimum floating-point number allowed on your computer:

```
>>> -2e400
-inf
```

You probably won't come across `inf` and `-inf` often as a programmer, unless you regularly work with extremely large numbers.

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 creates the two variables, `num1` and `num2`. Both `num1` and `num2` should be assigned the integer literal 25,000,000, one written with underscored and one without. Print `num1` and `num2` on two separate lines.
2. Write a script that creates assigns the floating-point literal `175000.0` to the variable `num` using exponential notation, and then prints `num` in the interactive window.
3. In the interactive window, try and find the smallest exponent `N` so that `2e<N>`, where `<N>` is replaced with your number, returns `inf`.

Leave feedback on this section »

5.2 Arithmetic Operators and Expressions

In this section, you'll learn how to do basic arithmetic with numbers in Python, such as addition, subtraction, multiplication, and division. Along the way, you'll learn some conventions for writing mathematical expressions in code.

Addition

Addition is performed with the `+` operator:

```
>>> 1 + 2
3
```

The two numbers on either side of the `+` operator are called **operands**. In the previous example, both operands are integers, but operands do not need to be the same type. You can add an `int` to a `float` with no problem:

```
>>> 1.0 + 2
3.0
```

Notice that the result of `1.0 + 2` is `3.0`, which is a `float`. Any time a `float` is added to a number, the result is another `float`. Adding two integers together always results in an `int`.

Note

PEP 8 recommends separating both operands from an operator with a space.

Python can evaluate `1+1` just fine, but `1 + 1` is the preferred format because it's generally considered easier to read. This rule of thumb applies to all of the operators in this section.

Subtraction

To subtract two numbers, just put a `-` in between them:

```
>>> 1 - 1
0

>>> 5.0 - 3
2.0
```

Just like adding two integers, subtracting two integers always results in an `int`. Whenever one of the operands is a `float`, the result is also a `float`.

The `-` operator is also used to denote negative numbers:

```
>>> -3
-3
```

You can subtract a negative number from another number, but as you can see below, this can sometimes look confusing:

```
>>> 1 - -3
4

>>> 1 --3
```

```
4

>>> 1- -3
4

>>>1--3
4
```

Of the four examples above, the first is the most PEP 8 compliant. That said, you can surround `-3` with parentheses to make it even clearer that the second `-` is modifying `3`:

```
>>> 1 - (-3)
4
```

Using parentheses is a good idea because it makes the code more explicit. Computers execute code, but humans read code. Anything you can do to make your code easier to read and understand is a good thing.

Multiplication

To multiply two numbers, use the `*` operator:

```
>>> 3 * 3
9

>>> 2 * 8.0
16.0
```

The type of number you get from multiplication follows the same rules as addition and subtraction. Multiplying two integers results in an `int`, and multiplying a number with a `float` results in a `float`.

Division

The `/` operator is used to divide two numbers:

```
>>> 9 / 3
3.0

>>> 5.0 / 2
2.5
```

Unlike addition, subtraction, and multiplication, division with the `/` operator always returns a `float`. If you want to make sure that you get an integer after dividing two numbers, you can use `int()` to convert the result:

```
>>> int(9 / 3)
3
```

Keep in mind that `int()` discards any fractional part of the number:

```
>>> int(5.0 / 2)
2
```

`5.0 / 2` returns the float `2.5`, and `int(2.5)` returns the integer `2` with the `.5` part removed.

Integer Division

If writing `int(5.0 / 2)` seems a little long-winded to you, Python provides a second division operator, `//`, called the **integer division** operator:

```
>>> 9 // 3
3

>>> 5.0 // 2
2.0

>>> -3 // 2
-2
```

The `//` operator first divides the number on the left by the number on the right and then rounds down to an integer. This might not give the value you expect when one of the numbers is negative.

For example, `-3 // 2` returns `-2`. First, `-3` is divided by `-2` to get `-1.5`. Then `-1.5` is rounded down to `-2`. On the other hand, `3 // 2` returns `1`

Another thing the above example illustrates is that `//` preserves the type of the left-hand number, which is why `9 // 3` returns the integer `3` and `5.0 // 2` returns a the float `2.0`.

Let's see what happens when you try to divide a number by `0`:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
```

Python gives you a `ZeroDivisionError`, letting you know that you just tried to break a fundamental rule of the universe.

Exponents

You can raise a number to a power using the `**` operator:

```
>>> 2 ** 2
4

>>> 2 ** 3
8

>>> 2 ** 4
16
```

Exponents don't have to be integers. They can also be floats:

```
>>> 3 ** 1.5
5.196152422706632

>>> 9 ** 0.5
```

```
3.0
```

Raising a number to the power of `0.5` is the same as taking the square root, but notice that even though the square root of `9` is an integer, Python returns the float `3.0`.

The `**` operator returns an integer if both operands are integers, and a float if any one of the operands is a floating-point number.

You can also raise numbers to negative powers:

```
>>> 2 ** -1
0.5

>>> 2 ** -2
0.25
```

The Modulus Operator

The `%` operator, or the **modulus**, returns the remainder of dividing the left operand by the right operand:

```
>>> 5 % 3
2

>>> 20 % 7
6

>>> 16 % 8
0
```

`3` divides `5` once with a remainder of `2`, so `5 % 3` is `2`. Similarly, `7` divides `20` twice with a remainder of `6`.

In the last example, `16` is divisible by `8`, so `16 % 8` is `0`. Any time the number to the left of `%` is divisible by the number to the right, the result is `0`.

One of the most common uses of `%` is to determine whether or not one number is divisible by another. For example, a number `n` is even if and only if `n % 2` is `0`.

What do you think `1 % 0` returns? Let's try it out:

```
>>> 1 % 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: integer division or modulo by zero
```

This makes sense because `1 % 0` is the remainder of dividing `1` by `0`. But you can't divide `1` by `0`, so Python raises a `ZeroDivisionError`.

> **Note**
>
> When you work in IDLE's interactive window, errors like `ZeroDivisionError` don't cause much of a problem. The error is displayed and a new prompt pops up allowing you to continue writing code.
>
> However, whenever Python encounters an error while running a script, execution stops. The program is said to have **crashed**. In Chapter 8, you'll learn how to handle errors so that your programs don't crash unexpectedly.

Things get a little tricky when you use the `%` operator with negative numbers:

```
>>> 5 % -3
-1

>>> -5 % 3
1

>>> -5 % -3
-2
```

These potentially shocking results are really quite well defined. To calculate the remainder `r` of dividing a number `x` by a number `y`, Python uses the equation `r = x - (y * (x // y))`.

For example, to find `5 % -3`, first find `(5 // -3)`. Since `5 / -3` is about `-1.67`, `5 // -3` is `-2`. Now multiply that by `-3` to get `6`. Finally, subtract `6` from `5` to get `-1`.

Arithmetic Expressions

You can combine operators to form complex expressions. An **expression** is a combination of numbers, operators, and parentheses that Python can compute, or **evaluate**, to return a value.

Here are some examples of arithmetic expressions:

```
>>> 2*3 - 1
5

>>> 4/2 + 2**3
10.0

>>> -1 + (-3*2 + 4)
-3
```

The rules for evaluating expressions work are the same as in everyday arithmetic. In school, you probably learned these rules under the name "order of operations."

The `*`, `/`, `//`, and `%` operators all have equal **precedence**, or priority, in an expression, and each of these has a higher precedence than the `+` and `-` operators. This is why `2*3 - 1` returns `5` and not `4`. `2*3` is evaluated first, because `*` has higher precedence than the `-` operator.

You may notice that the expressions in the previous example do not follow the rule for putting a space on either side of all of the operators. PEP 8 says the following about whitespace in complex expressions:

> "If operators with different priorities are used, consider

> adding whitespace around the operators with the lowest priority(ies). Use your own judgment; however, never use more than one space, and always have the same amount of whitespace on both sides of a binary operator."
>
> — PEP 8, Other Recommendations

Leave feedback on this section »

5.3 Challenge: Perform Calculations on User Input

Write a script called `exponent.py` that receives two numbers from the user and displays the first number raised to the power of the second number.

A sample run of the program should look like this (with example input that has been provided by the user included below):

```
Enter a base: 1.2
Enter an exponent: 3
1.2 to the power of 3 = 1.7279999999999998
```

Keep the following in mind:

1. Before you can do anything with the user's input, you will have to assign both calls to `input()` to new variables.
2. The `input()` function returns a string, so you'll need to convert the user's input into numbers in order to do arithmetic.
3. You can use an f-string to print the result.
4. You can assume that the user will enter actual numbers as input.

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

Leave feedback on this section »

5.4 Make Python Lie to You

What do you think `0.1 + 0.2` is? The answer is `0.3`, right? Let's see what Python has to say about it. Try this out in the interactive window:

```
>>> 0.1 + 0.2
0.30000000000000004
```

Well, that's… *almost* right! What in the heck is going on here? Is this a bug in Python?

No, it isn't a bug! It's a **floating-point representation error**, and it has nothing to do with Python. It's related to the way floating-point numbers are stored in a computer's memory.

The number `0.1` can be represented as the fraction `1/10`. Both the number `0.1` and it's fraction `1/10` are **decimal representations**, or **base 10 representations**. Computers, however, store floating-point numbers in base 2 representation, more commonly called **binary representation**.

When represented in binary, something familiar yet possibly unexpected happens to the decimal number `0.1`. The fraction `1/3` has no finite decimal representation. That is, `1/3 = 0.3333...` with infinitely many 3's after the decimal point. The same thing happens to the fraction `1/10` in binary.

The binary representation of `1/10` is the following infinitely repeating fraction:

```
0.00011001100110011001100110011...
```

Computers have finite memory, so the number `0.1` must be stored as an approximation and not as its true value. The approximation that gets stored is slightly higher than the actual value, and looks like this:

```
0.1000000000000000055511151231257827021181583404541015625
```

You may have noticed, however, that when asked to print `0.1`, Python prints `0.1` and not the approximated value above:

```
>>> 0.1
0.1
```

Python doesn't just chop off the digits in the binary representation for `0.1`. What actually happens is a little more subtle.

Because the approximation of `0.1` in binary is just that—an approximation—it is entirely possible that more than one decimal number have the same binary approximation.

For example, the numbers `0.1` and `0.10000000000000001` both have the same binary approximation. Python prints out the shortest decimal number that shares the approximation.

This explains why, in the first example of this section, `0.1 + 0.2` does not equal `0.3`. Python adds together the binary approximations for `0.1` and `0.2`, which gives a number which is *not* the binary approximation for `0.3`.

If all this is starting to make your head spin, don't worry! Unless you are writing programs for finance or scientific computing, you don't need to worry about the imprecision of floating-point arithmetic.

Leave feedback on this section »

5.5 Math Functions and Number Methods

Python has a few built-in functions you can use to work with numbers. In this section, you'll learn about three of the most common ones:

1. `round()`, for rounding numbers to some number of decimal places

2. `abs()`, for getting the absolute value of a number
3. `pow()`, for raising a number to some power

You'll also learn about a method that floating-point numbers have to check whether or not they have an integer value.

Let's go!

The `round()` function

You can use `round()` to round a number to the nearest integer:

```
>>> round(2.3)
2

>>> round(2.7)
3
```

`round()` has some unexpected behavior when the number ends in `.5`:

```
>>> round(2.5)
2

>>> round(3.5)
4
```

`2.5` gets rounded down to `2` and `3.5` is rounded up to `4`. Most people expect a number that ends in `.5` to get rounded up, so let's take a closer look at what's going on here.

Python 3 rounds numbers according to a strategy called **rounding ties to even**. A **tie** is any number whose last digit is `5`. `2.5` and `3.1415` are ties, but `1.37` is not.

When you round ties to even, you first look at the digit one decimal place to the left of the last digit in the tie. If that digit is even, you round down. If the digit is odd, you round up. That's why `2.5` rounds down to `2` and `3.5` round up to `4`.

Note

Rounding ties to even is the rounding strategy recommended for floating-point numbers by the IEEE (Institute of Electrical and Electronics Engineers) because it helps limit the impact rounding has on operations involving lots of numbers.

The IEEE maintains a standard called IEEE 754 for how floating-point numbers are dealt with on a computer. It was published in 1985 and is still commonly used by hardware manufacturers today.

You can round a number to a given number of decimal places by passing a second argument to `round()`:

```
>>> round(3.14159, 3)
3.142

>>> round(2.71828, 2)
2.72
```

The number `3.14159` is rounded to `3` decimal places to get `3.142`, and the number `2.71828` is rounded to `2` decimal places to get `2.72`.

The second argument of `round()` must be an integer. If it isn't, Python raises a `TypeError`:

```
>>> round(2.65, 1.4)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    round(2.65, 1.4)
TypeError: 'float' object cannot be interpreted as an integer
```

Sometimes `round()` doesn't get the answer quite right:

```
>>> # Expected value: 2.68
>>> round(2.675, 2)
2.67
```

`2.675` is a tie because it lies exactly halfway between the numbers `2.67` and `2.68`. Since Python rounds ties to the nearest even number, you would expect `round(2.675, 2)` to return `2.68`, but it returns `2.67` instead. This error is a result of floating-point representation error, and isn't a bug in the `round()` function.

Dealing with floating-point numbers can be frustrating, but this frustration isn't specific to Python. All languages that implement the IEEE floating-point standard have the same issues, including C/C++, Java, and JavaScript.

In most cases, though, the little errors encountered with floating-point numbers are negligible, and the results of `round()` are perfectly useful.

The `abs()` Function

The **absolute value** of a number *n* is just *n* if *n* is positive, and -*n* if *n* is negative. For example, the absolute value of `3` is `3`, while the absolute value of `-5` is `5`.

To get the absolute value of a number in Python, you use the `abs()` function:

```
>>> abs(3)
3

>>> abs(-5.0)
5.0
```

`abs()` always returns a positive number of the same type as its argument. That is, the absolute value of an integer is always a positive integer, and the absolute value of a float is always a positive float.

The `pow()` Function

In section 5.2, you learned how to raise a number to a power using the `**` operator. You can also use the `pow()` function. `pow()` takes two arguments. The first is the **base**, that is the number to be raised to a power, and the second argument is the **exponent**.

For example, the following uses `pow()` to raise `2` to the exponent `3`:

```
>>> pow(2, 3)
8
```

Just like `**`, the exponent in `pow()` can be negative:

```
>>> pow(2, -2)
0.25
```

So, what's the difference between `**` and `pow()`? The `pow()` function accepts an optional third argument that computes the first number raised to the power of the second number and then takes the modulo with respect to the third number.

In other words, `pow(x, y, z)` is equivalent to `(x ** y) % z`. Here's an example with `x = 2`, `y = 3`, and `z = 2`:

```
>>> pow(2, 3, 2)
0
```

First, `2` is raised to the power `3` to get `8`. Then `8 % 2` is calculated, which is `0` because `2` divides `8` with no remainder.

Check if a Float Is Integral

In Chapter 3 you learned about string methods like `.lower()`, `.upper()`, and `.find()`. Integers and floating-point numbers also have methods.

Number methods aren't used very often, but there is one that can be useful. Floating-point numbers have an `.is_integer()` method that returns `True` if the number is **integral**—meaning it has no fractional part—and returns `False` otherwise:

```
>>> num = 2.5
>>> num.is_integer()
False

>>> num = 2.0
>>> num.is_integer()
True
```

The `.is_integer()` method can be useful for validating user input. For example, if you are writing an app for a shopping cart for a store that sells pizzas, you will want to check that the quantity of pizzas the customer inputs is a whole number. You'll learn how to do these kinds of checks in Chapter 8.

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 asks the user to input a number and then displays that number rounded to two decimal places. When run, your program should look like this:

   ```
   Enter a number: 5.432
   5.432 rounded to 2 decimal places is 5.43
   ```

2. Write a script that asks the user to input a number and then displays the absolute value of that number. When run, your program should look like this:

   ```
   Enter a number: -10
   The absolute value of -10 is 10.0
   ```

3. Write a script that asks the user to input two numbers by using the `input()` function twice, then display whether or not the difference between those two number is an integer. When run, your program should look like this:

   ```
   Enter a number: 1.5
   Enter another number: .5
   The difference between 1.5 and .5 is an integer? True!
   ```

   If the user inputs two numbers whose difference is not integral, the output should look like this:

```
Enter a number: 1.5
Enter another number: 1.0
The difference between 1.5 and 1.0 is an integer? False!
```

Leave feedback on this section »

5.6 Print Numbers in Style

Displaying numbers to a user requires inserting numbers into a string. In Chapter 3, you learned how to do this with f-strings by surrounding a variable assigned to a number with curly braces:

```
>>> n = 7.125
>>> f"The value of n is {n}"
'The value of n is 7.125'
```

Those curly braces support a simple formatting language you can use to alter the appearance of the value in the final formatted string.

For example, to format the value of `n` in the above example to two decimal places, replace the contents of the curly braces in the f-string with `{n:.2f}`:

```
>>> n = 7.125
>>> f"The value of n is {n:.2f}"
'The value of n is 7.12'
```

The colon (`:`) after the variable `n` indicates that everything after it is part of the formatting specification. In this example, the formatting specification is `.2f`.

The `.2` in `.2f` rounds the number to two decimal places, and the `f` tells Python to display `n` as a **fixed-point number**. This means the number is displayed with exactly two decimal places, even if the original number has fewer decimal places.

When `n = 7.125`, the result of `{n:.2f}` is `7.12`. Just like `round()`, Python rounds ties to even when formatting numbers inside of strings. So, if you replace `n = 7.125` with `n = 7.126`, then the result of `{n:.2f}` is `"7.13"`:

```
>>> n = 7.126
>>> f"The value of n is {n:.2f}"
'The value of n is 7.13'
```

To round to one decimal places, replace `.2` with `.1`:

```
>>> n = 7.126
>>> f"The value of n is {n:.1f}"
'The value of n is 7.1'
```

When you format a number as fixed-point, it's always displayed with the precise number of decimal places specified:

```
>>> n = 1
>>> f"The value of n is {n:.2f}"
'The value of n is 1.00'
>>> f"The value of n is {n:.3f}"
'The value of n is 1.000'
```

You can insert commas to group the integer part of large numbers by the thousands with the `,` option:

```
>>> n = 1234567890
>>> f"The value of n is {n:,}"
'The value of n is 1,234,567,890'
```

To round to some number of decimal places and also group by thousands, put the `,` before the dot `.` in your formatting specification:

```
>>> n = 1234.56
>>> f"The value of n is {n:,.2f}"
'The value of n is 1,234.56'
```

The specifier `,.2f` is useful for displaying currency values:

```
>>> balance = 2000.0
>>> spent = 256.35
>>> balance = balance - spent

>>> f"After spending ${spent:.2f}, I was left with ${balance:,.2f}"
'After spending $256.35, I was left with $1,743.65'
```

Another useful option is `%`, which is used to display percentages. The `%` option multiplies number by `100` and displays it in fixed-point format, followed by a percentage sign.

The `%` option should always go at the end of your formatting specification, and you can't mix it with the `f` option. For example, `.1%` displays a number as a percentage with exactly one decimal place:

```
>>> ratio = 0.9
>>> f"Over {ratio:.1%} of Pythonistas say 'Real Python rocks!'"
"Over 90.0% of Pythonistas say 'Real Python rocks!'"

>>> # Display percentage with 2 decimal places
>>> f"Over {ratio:.2%} of Pythonistas say 'Real Python rocks!'"
"Over 90.00% of Pythonistas say 'Real Python rocks!'"
```

The formatting mini language is powerful and extensive. You've only seen the basics here. For more information, you are encouraged to read the official documentation.

Review Exercises

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

1. Print the result of the calculation `3 ** .125` as a fixed-point number with three decimal places.
2. Print the number `150000` as currency, with the thousands grouped with commas. Currency should be displayed with two decimal places.
3. Print the result of `2 / 10` as a percentage with no decimal places.

The output should look like `20%`.

Leave feedback on this section »

5.7 Complex Numbers

Python is one of the few programming languages that provides built-in support for complex numbers. While complex numbers do not come up often outside the domains of scientific computing and computer graphics, Python's support for them is one of it's strengths.

If you have ever taken a pre-calculus or higher level algebra math class, you may remember that a complex number is a number with two distinct components: a **real** component and an **imaginary** component.

There are several ways to denote complex numbers, but a common method is to indicate the real component with the letter i and the imaginary component with the letter j. For example, $1i + 2j$ is the complex number with real part 1 and imaginary part 2.

To create a complex number in Python, you simply write the real part, followed by a plus sign and the imaginary part with the letter j at the end:

```
>>> n = 1 + 2j
```

When you inspect the value of `n`, you'll notice that Python wraps the number with parentheses:

```
>>> n
(1+2j)
```

This convention helps eliminate any confusion that the displayed output may represent a string or a mathematical expression.

Imaginary numbers come with two properties, `.real` and `.imag`, that return the real and imaginary component of the number, respectively:

```
>>> n.real
1.0

>>> n.imag
2.0
```

Notice that Python returns both the real and imaginary components as floats, even though they were specified as integers.

Complex numbers also have a `.conjugate()` method that returns the complex conjugate of the number:

```
>>> n.conjugate()
(1-2j)
```

For any complex number, its **conjugate** is the complex number with the same real part and an imaginary part that is the same in absolute value but with the opposite sign. So in this case, the complex conjugate of `1 + 2j` is `1 - 2j`.

Note

The `.real` and `.imag` properties don't need parentheses after them like the method `.conjugate()` does.

The `.conjugate()` method is a function that performs an action on the complex number. `.real` and `.imag` don't perform any action, they just return some information about the number.

The distinction between methods and properties is a part of **object oriented programming**, which you will learn about in Chapter 10.

All of the arithmetic operators that work with floats and integers work with complex numbers also, except for the floor division (`//`) operator. Since this isn't a math book, we won't discuss the mechanics of complex arithmetic. Instead, here are some examples of using complex numbers with arithmetic operators:

```
>>> a = 1 + 2j
>>> b = 3 - 4j

>>> a + b
(4-2j)

>>> a - b
(-2+6j)

>>> a * b
(11+2j)

>>> a ** b
(932.1391946432212+95.9465336603419j)

>>> a / b
(-0.2+0.4j)

>>> a // b
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: can't take floor of complex number.
```

Interestingly, although not surprising from a mathematical point of view, `int` and `float` objects also have the `.real` and `.imag` properties, as well as the `.conjugate()` method:

```
>>> x = 42
>>> x.real
42
>>> x.imag
0
>>> x.conjugate()
42

>>> y = 3.14
>>> y.real
3.14
>>> y.imag
0.0
>>> y.conjugate()
3.14
```

For floats and integers, `.real` and `.conjugate()` always return the number itself, and `.imag` always returns `0`. One thing to notice, however, is that `n.real` and `n.imag` return an integer if `n` is an integer, and a float if `n` is a float.

Now that you have seen the basics of complex numbers, you might be wondering when you would ever need to use them. If you are learning Python for web development or automation, the truth is you may *never* need to use complex numbers.

On the other hand, complex numbers are important in domains such as scientific computing and computer graphics. If you ever work in those domains, you may find Python's built-in support for complex numbers useful.

A detailed look at those topics is beyond the scope of this book. However, you will get an introduction to the NumPy package, a common tool for scientific computing with Python, in Chapter 16.

Leave feedback on this section »

5.8 Summary and Additional Resources

In this chapter you learned all about working with numbers in Python. You saw that there are two basic types of numbers—integers and floating-point numbers—and that Python also has built-in support for complex numbers.

First you learned how to do basic arithmetic with numbers using the `+`, `-`, `*`, `/`, and `%` operators. You saw how to write arithmetic expressions, and learned what the best practices are in PEP 8 for formatting arithmetic expressions in your code.

Then you learned about floating-point numbers and how they may not always be 100% accurate. This limitation has nothing to do with Python. It is a fact of modern-day computing and is due to the way floating-point numbers are stored in a computer's memory.

Next you saw how to round numbers to a given decimal place with the `round()` function, and learned that `round()` rounds ties to even, which is different from the way most people learned to round numbers in school.

Finally, you saw numerous ways to format numbers for display.

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-5

Additional Resources

To learn more, check out these resources:

- Basic Data Types in Python
- How to Round Numbers in Python
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 6

Functions and Loops

Functions are the building blocks of almost every Python program. They're where the real action takes place!

You've already seen how to use several functions, including `print()`, `len()`, and `round()`. These are all **built-in functions** because they come built into the Python language itself. You can also create **user-defined functions** that perform specific tasks.

Functions break code into smaller chunks, and are great for tasks that a program uses repeatedly. Instead of writing the same code each time the program need to perform the task, just call the function!

But sometimes you need to repeat some code several times in a row, and this is where **loops** come in.

In this chapter, you will learn:

- How to create user-defined functions
- How to write `for` and `while` loops
- What scope is and why it is important

Let's dive in!

Leave feedback on this section »

6.1 What is a Function, Really?

In the past few chapters you used functions like `print()` and `len()` to display text and determine the length of a string. But what is a function, really?

In this section you'll take a closer look at `len()` to learn more about what a function is and how it is executed.

Functions Are Values

One of the most important properties of a function in Python is that functions are values and can be assigned to a variable.

In IDLE's interactive window, inspect the name `len` by typing the following in at the prompt:

```
>>> len
<built-in function len>
```

When you hit `Enter`, Python tells you that the name `len` is a variable whose value is a built-in function.

Just like integer values have a type called `int`, and strings have a type `str`, function values also have a type:

```
>>> type(len)
<class 'builtin_function_or_method'>
```

Like any other variable, you can assign any value you want to `len`:

```
>>> len = "I'm not the len you're looking for."
>>> len
"I'm not the len you're looking for."
```

Now `len` has a string value, and you can verify that the type is `str` with `type()`:

```
>>> type(len)
<class 'str'>
```

The variable name `len` is a keyword in Python, and even though you can change it's value, it's usually a bad idea to do so. Changing the value of `len` can make your code confusing because it's easy to mistake the new `len` for the built-in function.

Important

If you typed in the previous code examples, **you no longer have access to the built-in `len` function in IDLE**.

You can get it back with the following code:

```
>>> del len
```

The `del` keyword is used to un-assign a variable from a value. `del` stands for delete, but it doesn't delete the value. Instead, it detaches the name from the value and deletes the name.

Normally, after using `del`, trying to use the deleted variable name raises a `NameError`. In this case, however, the name `len` doesn't get deleted:

```
>>> len
<built-in function len>
```

Because `len` is a built-in function name, it gets reassigned to the original function value.

By going through each of these steps, we've seen that a function's name is separate from the function itself.

How Python Executes Functions

Now let's take a closer look at how Python executes a function.

The first thing to notice is that you can't execute a function by just typing its name. You must **call** the function to tell Python to actually execute it.

Let's look at how this works with `len()`:

```
>>> # Typing just the name doesn't execute the function.
>>> # IDLE inspects the variable as usual.
>>> len
<built-in function len>

>>> # Use parentheses to call the function.
>>> len()
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    len()
TypeError: len() takes exactly one argument (0 given)
```

In this example, Python raises a `TypeError` when `len()` is called because `len()` expects an argument.

An **argument** is a value that gets **passed** to the function as input. Some functions can be called with no arguments, and some can take as many arguments as you like. `len()` requires exactly one argument.

When a function is done executing, it **returns** a value as output. The return value usually — but not always — depends on the values of any arguments passed to the function.

The process for executing a function can be summarized in three steps:

1. The function is **called**, and any arguments are passed to the function as input.
2. The function **executes**, and some action is performed with the arguments.
3. The function **returns**, and the original function call is replaced with the return value.

Let's look at this in practice and see how Python executes the following line of code:

```
>>> num_letters = len("four")
```

First, `len()` is called with the argument `"four"`. The length of the string `"four"` is calculated, which is the number `4`. Then `len()` returns the number `4` and replaces the function call with the value.

So, after the function executes, the line of code looks like this:

```
>>> num_letters = 4
```

Then Python assigns the value `4` to `num_letters` and continues executing any remaining lines of code in the program.

Functions Can Have Side Effects

You've learned how to call a function and that they return a value when they are done executing. Sometimes, though, functions do more than just return a value.

When a function changes or affects something external to the function itself, it is said to have a **side effect**. You have already seen one function with a side effect: `print()`.

When you call `print()` with a string argument, the string is displayed in the Python shell as text. But `print()` doesn't return any text as a value.

To see what `print()` returns, you can assign the return value of `print()` to a variable:

```
>>> return_value = print("What do I return?")
What do I return?
>>> return_value
>>>
```

When you assign `print("What do I return?")` to `return_value`, the string

"What do I return?" is displayed. However, when you inspect the value of return_value, nothing is is shown.

print() returns a special value called None that indicates the absence of data. None has a type called NoneType:

```
>>> type(return_value)
<class 'NoneType'>
>>> print(return_value)
None
```

When you call print(), the text that gets displayed is not the return value. It is a side effect of print().

Now that you know that functions are values, just like strings and numbers, and have learned how functions are called and executed, let's take a look at how you can create your own user-defined functions.

Leave feedback on this section »

6.2 Write Your Own Functions

As you write longer and more complex programs, you may find that you need to use the same few lines of code repeatedly. Or maybe you need to calculate the same formula with different values several times in your code.

You might be tempted to copy and paste similar code to other parts of your program and modify it as needed, but this is usually a bad idea!

Repetitive code can be a nightmare to maintain. If you find a mistake in some code that's been copied and pasted all over the place, you'll end up having to apply the fix everywhere the code was copied. That's a lot of work, and you might miss a spot!

In this section, you'll learn how to define your own functions so that you can avoid repeating yourself when you need to reuse code. Let's go!

The Anatomy of a Function

Every function has two parts:

1. The **function signature** defines the name of the function and any inputs it expects.
2. The **function body** contains the code that runs every time the function is used.

Let's start by writing a function that takes two numbers as input and returns their product. Here's what this function might look like, with the signature, body, and return statement identified with comments:

```python
def multiply(x, y):  # Function signature
    # Function body
    product = x * y
    return product
```

It might seem odd to make a function for something as simple as the `*` operator. In fact, `multiply` is not a function you would probably write in a real-world scenario. But it makes a great first example for understanding how functions are created!

Let's break the function down to see what's going on.

The Function Signature

The first line of code in a function is called the **function signature**. It always starts with the `def` keyword, which is short for "define."

Let's look more closely at the signature of the `multiply` function:

```python
def multiply(x, y):
```

The function signature has four parts:

1. The `def` keyword

2. The function name, `multiply`
3. The parameter list, `(x, y)`
4. A colon (`:`) at the end of the line

When Python reads a line beginning with the `def` keyword, it creates a new function. The function is assigned to a variable with the same name as the function name.

> **Note**
>
> Since function names become variables, they must follow the same rules for variable names that you learned in Chapter 3.
>
> So, a function name can only contain numbers, letters, and underscores, and must not begin with a number.

The parameter list is a list of parameter names surrounded by opening and closing parentheses. It defines the function's expected inputs. `(x, y)` is the parameter list for the `multiply` function. It creates two parameters, `x` and `y`.

A **parameter** is sort of like a variable, except that it has no value. It is a placeholder for actual values that are provided whenever the function is called with one or more arguments.

Code in the function body can use parameters as if they are variables with real values. For example, the function body may contain a line of code with the expression `x * y`.

Since `x` and `y` have no value, `x * y` has no value. Python saves the expression as a template and fills in the missing values when the function is executed.

A function can have any number of parameters, including no parameters at all!

The Function Body

The **function body** is the code that gets run whenever the function is used in your program. Here's the function body for the multiply function:

```python
def multiply(x, y):
    # Function body
    product = x * y
    return product
```

multiply is a pretty simple function. It's body has only two lines of code!

The first line creates a variable called product and assigns to it the value x * y. Since x and y have no values yet, this line is really a template for the value product is assigned when the function is executed.

The second line of code is called a **return statement**. It starts with the return keyword and is followed by the variable product. When Python reaches the return statement, it stops running the function and returns the value of product.

Notice that both lines of code in the function body are indented. This is vitally important! Every line that is indented below the function signature is understood to be part of the function body.

For instance, the print() function in the following example is not a part of the function body because it is not indented:

```python
def multiply(x, y):
    product = x * y
    return product


print("Where am I?")  # Not in the function body.
```

If print() is indented, then it becomes a part of the function body even if there is a blank line between print() and the previous line:

```python
def multiply(x, y):
    product = x * y
    return product

    print("Where am I?")  # In the function body.
```

There is one rule that you must follow when indenting code in a function's body. Every line must be indented by the same number of spaces.

Try saving the following code to a file called `multiply.py` and running it from IDLE:

```python
def multiply(x, y):
    product = x * y
     return product  # Indented with one extra space.
```

IDLE won't run the code! A dialog box appears with the error "unexpected indent." Python wasn't expecting the return statement to be indented differently than the line before it.

Another error occurs when a line of code is indented less than the line above it, but the indentation doesn't match any previous lines. Modify the `multiply.py` file to look like this:

```python
def multiply(x, y):
     product = x * y
    return product  # Indented less than previous line.
```

Now save and run the file. IDLE stops it with the error "unindent does not match any outer indentation level." The return statement isn't indented with the same number of spaces as any other line in the function body.

> **Note**
>
> Although Python has no rules for the number of spaces used to indent code in a function body, PEP 8 recommends indenting with four spaces.
>
> We follow this convention throughout this book.

Once Python executes a `return` statement, the function stops running and returns the value. If any code appears below the `return` statement that is indented so as to be part of the function body, it will never run.

For instance, the `print()` function will never be executed in the following function:

```python
def multiply(x, y):
    product = x * y
    return product
    print("You can't see me!")
```

If you call this version of `multiply()`, you will never see the string `"You can't see me!"` displayed.

Calling a User-Defined Function

You call a user-defined function just like any other function. Type the function name followed by a list of arguments in between parentheses.

For instance, to call `multiply()` with the argument `2` and `4`, just type:

```python
multiply(2, 4)
```

Unlike built-in functions, user-defined functions are not available until they have been defined with the `def` keyword. You must define the function before you call it.

Try saving and running the following script:

```
num = multiply(2, 4)
print(num)

def multiply(x, y):
    product = x * y
    return product
```

When Python reads the line `num = multiply(2, 4)`, it doesn't recognize the name `multiply` and raises a `NameError`:

```
Traceback (most recent call last):
  File "C:Usersdaveamultiply.py", line 1, in <module>
    num = multiply(2, 4)
NameError: name 'multiply' is not defined
```

To fix the error, move the function definition to the top of the file:

```
def multiply(x, y):
    product = x * y
    return product

num = multiply(2, 4)
print(num)
```

Now when you save and run the script, the value `8` is displayed in the interactive window.

Documenting Your Functions

To get help with a function in IDLE's interactive window, you can use the `help()` function:

```
>>> help(len)
Help on built-in function len in module builtins:

len(obj, /)
    Return the number of items in a container.
```

When you pass a variable name or function name to `help()`, it displays some useful information about it. In this case, `help()` tells you that `len` is a built-in function that returns the number of items in a container.

Note

A **container** is a special name for an object that contains other objects. A string is a container because it contains characters.

You will learn about other container types in Chapter 9.

Let's see what happens when you call `help()` on the `multiply()` function:

```
>>> help(multiply)
Help on function multiply in module __main__:

multiply(x, y)
```

`help()` displays the function signature, but there isn't any information about what the function does. To better document `multiply()`, we need to provide a docstring.

A **docstring** is a triple-quoted string literal placed at the top of the function body. Docstrings are used to document what a function does and what kinds of parameters it expects.

Here's what `multiply()` looks like with a docstring added to it:

```
def multiply(x, y):
    """Return the product of two numbers x and y."""
    product = x * y
    return product
```

Update the `multiply.py` script with the docstring, then save and run the script. Now you can use `help()` in the interactive window to see the docstring:

```
>>> help(multiply)
Help on function multiply in module __main__:

multiply(x, y)
    Return the product of two numbers x and y.
```

PEP 8 doesn't say much about docstrings, except that every function should have one.

There are a number of standardized docstring formats, but we won't get into them here. Some general guidelines for writing docstrings can be found in PEP 257.

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 function called `cube()` with one number parameter and returns the value of that number raised to the third power. Test the function by displaying the result of calling your `cube()` function on a few different numbers.
2. Write a function called `greet()` that takes one string parameter called `name` and displays the text `"Hello <name>!"`, where `<name>` is replaced with the value of the `name` parameter.

Leave feedback on this section »

6.3 Challenge: Convert Temperatures

Write a script called `temperature.py` that defines two functions:

1. `convert_cel_to_far()` which takes one `float` parameter representing degrees Celsius and returns a float representing the same temperature in degrees Fahrenheit using the following formula:

   ```
   F = C * 9/5 + 32
   ```

2. `convert_far_to_cel()` which take one `float` parameter representing degrees Fahrenheit and returns a float representing the same temperature in degrees Celsius using the following formula:

   ```
   C = (F - 32) * 5/9
   ```

The script should first prompt the user to enter a temperature in degrees Fahrenheit and then display the temperature converted to Celsius.

Then prompt the user to enter a temperature in degrees Celsius and display the temperature converted to Fahrenheit.

All converted temperatures should be rounded to 2 decimal places.

Here's a sample run of the program:

```
Enter a temperature in degrees F: 72
72 degress F = 22.22 degrees C

Enter a temperature in degrees C: 37
37 degrees C = 98.60 degrees F
```

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

Leave feedback on this section »

6.4 Run in Circles

One of the great things about computers is that you can make them do the same thing over and over again, and they rarely complain or get tired.

A **loop** is a block of code that gets repeated over and over again either a specified number of times or until some condition is met. There are two kinds of loops in Python: **`while` loops** and **`for` loops**. In this section, you'll learn how to use both.

Let's start by looking at how `while` loops work.

The `while` Loop

`while` loops repeat a section of code while some condition is true. There are two parts to every `while` loop:

1. The **`while` statement** starts with the `while` keyword, followed by a **test condition**, and ends with a colon (:).
2. The **loop body** contains the code that gets repeated at each step of the loop. Each line is indented four spaces.

When a `while` loop is executed, Python evaluates the test condition and determines if it is true or false. If the test condition is true, then the code in the loop body is executed. Otherwise, the code in the body is skipped and the rest of the program is executed.

If the test condition is true and the body of the loop is executed, then once Python reaches the end of the body, it returns to the `while` statement and re-evaluates the test condition. If the test condition is still true, the body is executed again. If it is false, the body is skipped.

This process repeats over and over until the test condition fails, causing Python to loop over the code in the body of the `while` loop.

Let's look at an example. Type the following code into the interactive window:

```
>>> n = 1
>>> while n < 5:
...     print(n)
...     n = n + 1
...
1
2
3
4
```

First, the integer `1` is assigned to the variable `n`. Then a `while` loop is created with the test condition `n < 5`, which checks whether or not the value of `n` is less than `5`.

If `n` is less than `5`, the body of the loop is executed. There are two lines of code in the loop body. In the first line, the value of `n` is printed on the screen, and then `n` is **incremented** by `1` in the second line.

The loop execution takes place in five steps, described in the following table:

Step #	Value of `n`	Test Condition	What Happens
1	`1`	`1 < 5` (true)	`1` printed; `n` incremented to `2`
2	`2`	`2 < 5` (true)	`2` printed; `n` incremented to `3`
3	`3`	`3 < 5` (true)	`3` printed; `n` incremented to `4`
4	`4`	`4 < 5` (true)	`4` printed; `n` incremented to `5`
5	`5`	`5 < 5` (false)	Nothing printed; loop ends.

If you aren't careful, you can create an **infinite loop**. This happens when the test condition is always true. An infinite loop never terminates. The loop body keeps repeating forever.

Here's an example of an infinite loop:

```
>>> n = 1
>>> while n < 5:
...     print(n)
...
```

The only difference between this `while` loop and the previous one is that `n` is never incremented in the loop body. At each step of the loop, `n` is equal to `1`. That means the test condition `n < 5` is always true, and the number `1` is printed over and over again forever.

Note

Infinite loops aren't inherently bad. Sometimes they are exactly the kind of loop you need.

For example, code that interacts with hardware may use an infinite loop to constantly check whether or not a button or switch has been activated.

If you run a program that enters an infinite loop, you can force Python to quit by pressing `Ctrl+C`. Python stops running the program and raises a `KeyboardInterrupt` error:

```
Traceback (most recent call last):
  File "<pyshell#8>", line 2, in <module>
    print(n)
KeyboardInterrupt
```

Let's look at an example of a `while` loop in practice. One use of a `while` loop is to check whether or not user input meets some condition and, if not, repeatedly ask the user for new input until valid input is received.

For instance, the following program continuously asks a user for a positive number until a positive number is entered:

```python
num = float(input("Enter a positive number: "))

while num <= 0:
    print("That's not a positive number!")
    num = float(input("Enter a positive number: "))
```

First, the user is prompted to enter a positive number. The test condition `num <= 0` determines whether or not `num` is less than or equal to `0`.

If `num` is positive, then the test condition fails. The body of the loop is skipped and the program ends.

Otherwise, if `num` is `0` or negative, the body of the loop executes. The user is notified that their input was incorrect, and they are prompted again to enter a positive number.

`while` loops are perfect for repeating a section of code while some condition is met. They aren't well-suited, however, for repeating a section of code a specific number of times.

The `for` Loop

A `for` loop executes a section of code once for each item in a collection of items. The number of times that the code is executed is determined by the number of items in the collection.

Like its `while` counterpart, the `for` loop has two main parts:

1. The **`for` statement** begins with the `for` keyword, followed by a **membership expression**, and ends in a colon (:).
2. The **loop body** contains the code to be executed at each step of the loop, and is indented four spaces.

Let's look at an example. The following `for` loop prints each letter of the string `"Python"` one at a time:

```
for letter in "Python":
    print(letter)
```

In this example, the `for` statement is `for letter in "Python"`. The membership expression is `letter in "Python"`.

At each step of the loop, the variable `letter` is assigned the next letter in the string `"Python"`, and then the value of `letter` is printed.

The loops runs once for each character in the string `"Python"`, so the loop body executes six times. The following table summarizes the execution of this `for` loop:

Step #	Value of `letter`	What Happens
1	`"P"`	`P` is printed

Step #	Value of `letter`	What Happens
2	`"y"`	`y` is printed
3	`"t"`	`t` is printed
4	`"h"`	`h` is printed
5	`"o"`	`o` is printed
6	`"n"`	`n` is printed

To see why `for` loops are better for looping over collections of items, let's re-write the `for` loop in previous example as a `while` loop.

To do so, we can use a variable to store the index of the next character in the string. At each step of the loop, we'll print out the character at the current index and then increment the index.

The loop will stop once the value of the index variable is equal to the length of the string. Remember, indices start at `0`, so the last index of the string `"Python"` is `5`.

Here's how you might write that code:

```python
word = "Python"
index = 0

while index < len(word):
    print(word[index])
    index = index + 1
```

That's significantly more complex than the `for` loop version!

Not only is the `for` loop less complex, the code itself looks more natural. It more closely resembles how you might describe the loop in English.

Note

You may sometimes hear people describe some code as being particularly "Pythonic." The term **Pythonic** is generally used to describe code that is clear, concise, and uses Python's built-in features to its advantage.

In these terms, using a `for` loop to loop over a collection of items is more Pythonic than using a `while` loop.

Sometimes it's useful to loop over a range of numbers. Python has a handy built-in function `range()` that produces just that — a range of numbers!

For example, `range(3)` returns the range of integers starting with `0` and up to, but not including, `3`. That is, `range(3)` is the range of numbers `0`, `1`, and `2`.

You can use `range(n)`, where `n` is any positive number, to execute a loop exactly `n` times. For instance, the following `for` loop prints the string `"Python"` three times:

```
for n in range(3):
    print("Python")
```

You can also give a range a starting point. For example, `range(1, 5)` is the range of numbers `1`, `2`, `3`, and `4`. The first argument is the starting number, and the second argument is the endpoint, which is not included in the range.

Using the two-argument version of `range()`, the following `for` loop prints the square of every number starting with `10` and up to, but not including, `20`:

```
for n in range(10, 20):
    print(n * n)
```

Let's look at a practical example. The following program asks the user to input an amount and then displays how to split that amount between 2, 3, 4, and 5 people:

```
amount = float(input("Enter an amount: "))

for num_people in range(2, 6):
    print(f"{num_people} people: ${amount / num_people:.2f} each")
```

The `for` loop loops over the number `2`, `3`, `4`, and `5`, and prints the number of people and the amount each person should pay. The formatting specifier `,.2f` is used to format the amount as fixed-point number rounded to two decimal places and commas every three digits.

Running the program with the input `10` produces the following output:

```
Enter an amount: 10
2 people: $5.00 each
3 people: $3.33 each
4 people: $2.50 each
5 people: $2.00 each
```

`for` loops are generally used more often than `while` loops in Python. Most of the time, a `for` loop is more concise and easier to read than an equivalent `while` loop.

Nested Loops

As long as you indent the code correctly, you can even put loops inside of other loops.

Type the following into IDLE's interactive window:

```
for n in range(1, 4):
    for j in range(4, 7):
        print(f"n = {n} and j = {j}")
```

When Python enters the body of the first `for` loop, the variable `n` is assigned the value `1`. Then the body of the second `for` loop is executed and `j` is assigned the value `4`. The first thing printed is `n = 1 and j = 4`.

After executing the `print()` function, Python returns to the *inner* `for` loop, assigns to `j` the value of `5`, and then prints `n = 1 and j = 5`. Python doesn't return the outer `for` loop because the inner `for` loop, which is inside the body of the outer `for` loop, isn't done executing.

Next, `j` is assigned the value `6` and Python prints `n = 1 and j = 6`. At this point, the inner `for` loop is done executing, so control returns to the outer `for` loop.

The variable `n` gets assigned the value `2`, and the inner `for` loop executes a second time. That is, `j` is assigned the value `4` and `n = 2 and j = 4` is printed to the console.

The two loops continue to execute in this fashion, and the final output looks like this:

```
n = 1 and j = 4
n = 1 and j = 5
n = 1 and j = 6
n = 2 and j = 4
n = 2 and j = 5
n = 2 and j = 6
n = 3 and j = 4
n = 3 and j = 5
n = 3 and j = 6
```

A loop inside of another loop is called a **nested loop**, and they come up more often than you might expect. You can nest `while` loops inside of `for` loops, and vice versa, and even nest loops more than two levels deep!

Important

Nesting loops inherently increases the complexity of your code, as you can see by the dramatic increase in the number of steps run in the previous example compared to examples with a single `for` loop.

Using nested loops is sometimes the only way to get something done, but too many nested loops can have a negative effect on a program's performance.

Loops are a powerful tool. They tap into one of the greatest advantages computers provide as tools for computation: the ability to repeat the same task a vast number of times without tiring and without complaining.

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 `for` loop that prints out the integers 2 through 10, each on a new line, by using the `range()` function.

2. Use a `while` loop that prints out the integers 2 through 10 (Hint: You'll need to create a new integer first.)

3. Write a function called `doubles()` that takes one number as its input and doubles that number. Then use the `doubles()` function in a loop to double the number `2` three times, displaying each result on a separate line. Here is some sample output:

```
4
8
16
```

Leave feedback on this section »

6.5 Challenge: Track Your Investments

In this challenge, you will write a program called `invest.py` that tracks the growing amount of an investment over time.

An initial deposit, called the principal amount, is made. Each year, the amount increases by a fixed percentage, called the annual rate of return.

For example, a principal amount of $100 with an annual rate of return of 5% increases the first year by $5. The second year, the increase is 5% of the new amount $105, which is $5.25.

Write a function called `invest` with three parameters: the principal amount, the annual rate of return, and the number of years to calculate. The function signature might look something like this:

```python
def invest(amount, rate, years):
```

The function then prints out the amount of the investment, rounded to 2 decimal places, at the end of each year for the specified number of years.

For example, calling `invest(100, .05, 4)` should print the following:

```
year 1: $105.00
year 2: $110.25
year 3: $115.76
year 4: $121.55
```

To finish the program, prompt the user to enter an initial amount, an annual percentage rate, and a number of years. Then call `invest()` to display the calculations for the values entered by the user.

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

Leave feedback on this section »

6.6 Understand Scope in Python

Any discussion of functions and loops in Python would be incomplete without some mention of the issue of **scope**.

Scope can be one of the more difficult concepts to understand in programming, so in this section you will get a gentle introduction to it.

By the end of this section, you'll know what a **scope** is and why it is important. You will also learn the LEGB rule for **scope resolution**.

What Is a Scope?

When you assign a value to a variable, you are giving that value a name. Names are unique. For example, you can't assign the same name to two different numbers.

```
>>> x = 2
>>> x
2

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

When you assign `3` to `x`, you can no longer recall the value `2` with the name `x`.

This behavior makes sense. After all, if the variable `x` has the values `2` and `3` simultaneously, how do you evaluate `x + 2`? Should it be `4` or `5`?

As it turns out, there *is* a way to assign the same name to two different values. Try running the following script:

```
x = "Hello World"

def func():
    x = 2
```

```python
    print(f"Inside 'func', x has the value {x}")

func()
print(f"Outside 'func', x has the value {x}")
```

In this example, the variable `x` is assigned two different values. `x` is assigned `"Hello, World"` at the beginning, and is assigned `2` inside of `func()`.

The output of the script, which you might find surprising, looks like this:

```
Inside 'func', x has the value 2
Outside 'func', x has the value Hello World
```

How does `x` still have the value `"Hello World"` after calling `func()`, which changes the value of `x` to `2`?

The answer is that the function `func()` has a different **scope** than the code that exists outside of the function. That is, you can name an object inside `func()` the same name as something outside `func()` and Python can keep the two separated.

The function body has what is known as a **local scope**, with its own set of names available to it. Code outside of the function body is in the **global scope**.

You can think of a scope as a set of names mapped to objects. When you use a particular name in your code, such as a variable or a function name, Python checks the current scope to determine whether or not that name exists.

Scope Resolution

Scopes have a hierarchy. For example, consider the following:

```python
x = 5
```

```python
def outer_func():
    y = 3

    def inner_func():
        z = x + y
        return z

    return inner_func()
```

Note

The `inner_func()` function is called an **inner function** because it is defined inside of another function. Just like you can nest loops, you can also define functions within other functions!

You can read more about inner functions in Real Python's article Inner Functions—What Are They Good For?.

The variable `z` is in the local scope of `inner_func()`. When Python executes the line `z = x + y`, it looks for the variables `x` and `y` in the local scope. However, neither of them exist there, so it moves up to the scope of the `outer_func()` function.

The scope for `outer_func()` is an **enclosing** scope of `inner_func()`. It is not quite the global scope, and is not the local scope for `inner_func()`. It lies in between those two.

The variable `y` is defined in the scope for `outer_func()` and is assigned the value `3`. However, `x` does not exist in this scope, so Python moves up once again to the global scope. There it finds the name `x`, which has the value `5`. Now that the names `x` and `y` are resolved, Python can execute the line `z = x + y`, which assigns to `z` the value of `8`.

The LEGB Rule

A useful way to remember how Python resolves scope is with the **LEGB** rule. This rule is an acronym for **L**ocal, **E**nclosing, **G**lobal, **B**uilt-in.

Python resolves scope in the order in which each scope appears in the list LEGB. Here is a quick overview to help you remember how all of this works:

Local (L): The local, or current, scope. This could be the body of a function or the top-level scope of a script. It always represents the scope that the Python interpreter is currently working in.

Enclosing (E): The enclosing scope. This is the scope one level up from the local scope. If the local scope is an inner function, the enclosing scope is the scope of the outer function. If the scope is a top-level function, the enclosing scope is the same as the global scope.

Global (G): The global scope, which is the top-most scope in the script. This contains all of the names defined in the script that are not contained in a function body.

Built-in (B): The built-in scope contains all of the names, such as keywords, that are built-in to Python. Functions such as `round()` and `abs()` are in the built-in scope. Anything that you can use without first defining yourself is contained in the built-in scope.

Break the Rules

Consider the following script. What do you think the output is?

```python
total = 0

def add_to_total(n):
    total = total + n

add_to_total(5)
print(total)
```

You would think that script outputs the value `5`, right? Try running it to see what happens.

Something unexpected occurs. You get an error!

```
Traceback (most recent call last):
  File "C:/Users/davea/stuff/python/scope.py", line 6, in <module>
    add_to_total(5)
  File "C:/Users/davea/stuff/python/scope.py", line 4, in add_to_total
    total = total + n
UnboundLocalError: local variable 'total' referenced before assignment
```

Wait a minute! According to the LEGB rule, Python should have recognized that the name `total` doesn't exist in the `add_to_total()` function's local scope and moved up to the global scope to resolve the name, right?

The problem here is that the script attempts to make an assignment to the variable `total`, which creates a new name in the local scope. Then, when Python executes the right-hand side of the assignment it finds the name `total` in the local scope with nothing assigned to it yet.

These kinds of errors are tricky and are one of the reasons it is best to use unique variable and function names no matter what scope you are in.

You can get around this issue with the `global` keyword. Try running the following altered script:

```python
total = 0

def add_to_total(n):
    global total
    total = total + n

add_to_total(5)
print(total)
```

This time, you get the expected output `5`. Why's that?

The line `global total` tells Python to look in the global scope for the name `total`. That way, the line `total = total + n` does not create a new local variable.

Although this “fixes” the script, the use of the `global` keyword is considered bad form in general.

If you find yourself using `global` to fix problems like the one above, stop and think if there is a better way to write your code. Often, you’ll find that there is!

Leave feedback on this section »

6.7 Summary and Additional Resources

In this chapter, you learned about two of the most essential concepts in programming: functions and loops.

First, you learned how to define your own custom functions. You saw that functions are made up of two parts:

1. The **function signature**, which starts with the `def` keyword and includes the name of the function and the function’s parameters
2. The **function body**, which contains the code that runs whenever the function is called.

Functions help avoid repeating similar code throughout a program by creating re-usable components. This helps make code easier to read and maintain.

Then you learned about Python’s two kinds of loops:

1. **`while` loops** repeat some code while some condition remains true
2. **`for` loops** repeat some code for each element in a set of objects

Finally, you learned what a **scope** is and how Python resolves scope using the LEGB rule.

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-6

Additional Resources

To learn more about functions and loops, check out the following resources:

- Python “while” Loops (Indefinite Iteration)
- Python “for” Loops (Definite Iteration)
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 7

Finding and Fixing Code Bugs

Everyone makes mistakes—even seasoned professional developers!

IDLE is pretty good at catching mistakes like syntax and run-time errors, but there's a third type of error that you may have already experienced. **Logic errors** occur when an otherwise valid program doesn't do what was intended.

Logic errors cause unexpected behaviors called **bugs**. Removing bugs is called **debugging**, and a **debugger** is a tool that helps you hunt down bugs and understand why they are happening.

Knowing how to find and fix bugs in your code is a skill that you will use for your entire coding career!

In this chapter, you will:

- Learn how to use IDLE's Debug Control Window
- Practice debugging on a buggy function

Let's go!

Leave feedback on this section »

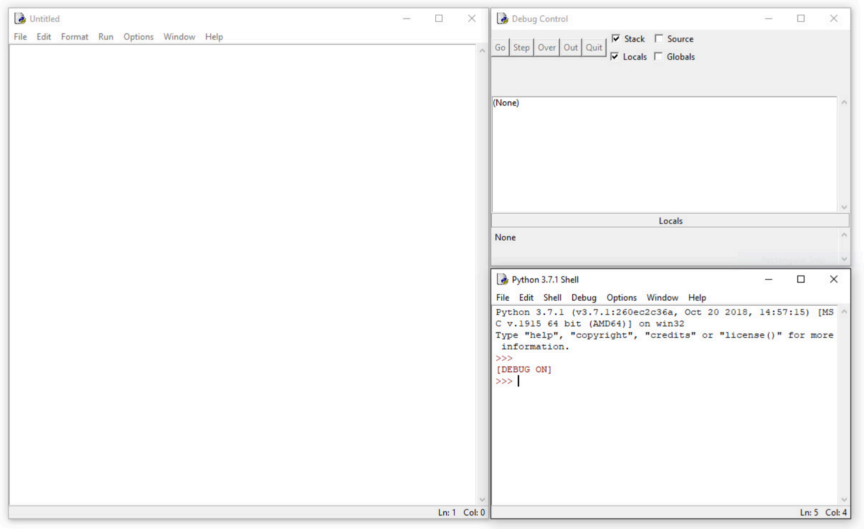
7.1 Use the Debug Control Window

The main interface to IDLE's debugger is through the Debug Control Window, which we'll refer to as the Debug window for short. You can open the Debug window by selecting *Debug › Debugger* from the menu in the interactive window. Go ahead and open the Debug window.

Note

If the *Debug* menu is missing from your menu bar, make sure the interactive window is in focus by clicking that window.

Open a new script window and arrange the three windows on your screen so that you can see all of them simultaneously. Here's one way you could rearrange the windows:



```
Python 3.7.1 (v3.7.1:260ec2c36a, Oct 20 2018, 14:57:15) [MS
C v.1915 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more
 information.
>>>
[DEBUG ON]
>>> 
```

Note

Whenever the Debug window is open, the prompt in the interactive window has `[DEBUG ON]` next to it to indicate that the debugger is open.

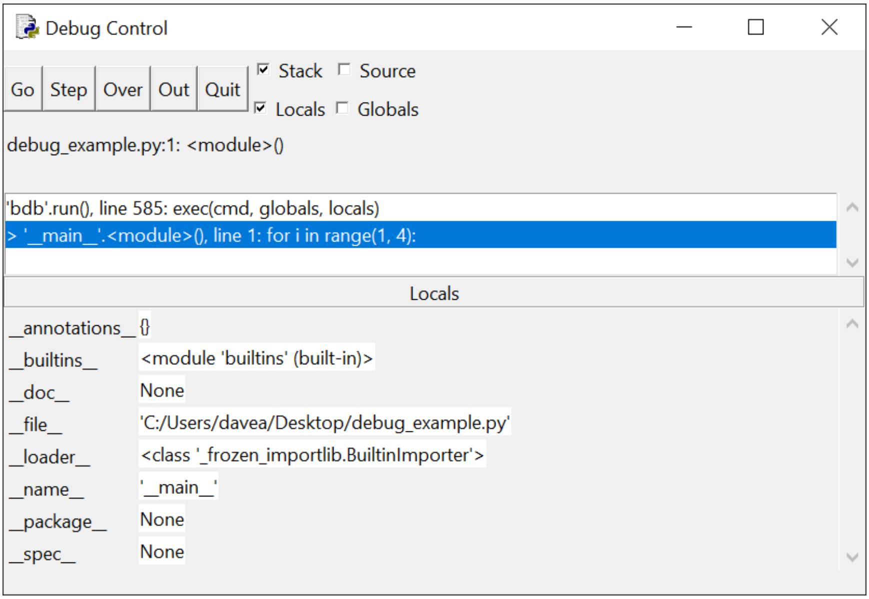
In this section you'll learn how the Debug window is organized, how to step through your code with the debugger one line at a time, and how to set breakpoints to help speed up the debugging process.

The Debug Control Window: An Overview

To see how the debugger works, let's start by writing a simple program without any bugs. Type the following into the script window:

```python
for i in range(1, 4):
    j = i * 2
    print(f"i is {i} and j is {j}")
```

When you save and run this script with the Debug window open, you'll notice that execution doesn't get very far. The Debug Control window will look like this:



☑ Stack □ Source
☑ Locals □ Globals

debug_example.py:1: <module>()

```
'bdb'.run(), line 585: exec(cmd, globals, locals)
> '__main__'.<module>(), line 1: for i in range(1, 4):
```

Locals

__annotations__	{}
__builtins__	<module 'builtins' (built-in)>
__doc__	None
__file__	'C:/Users/davea/Desktop/debug_example.py'
__loader__	<class '_frozen_importlib.BuiltinImporter'>
__name__	'__main__'
__package__	None
__spec__	None

Notice that the *Stack* panel at the top of the window contains the following message:

```
> '__main__'.<module>(), line 1: for i in range(1, 4):
```

This tells you that line 1 (which contains the code `for i in range(1, 4):`) is *about* to be run but has not started yet. The `'__main__'.module()` part of the message in the debugger refers to the fact that we're currently in the "main" section of the script, as opposed to being, for example, in a function definition before the main block of code has been reached.

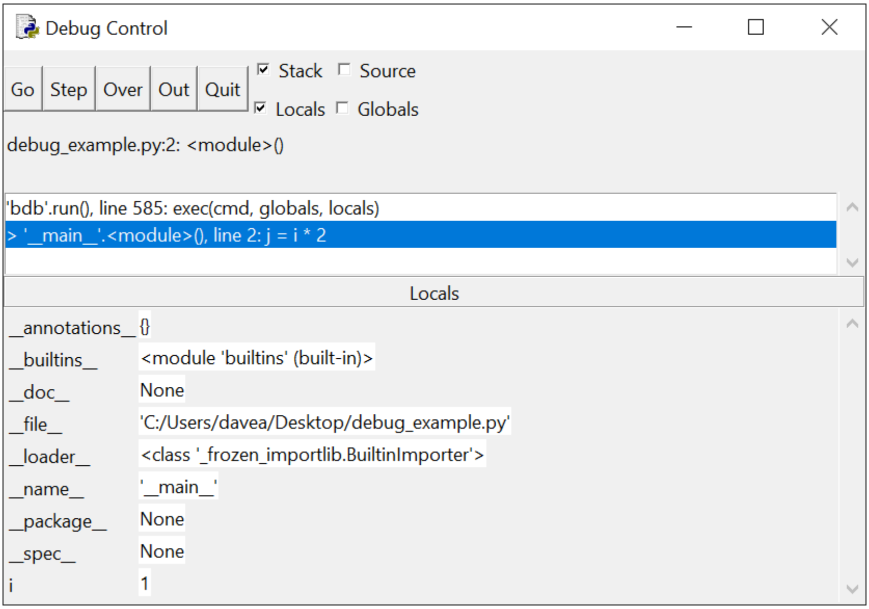
Below the *Stack* panel, there is a *Locals* panel that lists some strange looking stuff like `__annotations__`, `__builtins__`, `__doc__`, and so on. These are some internal system variables that you can ignore for now. As your program runs, you will see variables declared in the code displayed in this window so that you can keep track of their value.

There are five buttons located at the top left-hand corner of the Debug window: *Go*, *Step*, *Over*, *Out*, and *Quit*. These buttons control how the debugger moves through your code.

In the following sections, we'll explore what each of these buttons does, starting with the *Step* button.

The *Step* Button

Go ahead and click the *Step* button at the top left-hand corner of the Debug window. The Debug window changes a bit to look like this:



☑ Stack □ Source
☑ Locals □ Globals

```
debug_example.py:2: <module>()
'bdb'.run(), line 585: exec(cmd, globals, locals)
> '__main__'.<module>(), line 2: j = i * 2
```

Locals

__annotations__	{}
__builtins__	<module 'builtins' (built-in)>
__doc__	None
__file__	'C:/Users/davea/Desktop/debug_example.py'
__loader__	<class '_frozen_importlib.BuiltinImporter'>
__name__	'__main__'
__package__	None
__spec__	None
i	1

There are two differences to pay attention to here. First, the message in the *Stack* window changes to:

```
> '__main__'.<module>(), line 2: j = i * 2:
```

At this point, line 1 of your code was run, and the debugger has stopped just before executing line 2.

The second change to notice is the new variable `i` that is assigned the value `1` in the *Locals* panel. That's because the `for` loop in the first line of code created the variable `i` and assigned it the value `1`.

Continue hitting the *Step* button to walk through your code line by line, watching what happens in the debugger window. When you arrive at the line `print(f"i is {i} and j is {j}")`, you can see the output displayed in the interactive window one piece at a time.

More importantly, you can track the growing values of `i` and `j` as you step through the `for` loop. You can probably imagine how beneficial this feature is when trying to locate the source of bugs in your programs. Knowing each variables value at each line of code can help you pinpoint where things go wrong.

Breakpoints and the "Go" Button

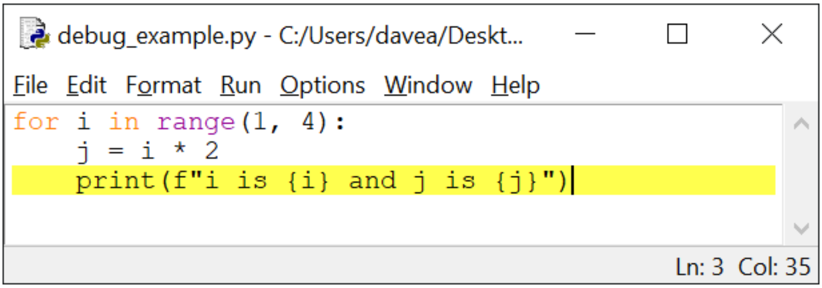
Often, you may know that the bug must be in a particular section of your code, but you may not know precisely where. Rather than clicking the *Step* button all day long, you can set a **breakpoint** that tells the debugger to run all code before the breakpoint continuously until the breakpoint is reached.

Breakpoints tell the debugger when to pause code execution so that you can take a look at the current state of the program. They don't actually break anything.

To set a breakpoint, right-click (Mac: `Ctrl+Click`) on the line of code in your script window you would like to pause at and select *Set Breakpoint*. IDLE highlights the line in yellow to indicate that your breakpoint has been set. You can remove a breakpoint at any time by right-clicking on the line with a breakpoint and selecting *Clear Breakpoint*.

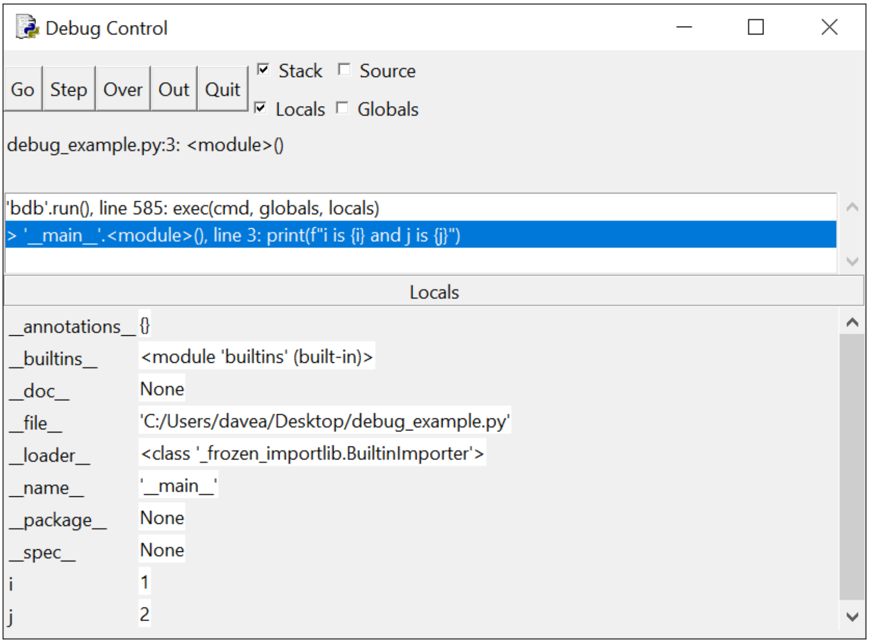
Go ahead and press the *Quit* button at the top of the Debug Control Window to turn off the debugger for now. This won't close the window, and you'll want to keep it open because you'll be using it again in just a moment.

Set a breakpoint on the line of code with the `print()` statement. The script window should now look like this:


debug_example.py - C:/Users/davea/Deskt...
File Edit Format Run Options Window Help
Ln: 3 Col: 35

```
for i in range(1, 4):
    j = i * 2
    print(f"i is {i} and j is {j}")
```

Now run the script by pressing `F5`. Just like before, the *Stack* panel of the Debug Control Window indicates that debugger has started and is waiting to execute line 1. This time, instead of clicking on the *Step* button, click on the *Go* button and watch what happens to the Debug window:



☑ Stack □ Source
☑ Locals □ Globals

```
debug_example.py:3: <module>()
```

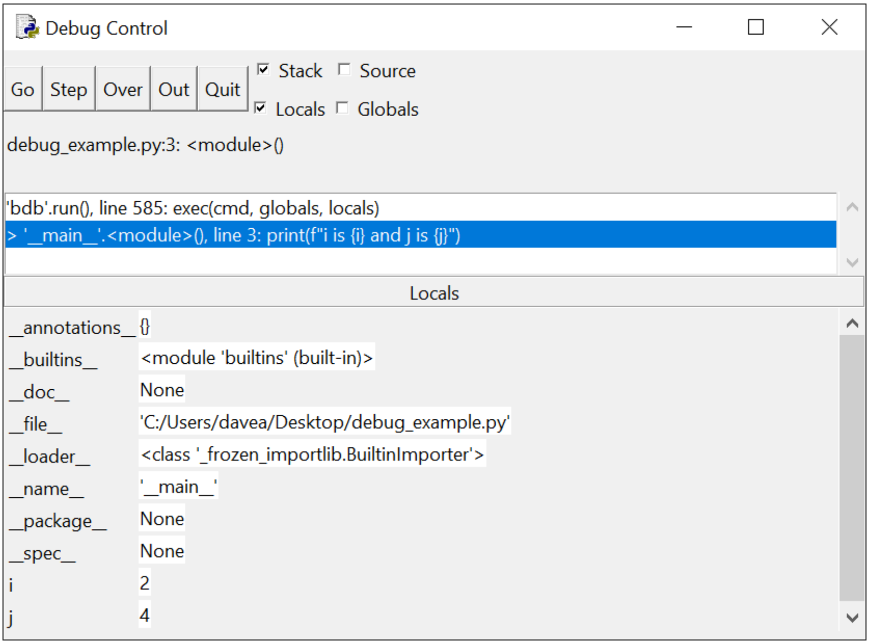
```
'bdb'.run(), line 585: exec(cmd, globals, locals)
> '__main__'.<module>(), line 3: print(f"i is {i} and j is {j}")
```

__annotations__	{}
__builtins__	<module 'builtins' (built-in)>
__doc__	None
__file__	'C:/Users/davea/Desktop/debug_example.py'
__loader__	<class '_frozen_importlib.BuiltinImporter'>
__name__	'__main__'
__package__	None
__spec__	None
i	1
j	2

The *Stack* panel now shows following message indicating that it is waiting to execute line 3:

```
> '__main__'.<module>(), line 3: print(f"i is {i} and j is {j}")
```

If you look at the *Locals* panel, you will see that both variable `i` and `j` have the values `1` and `2`, respectively. By clicking on *Go*, you told the debugger to run your code continuously until reaching either a breakpoint or the end of the program. Now press "Go" again. The Debug window now looks like this:



debug_example.py:3: <module>()

```
'bdb'.run(), line 585: exec(cmd, globals, locals)
> '__main__'.<module>(), line 3: print(f"i is {i} and j is {j}")
```

__annotations__	{}
__builtins__	<module 'builtins' (built-in)>
__doc__	None
__file__	'C:/Users/davea/Desktop/debug_example.py'
__loader__	<class '_frozen_importlib.BuiltinImporter'>
__name__	'__main__'
__package__	None
__spec__	None
i	2
j	4

Do you see what changed? The same message as before is displayed in the *Stack* panel, indicating the debugger is waiting to execute line 3 again. However, now the values of the variables `i` and `j` are `2` and `4`. The interactive window also displays the output from running the line with `print()` in it the first time.

Each time you press the *Go* button, the debugger runs the code continuously until the next breakpoint is reached. Since you set the breakpoint on line 3, which is inside of the `for` loop, the debugger stops on this line each time it goes through the loop.

Press *Go* a third time. Now `i` and `j` have the values `3` and `6`. What do you think happens when you press *Go* one more time? Since the `for` loop only iterates 3 times, when you press *Go* this time, the program finishes running.

"Over" and "Out"

The *Over* button works as sort of a combination of *Step* and *Go*. It steps over a function or loop. In other words, if you're about to *Step* into a function with the debugger, you can still run that function's code without having to *Step* all the way through each line of it. The *Over* button takes you directly to the result of running that function.

Likewise, if you're already inside of a function or loop, the *Out* button executes the remaining code inside the function or loop body and then pauses.

In the next section, you'll look at some buggy code and learn how to fix it with IDLE.

Leave feedback on this section »

7.2 Squash Some Bugs

Now that you've gotten comfortable using the Debug Control Window let's take a look at a buggy program.

The following code defines a function `add_underscores()` that takes a single string object `word` as an argument and returns a new string containing a copy `word` with each character surrounded by underscores. For example, `add_underscores("python")` should return `"_p_y_t_h_o_n_"`.

Here's the buggy code:

```python
def add_underscores(word):
    new_word = "_"
    for i in range(0, len(word)):
        new_word = word[i] + "_"
    return new_word

phrase = "hello"
print(add_underscores(phrase))
```

Save and run the above script. The expected output is `_h_e_l_l_o_`, but instead all you see is `o_`, a space followed by a single underscore. If you already see what the problem with the code is, don't just fix it. The point of this section is to learn how to use IDLE's debugger to identify the problem. If you don't see what the problem is, don't worry! By the end of this section, you'll have found it and will be able to identify problems like it in other code you encounter.

> **Note**
>
> When working with real-world problems, debugging can often be difficult and time-consuming, and bugs can be subtle and hard to identify. While this section looks at a relatively simple bug, the important thing to take away from this is the methodology used to inspect the code.

Debugging is problem-solving, and as you become more experienced, you will develop your own approaches. In this section, you'll learn a simple four-step method to help get you started:

1. Guess which section of code may contain the bug.
2. Set a breakpoint and inspect the code by stepping through the buggy section one line at a time, keeping track of important variables along the way.
3. Identify the line of code, if any, with the error and make a change to solve the problem.
4. Repeat steps 1–3 as needed until the code works as expected.

Step 1: Make a Guess About Where the Bug Is Located

The first step is to identify the section of code that likely contains the bug. You may not be able to identify exactly where the bug is at first, but you can usually make a reasonable guess about which section of your code has an error.

Notice that the script is split into two distinct sections: a function definition (where `add_underscores()` is defined), and a "main" code block that defines a variable `phrase` with the value `"hello"` and then prints the result of calling `add_underscores(phrase)`.

Look at the "main" section:

```python
phrase = "hello"
print(add_underscores(phrase))
```

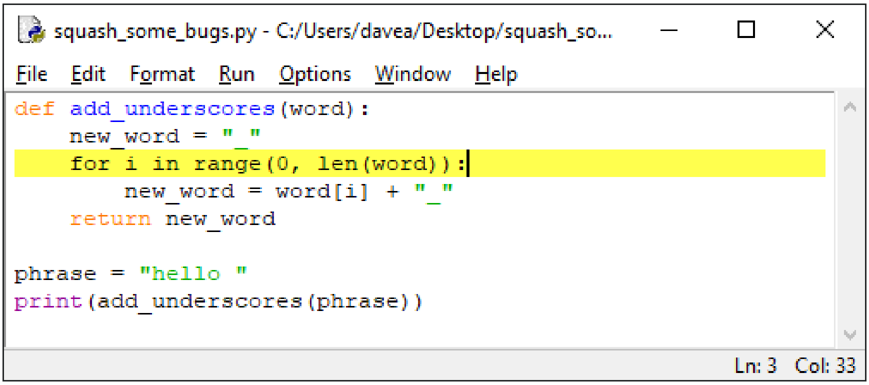
Do you think the problem could be here? It doesn't look like it, right? Everything about those two lines of code looks good. So, the problem must be in the function definition:

```python
def add_underscores(word):
    new_word = "_"
    for i in range(0, len(word)):
        new_word = word[i] + "_"
    return new_word
```

The first line of code inside the function creates a variable `new_word` with the value `"_"`. All good there, so we can conclude that the problem is somewhere in the body of the `for` loop.

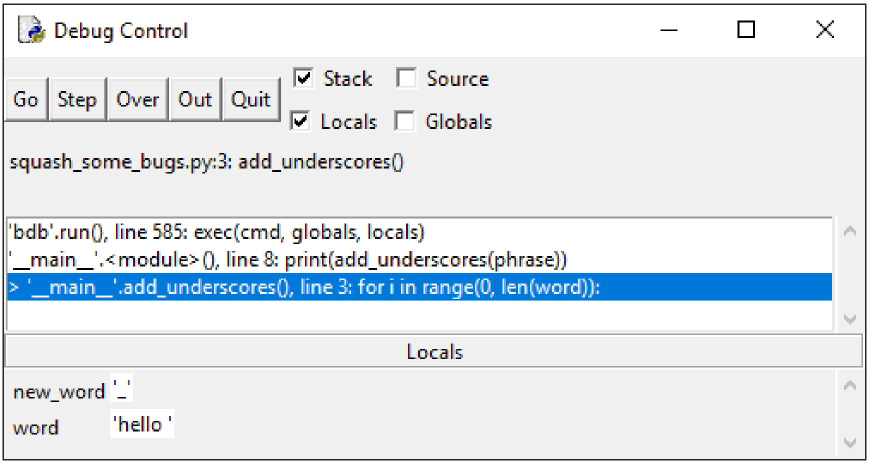
Step 2: Set a Breakpoint and Inspect the Code

Now that you've identified where the bug must be, set a breakpoint at the start of the `for` loop so that you can trace out exactly what's happening inside with the Debug Control Window:


squash_some_bugs.py - C:/Users/davea/Desktop/squash_so...
File Edit Format Run Options Window Help
Ln: 3 Col: 33

```
def add_underscores(word):
    new_word = "_"
    for i in range(0, len(word)):
        new_word = word[i] + "_"
    return new_word

phrase = "hello "
print(add_underscores(phrase))
```

Now open the Debug Control Window and run the script. Execution still pauses on the very first line it sees (which is defining the function). Press the “Go” button to run through the code until the breakpoint is encountered. The Debug window will now look like this:



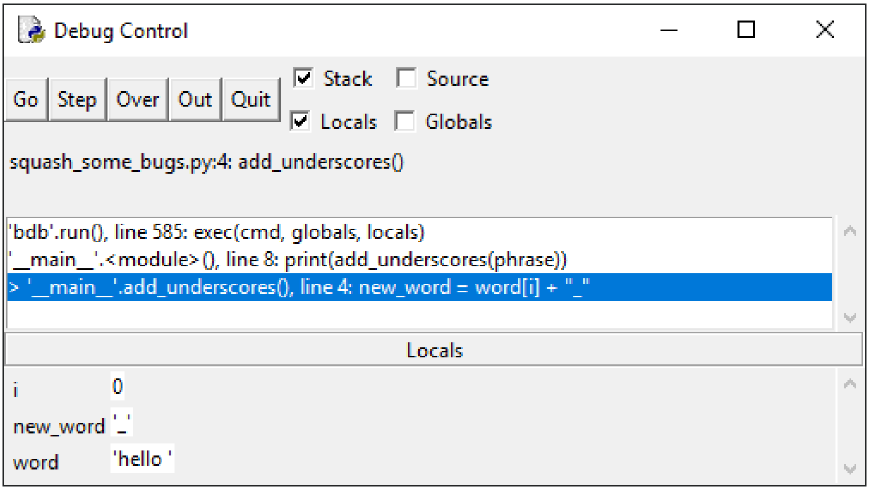
squash_some_bugs.py:3: add_underscores()

```
'bdb'.run(), line 585: exec(cmd, globals, locals)
'__main__'.<module>(), line 8: print(add_underscores(phrase))
> '__main__'.add_underscores(), line 3: for i in range(0, len(word)):
```

```
new_word '_'
word     'hello '
```

At this point, the code is paused just before entering the `for` loop in the `add_underscores()` function. Notice that two local variables, `word` and `new_word` are displayed in the *Locals* panel. Currently, `word` has the value `"hello"` and `new_word` has the value `"_"`, as expected.

Click the *Step* button once to enter the for loop. The Debug window changes and a new variable i with the value 0 is displayed in the "Locals" panel. i is the counter used in the for loop, and you can use it to keep track of which iteration of the for loop you are currently looking at:



☑ Stack □ Source
☑ Locals □ Globals

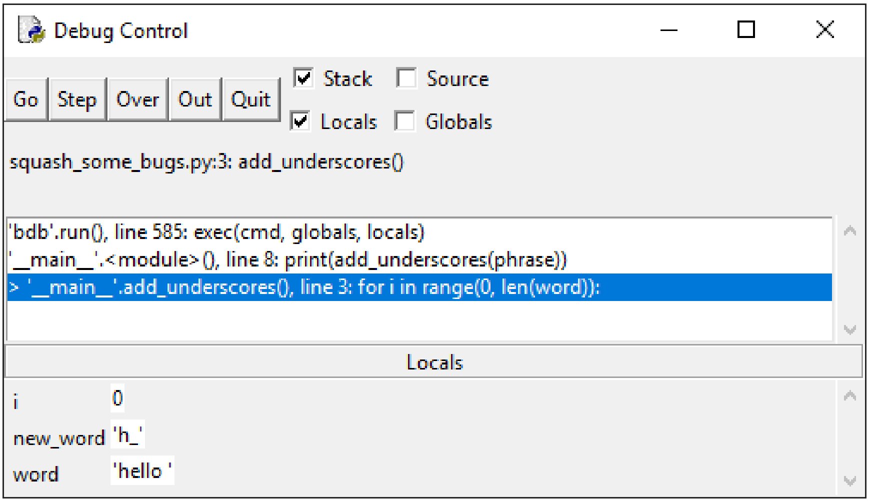
```
squash_some_bugs.py:4: add_underscores()
```

```
'bdb'.run(), line 585: exec(cmd, globals, locals)
'__main__'.<module>(), line 8: print(add_underscores(phrase))
> '__main__'.add_underscores(), line 4: new_word = word[i] + "_"
```

Locals

```
i         0
new_word  '_'
word      'hello '
```

Click *Step* one more time. If you look at the *Locals* panel, you'll see that the variable new_word has taken on the value "h_":



☑ Stack □ Source
☑ Locals □ Globals

```
squash_some_bugs.py:3: add_underscores()

'bdb'.run(), line 585: exec(cmd, globals, locals)
'__main__'.<module>(), line 8: print(add_underscores(phrase))
> '__main__'.add_underscores(), line 3: for i in range(0, len(word)):
```

Locals

```
i          0
new_word   'h_'
word       'hello'
```

This isn't right. Originally, `new_word` had the value `"_"` and on the second iteration of the `for` loop it should now have the value `"_h_"`. If you click *Step* a few more times, you'll see that `new_word` gets set to `e_`, then `l_`, and so on.

Step 3: Identify the Error and Attempt to Fix It

The conclusion you can make at this point is that `new_word` is overwritten at each iteration of the `for` loop with the next character in the string `"hello"` and a trailing underscore. Since there is only one line of code inside the `for` loop, you know that the problem must be with the following code:

```
new_word = word[i] + "_"
```

Look at that closely. This line tells Python to get the next character of `word`, tack an underscore to the end of it, and assign this new string to the variable `new_word`. This is exactly the behavior you've witnessed by stepping through the `for` loop!

To fix the problem, you need to tell Python to concatenate the string `word[i] + "_"` to the existing value of `new_word`. Press the "Quit" button in the Debug Control Window, but don't close that window just yet. Open the script window and change the line inside the `for` loop to:

```
new_word = new_word + word[i] + "_"
```

Step 4: Repeat Steps 1–3 Until the Bug is Gone

Save the new changes to your script and run it again. In the Debug window, press the *Go* button to execute the code up until the breakpoint.

Note

If you closed the debugger in the previous step without clicking on *Quit*, you may see the following error when re-opening the Debug Control Window:

```
You can only toggle the debugger when idle
```

Always be sure to click *Go* or *Quit* when you're finished with a debugging session instead of just closing the debugger, or you might have trouble reopening it. To get rid of this error, you'll have to close IDLE and re-open it.

Just like before, your script is now paused just before entering the `for` loop in the `add_underscores()` function. Press the *Step* button repeatedly and watch what happens to the `new_word` variable at each iteration. What do you see now? Success! Everything is working as expected!

In this example, your first attempt at fixing the bug worked, so you don't need to repeat steps 1–3 anymore. However, this won't always be the case. Sometimes you'll have to repeat the process several times before you've fixed a bug.

It's also important to keep in mind that tools like debuggers don't tell you how to fix a bug. They only help you identify where exactly a problem occurs in your code.

Alternative Ways to Find Mistakes in Your Code

Debugging can be tricky and time-consuming, but sometimes it's the most reliable way to find errors that you've overlooked. However, before you open a debugger, it is sometimes simpler to locate errors using well placed `print()` functions to display the values of your variables.

For example, instead of debugging the previous script with the Debug Control Window, you could add the following line to the end of the `for` loop in the `add_underscores()` function:

```
print(f"i = {i}; new_word = {new_word}")
```

The altered script would then look like this:

```
def add_underscores(word):
    new_word = "_"
    for i in range(0, len(word)):
        new_word = word[i] + "_"
        print(f"i = {i}; new_word = {new_word}")
    return new_word

phrase = "hello"
print(add_underscores(phrase))
```

When you run the script, the interactive window displays the following output:

```
i = 0; new_word = h_
i = 1; new_word = e_
i = 2; new_word = l_
i = 3; new_word = l_
i = 4; new_word = o_
o_
```

This shows you what the value of `new_word` is at each iteration of the `for` loop. The final line containing just a single underscore is the result of running `print(add_underscore(phrase))` at the end of the script.

By looking at the above output, you could come to the same conclusion you did while debugging with the Debug Control Window. That is, the problem is that `new_word` is overwritten at each iteration.

Many Python programmers prefer this simple method for some quick and dirty debugging on the fly. It is a handy technique but has some disadvantages when compared to IDLE's debugger.

The most significant disadvantage is that debugging with the `print()` function requires you to run your entire script each time you want to inspect the values of your variables. For long scripts, this can be an enormous waste of time compared to setting breakpoints and using the "Go" button in the Debug Control Window.

Another disadvantage is that you'll have to remember to remove those `print()` function calls from your code when you are done debugging it. Otherwise, users may see unnecessary and potentially confusing output when they run your program.

The example loop in this section may be a good example for illustrating the process of debugging, but it is not the best example of Pythonic code. The use of the index `i` is a giveaway that there might be a better way to write the loop.

One way to improve this loop is to iterate over the characters in the string `word` directly. Here's one way to do that:

```python
def add_underscores(word):
    new_word = "_"
    for char in word:
        new_word = new_word + char + "_"
    return new_word
```

The process of re-writing existing code to be cleaner, easier to read and understand, or adhere to code standards set by a team is called **refactoring**. We won't discuss refactoring much in this course, but it is an essential part of writing professional quality code.

Leave feedback on this section »

7.3 Summary and Additional Resources

In this chapter, you learned about IDLE's Debug window. You saw how to inspect the values of variables, insert breakpoints, and use the *Step*, *Go*, *Over* and *Out* buttons. You also got some practice debugging a function that didn't work correctly using a four-step process for identifying and removing bugs:

1. Guess where the bug is located
2. Set a breakpoint and inspect the code
3. Identify the error and attempt to fix it
4. Repeat steps 1–3 until the error is fixed

Debugging is as much an art as it is a science. The only way to master debugging is to get a lot of practice with it!

One way to get some practice is to open the Debug Control Window and use it to step through your code as you work on the exercises and challenges throughout the rest of this book.

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-7

Additional Resources

For more information on debugging Python code, check out the following resources:

- Python Debugging With Pdb
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 8

Conditional Logic and Control Flow

Up until this point, all of the code you have seen in this book is **unconditional**. That is, the code does not make any choices. Every line of code is executed in the order that is written or that functions are called, with possible repetitions inside of loops.

In this chapter, you will learn how to write programs that perform different actions based on different conditions using **conditional logic**. Paired with functions and loops, conditional logic allows you to write complex programs that handle many different situations.

In this chapter, you will learn how to:

- Compare the values of two or more variables
- Write `if` statements to control the flow of your programs
- Handle errors with `try` and `except`
- Apply conditional logic to create simple simulations

Let's get started!

Leave feedback on this section »

8.1 Compare Values

Conditional logic is based on performing different actions depending on whether or not some expression, called a **conditional**, is true or false. This idea is not specific to computers. Humans use conditional logic all the time to make decisions.

For example, the legal age for purchasing alcoholic beverages in the United States is 21. The statement "If you are at least 21 years old, then you may purchase a beer" is an example of conditional logic. The phrase "you are at least 21 years old" is a conditional because it may be either true or false.

In computer programming, conditionals often take the form of comparing two values, such as determining if one value is greater than another, or whether or not two values are equal to each other. A standard set of symbols called **boolean comparators** are used to make comparisons, and most of them may already be familiar to you.

The following table describes these boolean comparators:

Boolean Comparator	Example	Meaning
`>`	`a > b`	`a` greater than `b`
`<`	`a < b`	`a` less than `b`
`>=`	`a >= b`	`a` greater than or equal to `b`
`<=`	`a <= b`	`a` less than or equal to `b`
`!=`	`a != b`	`a` not equal to `b`
`==`	`a == b`	`a` equal to `b`

The term **boolean** is derived from the last name of the English mathematician George Boole, whose works helped lay the foundations of modern computing. In Boole's honor, conditional logic is sometimes called **boolean logic**, and conditionals are sometimes called **boolean expressions**.

There is also a fundamental data type called the **boolean**, or `bool` for short, which can have only one of two values. In Python, these values are conveniently named `True` and `False`:

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

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

The result of evaluating a conditional is always a boolean value:

```
>>> 1 == 1
True

>>> 3 > 5
False
```

In the first example, since `1` is equal to `1`, the result of `1 == 1` is `True`. In the second example, `3` is not greater than `5`, so the result is `False`.

Important

A common mistake when writing conditionals is to use the assignment operator `=`, instead of `==`, to test whether or not two values are equal.

Fortunately, Python will raise a `SyntaxError` if this mistake is encountered, so you'll know about it before you run your program.

You may find it helpful to think of boolean comparators as asking a question about two values. `a == b` asks whether or not `a` and `b` have the same value. Likewise, `a != b` asks whether or not `a` and `b` have different values.

Conditional expressions are not limited to comparing numbers. You may also compare values such as strings:

```
>>> "a" == "a"
True
```

```
>>> "a" == "b"
False

>>> "a" < "b"
True

>>> "a" > "b"
False
```

The last two examples above may look funny to you. How could on string be greater than or less than another?

The comparators < and > represent the notions of greater than and less than when used with numbers, but more generally they represent the notion of order. In this regard, `"a" < "b"` checks if the string `"a"` comes before the string `"b"`. But how are string ordered?

In Python, strings are ordered **lexicographically**, which is a fancy way to say they are ordered as they would appear in a dictionary. So you can think of `"a" < "b"` as asking whether or not the letter `a` comes before the letter `b` in the dictionary.

Lexicographic ordering extends to strings with two or more characters by looking at each component letter of the string:

```
>>> "apple" < "astronaut"
True

>>> "beauty" > "truth"
False
```

Since strings can contain characters other than letters of the alphabet, the ordering must extend to those other characters as well.

We won't go in to the details of how characters other than letters are ordered. In practice, the < and > comparators are most often used with numbers, not strings.

Review Exercises

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

1. For each of the following conditional expressions, guess whether they evaluate to `True` or `False`. Then type them into the interactive window to check your answers:

```
1 <= 1
1 != 1
1 != 2
"good" != "bad"
"good" != "Good"
123 == "123"
```

2. For each of the following expressions, fill in the blank (indicated by __) with an appropriate boolean comparator so that the expression evaluates to `True`:

```
3 __ 4
10 __ 5
"jack" __ "jill"
42 __ "42"
```

Leave feedback on this section »

8.2 Add Some Logic

In addition to boolean comparators, Python has special keywords called **logical operators** that can be used to combine boolean expressions. There are three logical operators: `and`, `or`, and `not`.

Logical operators are used to construct compound logical expressions. For the most part, these have meanings similar to their meaning in the English language, although the rules regarding their use in Python are much more precise.

The `and` Keyword

Consider the following statements:

1. Cats have four legs.
2. Cats have tails.

In general, both of these statements are true.

When we combine these two statements using `and`, the resulting sentence "cats have four legs and cats have tails" is also a true statement. If both statements are negated, the compound statement "cats do not have four legs and cats do not have tails" is false, too.

Even when we mix and match false and true statements, the compound statement is false. "Cats have four legs and cats do not have tails" and "cats do not have four legs and cats have tails" are both false statements.

When two statements *P* and *Q* are combined with `and`, the **truth value** of the compound statement "*P* and *Q*" is true if and only if both *P* and *Q* are true.

Python's `and` operator works exactly the same way. Here are four example of compound statements with `and`:

```
>>> 1 < 2 and 3 < 4  # Both are True
True
```

Both statements are `True`, so the combination is also `True`.

```
>>> 2 < 1 and 4 < 3  # Both are False
False
```

Both statements are `False`, so their combination is also `False`.

```
>>> 1 < 2 and 4 < 3  # Second statement is False
False
```

`1 < 2` is `True`, but `4 < 3` is `False`, so their combination is `False`.

```
>>> 2 < 1 and 3 < 4  # First statement is False
False
```

`2 < 1` is `False`, and `3 < 4` is `True`, so their combination is `False`.

The following table summarizes the rules for the `and` operator:

Combination using `and`	Result
`True and True`	`True`
`True and False`	`False`
`False and True`	`False`
`False and False`	`False`

You can test each of these rules in the interactive window:

```
>>> True and True
True

>>> True and False
False

>>> False and True
False

>>> False and False
False
```

The `or` Keyword

When we use the word "or" in everyday conversation, sometimes we mean an **exclusive or**. That is, only the first option or the second option can be true.

For example, the phrase "I can stay or I can go" uses the exclusive or. I can't both stay and go. Only one of these options can be true.

In Python the `or` keyword is inclusive. That is, if P and Q are two expressions, the statement “*P* or *Q*” is true if any of the following are true:

1. *P* is true
2. *Q* is true
3. Both *P* and *Q* are true

Let’s look at some examples using numerical comparisons:

```
>>> 1 < 2 or 3 < 4  # Both are True
True

>>> 2 < 1 or 4 < 3  # Both are False
False

>>> 1 < 2 or 4 < 3  # Second statement is False
True

>>> 2 < 1 or 3 < 4  # First statement is False
True
```

Note that if any part of a compound statement is `True`, even if the other part is `False`, the result is always true `True`. The following table summarizes these results:

Combination using `or`	Result
`True or True`	`True`
`True or False`	`True`
`False or True`	`True`
`False or False`	`False`

Again, you can verify all of this in the interactive window:

```
>>> True or True
True
```

```
>>> True or False
True

>>> False or True
True

>>> False or False
False
```

The not Keyword

The not keyword reverses the truth value of a single expression:

Use of not	Result
not True	False
not False	True

You can verify this in the interactive window:

```
>>> not True
False

>>> not False
True
```

One thing to keep in mind with not, though, is that it doesn't always behave the way you might expect when combined with comparators like ==. For example, not True == False returns True, but False == not True will raise an error:

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

>>> False == not True
  File "<stdin>", line 1
```

```
    False == not True
                ^
SyntaxError: invalid syntax
```

This happens because Python parses logical operators according to an **operator precedence**, just like arithmetic operators have an order of precedence in everyday math.

The order of precedence for logical and boolean operators, from highest to lowest, is described in the following table. Operators on the same row have equal precedence.

Operator Order of Precedence (Highest to Lowest)
<, <=, ==, >=, >
not
and
or

Looking again at the expression `False == not True`, `not` has a lower precedence than `==` in the order of operations. This means that when Python evaluates `False == not True`, it first tries to evaluate `False == not` which is syntactically incorrect.

You can avoid the `SyntaxError` by surrounding `not True` with parentheses:

```
>>> False == (not True)
True
```

Grouping expressions with parentheses is a great way to clarify which operators belong to which part of a compound expression.

Building Complex Expressions

You can combine the `and`, `or` and `not` keywords with `True` and `False` to create more complex expressions. Here's an example of a more complex expression:

```
True and not (1 != 1)
```

What do you think the value of this expression is?

To find out, break the expression down by starting on the far right side. `1 != 1` is `False`, since `1` has the same value as itself. So you can simplify the above expression as follows:

```
True and not (False)
```

Now, `not (False)` is the same as `not False`, which is `True`. So you can simplify the above expression once more:

```
True and True
```

Finally, `True and True` is just `True`. So, after a few steps, you can see that `True and not (1 != 1)` evaluates to `True`.

When working through complicated expressions, the best strategy is to start with the most complicated part of the expression and build outward from there.

For instance, try evaluating the following expression:

```
("A" != "A") or not (2 >= 3)
```

Start by evaluating the two expressions in parentheses. `"A" != "A"` is `False` because `"A"` is equal to itself. `2 >= 3` is also `False` because `2` is smaller than `3`. This gives you the following equivalent, but simpler, expression:

```
(False) or not (False)
```

Since `not` has a higher precedence than `or`, the above expression is equivalent to the following:

```
False or (not False)
```

`not False` is `True`, so you can simplify the expression once more:

```
False or True
```

Finally, since any compound expression with `or` is `True` if any one of the expressions on the left or right of the `or` is `True`, you can conclude that `("A" != "A") or not (2 >= 3)` is `True`.

Grouping expressions in a compound conditional statement with parentheses improves readability. Sometimes, though, parenthesis are required to produce the expected value.

For example, upon first inspection, you may expect the following to output `True`, but it actually returns `False`:

```
>>> True and False == True and False
False
```

The reason this is `False` is that the `==` operator has a higher precedence than `and`, so Python interprets the expression as `True and (False == True) and False`. Since `False == True` is `False`, this is equivalent to `True and False and False`, which evaluates to `False`.

The following shows how to add parentheses so that the expression evaluates to `True`:

```
>>> (True and False) == (True and False)
True
```

Logical operators and boolean comparators can be confusing the first time you encounter them, so if you don't feel like the material in this section comes naturally, don't worry!

With a little bit of practice, you'll be able to make sense of what's going on and build your own compound conditional statements when you need them.

Review Exercises

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

1. Figure out what the result will be (`True` or `False`) when evaluating the following expressions, then type them into the interactive window to check your answers:

   ```
   (1 <= 1) and (1 != 1)
   not (1 != 2)
   ("good" != "bad") or False
   ("good" != "Good") and not (1 == 1)
   ```

2. Add parentheses where necessary so that each of the following expressions evaluates to `True`:

   ```
   False == not True
   True and False == True and False
   not True and "A" == "B"
   "B" and not "A" != "B"
   ```

Leave feedback on this section »

8.3 Control the Flow of Your Program

Now that we can compare values to one other with boolean comparators and build complex conditional statements with logical operators, we can add some logic to our code performs different actions for different conditions.

The `if` Statement

An `if` statement tells Python to only execute a portion of code if a condition is met.

For example, the following `if` statement will print `2 and 2 is 4` if the conditional `2 + 2 == 4` is `True`:

```python
if 2 + 2 == 4:
    print("2 and 2 is 4")
```

In English, you can read this as “if 2 + 2 is 4, then print the string `'2 and 2 is 4'`.”

Just like `while` loops, an `if` statement has three parts:

1. The `if` keyword
2. A test condition, followed by a colon
3. An indented block of code that is executed if the test condition is `True`

In the above example, the test condition is `2 + 2 == 4`. Since this expression is `True`, executing the `if` statement in IDLE displays the text `2 and 2 is 4`.

If the test condition is `False` (for instance, `2 + 2 == 5`), Python skips over the indented block of code and continued execution on the next non-indented line.

For example, the following `if` statement does not print anything:

```
if 2 + 2 == 5:
    print("Is this the mirror universe?")
```

A universe where `2 + 2 == 5` is `True` would be pretty strange indeed!

Note

Leaving off the colon (`:`) after the test condition in an `if` statement raises a `SyntaxError`:

```
>>> if 2 + 2 == 4
SyntaxError: invalid syntax
```

Once the indented code block in an `if` statement is executed, Python will continue to execute the rest of the program.

Consider the following script:

```
grade = 95
```

```
if grade >= 70:
    print("You passed the class!")

print("Thank you for attending.")
```

The output looks like this:

```
You passed the class!
Thank you for attending.
```

Since `grade` is `95`, the test condition `grade >= 70` is `True` and the string `"You passed the class!"` is printed. Then the rest of the code is executed and `"Thank you for attending."` is printed.

If you change the value of `grade` to `40`, the output looks like this:

```
Thank you for attending.
```

The line `print("Thank you for attending.")` is executed whether or not `grade` is greater than or equal to `70` because it is after the indented code block in the `if` statement.

A failing student will not know that they failed if all they see from your code is the text `"Thank you for attending."`.

Let's add another `if` statement to tell the student they did not pass if their grade is less than `70`:

```
grade = 40

if grade >= 70:
    print("You passed the class!")

if grade < 70:
    print("You did not pass the class :(")

print("Thank you for attending.")
```

The output now looks like this:

```
You did not pass the class :(
Thanks for attending.
```

In English, we can describe an alternate case with the word “otherwise.” For instance, “If your grade is 70 or above, you pass the class. Otherwise, you do not pass the class.”

Fortunately, there is a keyword that does for Python what the word “otherwise” does in English.

The `else` Keyword

The `else` keyword is used after an `if` statement in order to execute some code only if the `if` statement’s test condition is `False`.

The following script uses `else` to shorten the code in the previous script for displaying whether or not a student passed a class:

```python
grade = 40

if grade >= 70:
    print("You passed the class!")
else:
    print("You did not pass the class :(")

print("Thank you for attending.")
```

In English, the `if` and `else` statements together read as ”If the grade is at least 70, then print the string `"You passed the class!"`; otherwise, print the string `"You did not pass the class :("`.

Notice that the `else` keyword has no test condition, and is followed by a colon. No condition is needed, because it executes for any condition that fails the `if` statement’s test condition.

Important

Leaving off the colon (`:`) from the `else` keyword will raise a `SyntaxError`:

```
>>> if 2 + 2 == 5:
...     print("Who broke my math?")
... else
SyntaxError: invalid syntax
```

The output from the above script is:

```
You did not pass the class :(
Thank you for attending.
```

The line that prints `"Thank you for attending."` still runs, even if the indented block of code after `else` is executed.

The `if` and `else` keywords work together nicely if you only need to test a condition with exactly two states.

Sometimes, you need to check three or more conditions. For that, you use `elif`.

The `elif` Keyword

The `elif` keyword is short for "else if" and can be used to add additional conditions after an `if` statement.

Just like `if` statements, `elif` statements have three parts:

1. The `elif` keyword
2. A test condition, followed by a colon
3. An indented code block that is executed if the test condition evaluates to `True`

Important

Leaving off the colon (`:`) at the end of an `elif` statement raises a `SyntaxError`:

```
>>> if 2 + 2 == 5:
...     print("Who broke my math?")
... elif 2 + 2 == 4
SyntaxError: invalid syntax
```

The following script combines `if`, `elif`, and `else` to print the letter grade a student earned in a class:

```
grade = 85  # 1

if grade >= 90:  # 2
    print("You passed the class with a A.")
elif grade >= 80:  # 3
    print("You passed the class with a B.")
elif grade >= 70:  # 4
    print("You passed the class with a C.")
else:  # 5
    print("You did not pass the class :(")

print("Thanks for attending.")  # 6
```

Both `grade >= 80` and `grade >= 70` are `True` when `grade` is `85`, so you might expect both `elif` blocks on lines `3` and `4` to be executed.

However, only the first block for which the test condition is `True` is executed. All remaining `elif` and `else` blocks are skipped, so executing the script has the following output:

```
You passed the class with a B.
Thanks for attending.
```

Let's break down the execution of the script step-by-step:

1. `grade` is assigned the value `85` in the line marked `1`.
2. `grade >= 90` is `False`, so the `if` statement marked `2` is skipped.
3. `grade >= 80` is `True`, so the block under the `elif` statement in line `3` is executed, and `"You passed the class with a B."` is printed.
4. The `elif` and `else` statements in lines `4` and `5` are skipped, since the condition for the `elif` statement on line `3` was met.
5. Finally, line `6` is executed and `"Thanks for attending."` is printed.

The `if`, `elif`, and `else` keywords are some of the most commonly used keywords in the Python language. With them you can write code that responds differently to different conditions and some much more complicated problems code with no conditional logic.

Nested `if` Statements

Just like `for` and `while` loops can be nested within one another, you nest an `if` statement inside another to create complicated decision making structures.

Consider the following scenario. Two people play a one-on-one sport against one another. You must decide which of two players wins depending on the players' scores and the sport they are playing:

- If the two players are playing basketball, the player with the greatest score wins.
- If the two players are playing golf, then the player with the lowest score wins.
- In either sport, if the two scores are equal, the game is a draw.

The following program solves this using nested `if` statements:

```python
sport = input("Enter a sport: ")
p1_score = input("Enter player 1 score: ")
p2_score = input("Enter player 2 score: ")
```

```
# 1
if sport.lower() == "basketball":
    if p1_score == p2_score:
        print("The game is a draw.")
    elif p1_score > p2_score:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")
# 2
elif sport.lower() == "golf":
    if p1_score == p2_score:
        print("The game is a draw.")
    elif p1_score < p2_score:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")
# 3
else:
    print("Unknown sport.")
```

The output of the script depends on the input value. Here’s a sample execution using `basketball` as the sport:

```
Enter a sport: basketball
Player 1 score: 75
Player 2 score: 64
Player 1 wins.
```

Here’s the output with the same player scores, but `golf` as the sport:

```
Enter a sport: golf
Player 1 score: 75
Player 2 score: 64
Player 2 wins.
```

If you enter anything besides `basketball` or `golf` for the sport, the program displays `Unknown sport.`

All together, there are seven possible ways that the program can run, which are described in the following table:

Sport	Score values
`basketball`	`player1_score == player2_score`
`basketball`	`player1_score > player2_score`
`basketball`	`player1_score < player2_score`
`golf`	`player1_score == player2_score`
`golf`	`player1_score > player2_score`
`golf`	`player1_score < player2_score`
everything else	any combination

Nested `if` statements can create many possible ways that your code can run. If you have many deeply nested if statements (more than two levels), then the number of possible ways the code can execute grows quickly.

Note

The complexity that results from using deeply nested if statements may make it difficult to predict how your program will behave under given conditions.

For this reason, nested `if` statements are generally discouraged.

Let's see how we simplify the previous program by removing nested `if` statements.

First, regardless of the sport, the game is a draw if `player1_score` is equal to `player2_score`. So, we can move the check for equality out from the nested `if` statements under each sport to make a single `if` statement:

```python
if p1_score == p2_score:
    print("The game is a draw.")
```

```
elif sport.lower() == "basketball":
    if p1_score > p2_score:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")

elif sport.lower() == "golf":
    if p1_score < p2_score:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")

else:
    print("Unknown sport.")
```

Now there are only six ways that the program can execute.

That's still quite a few ways. Can you think of any way to make the program simpler?

Here's one way to simplify it. Player 1 wins if the sport is basketball and their score is greater than player 2's score, or if the sport is golf and their score is less than player 2's score.

We can describe this with compound conditional expressions:

```
p1_wins_basketball = sport == "basketball" and p1_score > p2_score
p1_wins_golf = sport == "golf" and p1_score < p2_score
p1_wins = player1_wins_basketball or player1_wins_golf
```

Now `p1_wins` will be `True` if player 1 wins the basketball game or the golf game, and will be `False` otherwise.

Using this code, you can simplify the program quite a bit:

```
if p1_score == p2_score:
    print("The game is a draw.")
```

```
elif sport.lower() == "basketball" or sport.lower() == "golf":
    p1_wins_basketball = sport == "basketball" and p1_score > p2_score
    p1_wins_golf = sport == "golf" and p1_score < p2_score
    p1_wins = p1_wins_basketball or p1_wins_golf

    if p1_wins:
        print("Player 1 wins.")
    else:
        print("Player 2 wins.")
else:
    print("Unknown sport")
```

In this revised version of the program, there are only four ways the program can execute, and the code is easier to understand.

Nested `if` statements are sometimes necessary. However, if you find yourself writing lots of nested `if` statements, it might be a good idea to stop and think about how you might simplify your code.

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 prompts the user to enter a word using the `input()` function, stores that input in a variable, and then displays whether the length of that string is less than 5 characters, greater than 5 characters, or equal to 5 characters by using a set of `if`, `elif` and `else` statements.

Leave feedback on this section »

8.4 Challenge: Find the Factors of a Number

A factor of a positive integer *n* is any positive integer less than or equal to *n* that divides *n* with no remainder.

For example, `3` is a factor of `12` because `12` divided by `3` is `4`, with no remainder. However, `5` is not a factor of `12` because `5` goes into `12` twice with a remainder of `2`.

Write a script `factors.py` that asks the user to input a positive integer and then prints out the factors of that number. Here's a sample run of the program with output:

```
Enter a positive integer: 12
1 is a factor of 12
2 is a factor of 12
3 is a factor of 12
4 is a factor of 12
6 is a factor of 12
12 is a factor of 12
```

Hint: Recall from Chapter 5 that you can use the `%` operator to get the remainder of dividing one number by another.

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

Leave feedback on this section »

8.5 Break Out of the Pattern

In Chapter 6 you learned how to repeat a block of code many times using a `for` or `while` loop. Loops are useful for performing a repetitive task and for applying some processing to many different inputs.

Combining `if` statements with `for` loops opens up powerful techniques for controlling how code is run.

In this section, you'll learn how to write `if` statements that are nested in `for` loops and learn about two keywords — `break` and `continue` — that allow you to more precisely control the flow of execution through a loop.

if Statements and for Loops

The block of code in a for loop is just like any other block of code. That means you can nest an if statement in a for loop just like you can anywhere else in your code.

The following example uses a for loop with an if statement to compute and display the sum of all even integers less than 100:

```
sum_of_evens = 0

for n in range(1, 100):
    if n % 2 == 0:
        sum_of_evens = sum_of_evens + n

print(sum_of_evens)
```

First, the sum_of_evens variable is initialized to 0. Then the program loops over the numbers 1 to 99 and adds the even values to sum_of_evens. The final value of sum_of_evens is 2450.

break

The break keyword tells Python to literally break out of a loop. That is, the loop stops completely and any code after the loop is executed.

For example, the following code loops over the numbers 0 to 3, but stops the loop when the number 2 is encountered:

```
for n in range(0, 4):
    if n == 2:
        break
    print(n)

print(f"Finished with n = {n}")
```

Only the first two numbers are printed in the output:

```
0
1
Finished with i = 2
```

continue

The `continue` keyword is used to skip any remaining code in the loop body and continue on to the next iteration.

For example, the following code loops over the numbers `0` to `3`, printing each number as is goes, but skips the number `2`:

```python
for i in range(0, 4):
    if i == 2:
        continue
    print(i)

print(f"Finished with i = {i}")
```

All the numbers except for `2` are printed in the output:

```
0
1
3
Finished with i = 3
```

Note

It’s always a good idea to give short but descriptive names to your variables that make it easy to tell what they are supposed to represent.

The letters `i`, `j` and `k` are exceptions because they are so common in programming.

These letters are almost always used when we need a “throwaway” number solely for the purpose of keeping count while working through a loop.

To summarize, the break keyword is used to stop a loop if a certain condition is met, and the continue keyword is used to skip an iteration of a loop when a certain condition is met.

for...else Loops

Loops can have their own else clause in Python, although this structure isn't used very frequently.

Let's look at an example:

```
phrase = "it marks the spot"

for character in phrase:
    if character == "X":
        break
else:
    print("There was no 'X' in the phrase")
```

The for loop in this example loops over the characters phrase "it marks the spot" and stops if the letter "X" is found.

If you run the code in the example, you'll see that There was no 'X' in the phrase is printed to the console.

Now try changing phrase to the string "X marks the spot". When you run the same code with this phrase, there is no output. What's going on?

Any code in the else block after a for loop is executed only if the for loop completes without encountering a break statement.

So, when you run the code with phrase = "it marks the spot", the line of code containing the break statement is never run since there is no X character in the phrase, which means that the else block is executed and the string "There was no 'X' in the phrase" is displayed.

On the other hand, when you run the code with phrase = "X marks the spot", the line containing the break statement *does* get executed, so the else block is never run and no output gets displayed.

Here's a practical example that gives a user three attempts to enter a password:

```
for n in range(3):
    password = input("Password: ")
    if password == "I<3Bieber":
        break
    print("Password is incorrect.")
else:
    print("Suspicious activity. The authorities have been alerted.")
```

This example loops over the number 0 to 2. On each iteration, the user is prompted to enter a password. If the password entered is correct, then break is used to exit the loop. Otherwise, the user is told that the password is incorrect and given another attempt.

After three unsuccessful attempts, the for loop terminates without ever executing the line of code containing break. In that case, the else block is executed and the user is warned that the authorities have been alerted.

Note

We have focused on for loops in this section because they are generally the most common kind of loops.

However, everything discussed here also works for while loops. That is, you can use break and continue inside a while loop. while loops can even have an else clause!

Using conditional logic inside the body of a loop opens up several posibilities for controlling how your code executes.

You can stop loops early with the break keyword or skip an iteration with continue. You can even make sure some code only runs if a loop completes without ever encountering a break statement.

These are some powerful tools to have in your tool kit!

Review Exercises

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

1. Using `break`, write a program that repeatedly asks the user for some input and only quits if the user enters `"q"` or `"Q"`. Recall that a simple way to create an infinite loop is with `while True:`.
2. Using `continue`, write a program that loops over the number `1` to `50` and prints only the numbers that are multiples of `3`.

Leave feedback on this section »

8.6 Recover From Errors

Encountering errors in your code might be frustrating, but it's totally normal! It happens to event the best programmers.

Programmers often refer to run-time errors as **exceptions**. So, when you encounter an error, congratulate yourself! You've just made the code do something exceptional!

Errors aren't always a bad thing. That is, they don't always mean you made a mistake. For example, trying to divide the `1` by `0` results in a `ZeroDivisionError`. If the divisor is entered by a user, you have no way of knowing ahead of time whether or not the user will enter a `0`!

In order to create robust programs, you need to be able to handle errors caused by invalid user input — or any other unpredictable source. In this section you'll learn how.

A Zoo of Exceptions

When you encounter an exception, it's useful to know what went wrong. Python has a number of built-in exception types that describe different kinds of errors.

Throughout this book you have seen several different errors. Let's collect them here and add a few new ones to the list.

ValueError

A `ValueError` occurs when an operation encounters an invalid value. For example, trying to convert the string `"not a number"` to an integer results in a `ValueError`:

```
>>> int("not a number")
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    int("not a number")
ValueError: invalid literal for int() with base 10: 'not a number'
```

The name of the exception is displayed on the last line, followed by a description of the specific problem that occurred. This is the general format for all Python exceptions.

TypeError

A `TypeError` occurs when an operation is performed on a value of the wrong type. For example, trying to add a string and an integer will result in a `TypeError`:

```
>>> "1" + 2
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    "1" + 2
TypeError: can only concatenate str (not "int") to str
```

NameError

A `NameError` occurs when you try to use a variable name that hasn't been defined yet:

```
>>> print(does_not_exist)
Traceback (most recent call last):
  File "<pyshell#3>", line 1, in <module>
    print(does_not_exist)
NameError: name 'does_not_exist' is not defined
```

`ZeroDivisionError`

A `ZeroDivisionError` occurs when the divisor in a division operation is `0`:

```
>>> 1 / 0
Traceback (most recent call last):
  File "<pyshell#4>", line 1, in <module>
    1 / 0
ZeroDivisionError: division by zero
```

`OverflowError`

An `OverflowError` occurs when the result of an arithmetic operation is too large. For example, trying to raise the value `2.0` to the power `1_000_000` results in an `OverflowError`:

```
>>> pow(2.0, 1_000_000)
Traceback (most recent call last):
  File "<pyshell#6>", line 1, in <module>
    pow(2.0, 1_000_000)
OverflowError: (34, 'Result too large')
```

You may recall from Chapter 5 that integers in Python have unlimited precision. This means that `OverflowErrors` can only occur with floating-point numbers.

Raising the *integer* `2` to the value `1_000_000` will not raise an `OverflowError`!

A list of Python's built-in exceptions can be found in the docs.

The `try` and `except` Keywords

Sometimes you can predict that a certain exception might occur. Instead of letting the program crash, you can catch the error if it occurs and do something else instead.

For example, you might need to ask the user to input an integer. If the user enters a non-integer value, such as the string `"a"`, you need to let them know that they entered an invalid value.

To prevent the program from crashing you can use the `try` and `except` keywords. Let's look at an example:

```python
try:
    number = int(input("Enter an integer: "))
except ValueError:
    print("That was not an integer")
```

The `try` keyword is used to indicate a `try` block and is followed by a colon. The code indented after `try` is executed. In this case, the user is asked to input an integer. Since `input()` returns a string, the user input is converted to an integer with `int()` and the result is assigned to the variable `number`.

If the user inputs a non-integer value, the `int()` operation will raise a `ValueError`. If that happens, the code indented below the line `except ValueError` is executed. So, instead of the program crashing, the string `"That was not an integer"` is displayed.

If the user does input a valid integer value, then the code in the `except ValueError` block is never executed.

On the other hand, if a different kind of exception had occurred, such as a `TypeError`, then the program will crash. The above example only handles one type of exception — a `ValueError`.

You can handle multiple exception types by separating the exception names with commas and putting the list of names in parentheses:

```python
def divide(num1, num2):
    try:
        print(num1 / num2)
    except (TypeError, ZeroDivisionError):
        print("encountered a problem")
```

In this example, the function `divide()` takes two parameters `num1` and `num2` and prints the result of dividing `num1` by `num2`.

If `divide()` is called with an argument that is a string, then the division operation will raise a `TypeError`. Additionally, if `num2` is `0`, then a `ZeroDivisionError` is raised.

The line `except (TypeError, ZeroDivisionError)` will handle both of these exceptions and display the string `"encountered an error"` if either exception is raised.

Many times, though, it is helpful to catch each error individually so that you can display text that is more helpful to the user. To do this, you can use multiple `except` blocks after a `try` block:

```python
def divide(num1, num2):
    try:
        print(num1 / num2)
    except TypeError:
        print("Both arguments must be numbers")
    except ZeroDivisionError:
        print("num2 must not be 0")
```

In this example, the `ValueError` and `ZeroDivisionError` are handled separately. This way, a more descriptive message is displayed if something goes wrong.

If one of `num1` or `num2` is not a number, then a `TypeError` is raised and the message `"Both arguments must be numbers"` is displayed. If `num2` is `0`, then a `ZeroDivisionError` is raised and the message `"num2 must not be 0"` is displayed.

The “Bare” `except` Clause

You can use the `except` keyword by itself without naming specific exceptions:

```
try:
    # Do lots of hazardous things that might break
except:
    print("Something bad happened!")
```

If any exception is raised while executing the code in the `try` block, the `except` block will run and the message `"Something bad happened!"` will be displayed.

This might sound like a great way to ensure your program never crashes, **but this is actually bad idea and the pattern is generally frowned upon**!

There are a couple of reasons for this, but the most important reason for new programmers is that catching every exception could hide bugs in your code, giving you a false sense of confidence that your code works as expected.

If you only catch specific exceptions, then when unexpected errors are encountered, Python will print the traceback and error information giving you more information to work with when debugging your code.

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 repeatedly asks the user to input an integer, displaying a message to “try again” by catching the `ValueError` that is raised if the user did not enter an integer.

   Once the user enters an integer, the program should display the number back to the user and end without crashing.

2. Write a program that asks the user to input a string and an integer *n*. Then display the character at index *n* in the string.

   Use error handling to make sure the program doesn't crash if the user does not enter an integer or the index is out of bounds. The program should display a different message depending on what error occurs.

Leave feedback on this section »

8.7 Simulate Events and Calculate Probabilities

In this section, we'll apply some of the concepts we've learned about loops and conditional logic to a real world problem: simulating events and calculating probabilities.

We'll be running a simple simulation known as a Monte Carlo experiment. Each experiment consists of a **trial**, which is just some process that can be repeated — such as flipping a coin – that generated some outcome — such as landing on heads or tails. The trial is repeated over and over again in order to calculate the probability that some outcome occurs.

In order to do this, we need to add some randomness to our code.

The `random` module

Python provides several functions for generating random numbers in the `random` module. A **module** is a collection of related code. Python's **standard library** is an organized collection of modules that you can **import** into your own code in order to solve various problems.

To import the `random` module, type the following into IDLE's interactive window:

```
>>> import random
```

Now we can use functions from the random module in our code. For example, the `randint()` has two required parameters called `a` and `b` and returns a random integer that is greater than or equal to `a` and less than or equal to `b`. Both `a` and `b` must be integers.

For example, the following code produces a random integer between `1` and `10`:

```
>>> random.randint(1, 10)
9
```

Since the result is random, your output will probably be different than `9`. If you type the same code in again, you will likely get a different number.

Since `randint()` is located in the `random` module, you must type `random` followed by a dot (`.`) and then the function name in order to use it.

It is important to remember that when using `randint()`, the two parameters `a` and `b` must both be integers, and the output might be equal to one of `a` and `b`. For instance, `random.randint(0, 1)` randomly returns either a `0` or a `1`.

Furthermore, each integer between `a` and `b` is equally likely to be return by `randint()`. So, for `randint(1, 10)`, each integer between `1` and `10` has a 10% chance of being returned. For `randint(0, 1)`, there is a 50% chance a `0` is returned.

Flipping Fair Coins

Let's see how to use `randint()` to simulate flipping a fair coin. By a fair coin, we mean a coin that, when flipped, has an equal chance of landing on heads or tails.

One trial for our experiment will be flipping the coin. The outcome is either a heads or a tails. The question is: in general, over many coin flips, what is the ratio of heads to tails?

Let's think about how to solve this problem. We'll need to keep track of how many times we get a heads or tails, so we need a heads tally and a tails tally. Each trial has two steps:

1. Flip the coin.
2. If the coin lands on heads, update the heads tally. Otherwise, the coin lands on tails so update the tails tally.

We need to repeat the trial many times, say 10,000. A `for` loop over `range(10_000)` is a good choice for doing something like that.

Now that we have a plan, let's start by writing a function called `coin_flip()` that randomly returns the string `"heads"` or the string `"tails"`. We can do this using `random.randint(0, 1)`. We'll use `0` to represent heads and `1` for tails.

Here's the code for the `coin_flip()` function:

```
import random

def coin_flip():
    """Randomly return 'heads' or 'tails'."""
    if random.randint(0, 1) == 0:
        return "heads"
    else:
        return "tails"
```

If `random.randint(0, 1)` returns a `0`, then `coin_flip()` returns `"heads"`. Otherwise, `coin_flip()` returns `"tails"`.

Now we can write a `for` loop that flips the coin 10,000 times and updates a heads or tails tally accordingly:

```
# First initialize the tallies to 0
heads_tally = 0
tails_tally = 0
```

```python
for trial in range(10_000):
    if coin_flip() == "heads":
        heads_tally = heads_tally + 1
    else:
        tails_tally = tails_tally + 1
```

First, two variables `heads_tally` and `tails_tally` are created and both are initialized to the integer `0`.

Then the `for` loop runs 10,000 times. Each time, the `coin_flip()` function is called. If it returns the string `"heads"`, then the `heads_tally` variable is incremented by `1`. Otherwise `tails_tally` is incremented by `1`.

Finally, we can print the ratio of heads and tails:

```python
ratio = heads_tally / tails_tally
print(f"The ratio of heads to tails is {ratio}")
```

If you save the above code to a script and run it a few times, you will see that the result is usually between `.98` and `1.02`. If you increase the `range(10_000)` in the for loop to, say, `range(50_000)`, the results should get closer to `1.0`.

This behavior makes sense. Since the coin is fair, we should expect that after many flips, the number of heads and roughly equal to the number of tails.

In life, things aren't always fair. A coin may have a slight tendency to land on heads instead of tails, or vice versa. So, how do you simulate something like an unfair coin?

Tossing Unfair Coins

`randint()` returns a `0` or a `1` with equal probability. If `0` represents tails and `1` represents heads, then to simulate an unfair coin we need a way to return one of `0` or `1` with a higher probability.

The `random()` function can be called without any arguments and returns a floating-point number greater than or equal to `0.0` but less than `1.0`. Each possible return value is equally likely. In probability theory, this is known as a uniform probability distribution.

One consequence of this is that, given a number `n` between `0` and `1`, the probability that `random()` returns a number less than `n` is just `n` itself. For example, the probability that `random()` is less than `.8` is `.8` and the probability that `random()` is less than `.25` is `.25`.

Using this fact, we can write a function that simulates a coin flip, but returns tails with a specified probability:

```python
import random

def unfair_coin_flip(probability_of_tails):
    if random.random() < probability_of_tails:
        return "tails"
    else:
        return "heads"
```

For example, `unfair_coin_flip(.7)` has a 70% chance of returning `"heads"`.

Let's re-write the coin flip experiment from earlier using `unfair_coin_flip()` to run each trial with an unfair coin:

```python
heads_tally = 0
tails_tally= 0

for trial in range(10_000):
    if unfair_coin_flip(.7) == "heads":
        heads_tally = heads_tally + 1
    else:
        tails_tally = tails_tally + 1

ratio = heads_tally / tails_tally
print(f"The ratio of heads to tails is {ratio}")
```

Running this simulation a few times shows that the ratio of heads to tails has gone down from `1` in the experiment with a fair coin to about `.43`.

In this section you learned about the `randint()` and `random()` functions in the `random` module and saw how to use conditional logic and loops to write some coin toss simulations. Simulations like these are used in numerous disciplines to make predictions and test computer models of real world events.

The `random` module provides many useful functions for generating random numbers and writing simulations. You can learn more about `random` in Real Python's Generating Random Data in Python (Guide).

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 function called `roll()` that uses the `randint()` function to simulate rolling a fair die by returning a random integer between `1` and `6`.
2. Write a script that simulates 10,000 rolls of a fair die and displays the average number rolled.

Leave feedback on this section »

8.8 Challenge: Simulate a Coin Toss Experiment

Suppose you flip a fair coin repeatedly until it lands on both heads and tails at least once each. In other words, after the first flip, you continue to flip the coin it until it lands on something different.

Doing this generates a sequence of heads and tails. For example, the first time you do this experiment, the sequence might be heads, heads, then tails.

How many flips are needed on average to for a coin to land on a both a heads and a tails?

Write a simulation that runs 10,000 trials of the experiment and prints the average number of flips per trial.

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

Leave feedback on this section »

8.9 Challenge: Simulate an Election

With some help from the `random` module and a little condition logic, you can simulate an election between two candidates.

Suppose two candidates, Candidate A and Candidate B, are running for mayor in a city with three voting regions. The most recent polls show that Candidate A has the following chances for winning in each region:

- Region 1: 87% chance of winning
- Region 2: 65% chance of winning
- Region 3: 17% chance of winning

Write a program that simulates the election 10,000 times and prints the percentage of where Candidate A wins.

To keep things simple, assume that a candidate wins the election is they win in at least two of the three regions.

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

Leave feedback on this section »

8.10 Summary and Additional Resources

In this chapter, you learned about conditional statements and conditional logic. You saw how to compare values using comparison operators like `<`, `>`, `<=`, `>=`, `!=`, and `==`. You also saw how to build complex conditional statements using `and`, `or` and `not`.

Next, you saw how to control the flow of your program using `if` statements. You learned how to create branches in your program using `if...else` and `if...elif...else`. You also learned how to control precisely how code is executed inside of an `if` block using `break` and `continue`.

You learned about the `try...except` pattern to handle errors that may occur during run-time. This is an important construct that allows your programs to handle the unexpected gracefully, and keep users of your programs happy that the program doesn't crash.

Finally, you applied the techniques you learned in this chapter and used the `random` module to build some simple simulations.

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-8

Additional Resources

A wise Vulcan once said:

> Logic is the beginning of wisdom, not the end.
>
> — Spock, *Star Trek*

Check out the following resources to learn more about conditional logic:

- Operators and Expressions in Python
- Conditional Statements in Python
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 9

Tuples, Lists, and Dictionaries

So far, you have been working with fundamental data types like `str`, `int`, and `float`. Many real-world problems are easier to solve when simple data types are combined into more complex data structures.

A **data structure** models a collection of data, such as a list of numbers, a row in a spreadsheet, or a record in a database. Modeling the data that your program interacts with using the right data structure is often the key to writing simple and effective code.

Python has three built-in data structures that are the focus of this chapter: tuples, lists, and dictionaries.

In this chapter, you will learn:

- How to work with tuples, lists, and dictionaries
- What immutability is and why it is important
- When to use different data structures

Let's dive in!

Leave feedback on this section »

9.1 Tuples Are Immutable Sequences

Perhaps the simplest compound data structure is a sequence of items.

A **sequence** is an ordered list of values. Each element in a sequence is assigned an integer, called an **index**, that determines the order in which the values appear. Just like strings, the index of the first value in a sequence is `0`.

For example, the letters of the English alphabet form a sequence whose first element is A and last element is Z. Strings are also sequences. The string `"Python"` has six elements, starting with `"P"` at index `0`, and `"n"` at index `5`.

Some real-world examples of sequences include the values emitted by a sensor every second, the sequence of a student's test scores, or the sequence of daily stock values for some company over a period of time.

In this section, you'll learn how to use Python's built-in `tuple` data type to create sequences of values.

What is a Tuple?

The word **tuple** comes from mathematics, where it is used to describe a finite ordered sequence of values.

Usually, mathematicians write tuples by listing each element, separated by a comma, inside a pair of parentheses. `(1, 2, 3)` is a tuple containing three integers.

Tuples are **ordered** because their elements appear in an ordered fashion. The first element of `(1, 2, 3)` is `1`, the second element is `2`, and the third is `3`.

Python borrows both the name and the notation for tuples from mathematics.

How to Create a Tuple

There are a few ways to create a tuple in Python. We will cover two of them:

1. Tuple literals
2. The `tuple()` built-in

Tuple Literals

Just like a string literal is a string that is explicitly created by surrounding some text with quotes, a **tuple literal** is a tuple that is written out explicitly as a comma-separated list of values surrounded by parentheses.

Here's an example of a tuple literal:

```
>>> my_first_tuple = (1, 2, 3)
```

This creates a tuple containing the integers `1`, `2`, and `3`, and assigns it to the name `my_first_tuple`.

You can check check that `my_first_tuple` is a `tuple` using `type()`:

```
>>> type(my_first_tuple)
<class 'tuple'>
```

Unlike strings, which are sequences of characters, tuples may contain any type of value, including values of different types. The tuple `(1, 2.0, "three")` is perfectly valid.

There is a special tuple that doesn't contain any values. This tuple is called the **empty tuple** and can be created by typing two parentheses without anything between them:

```
>>> empty_tuple = ()
```

At first glance, the empty tuple may seem like a strange and useless concept, but it is actually quite practical.

For example, suppose you are asked to provide a tuple containing all the integers that are both even and odd. No such integer exists, but the empty tuple allows you to provide the requested tuple.

How do you think you create a tuple with exactly one element? Try out the following in IDLE:

```
>>> x = (1)
>>> type(x)
<class 'int'>
```

When you surround a value with parentheses, but don't include any commas, Python interprets the value not as a tuple but as the type of value inside the parentheses. So, in this case, `(1)` is a just a weird way of writing the integer `1`.

To create the tuple containing the single value `1`, you need to include a comma after the `1`:

```
>>> x = (1,)
>>> type(x)
<class 'tuple'>
```

A tuple containing a single element might seem as a strange as the empty tuple. Couldn't you just drop all this tuple business and just use the value itself?

It all depends on the problem you are solving.

If you are asked to provide a tuple containing all prime numbers that are also even, you must provide the tuple `(2,)` since `2` is the only even prime number. The value `2` isn't a good solution because it isn't a tuple.

This might seem overly pedantic, but programming often involves a certain amount of pedantry. Computers are, after all, the ultimate pedants.

The `tuple()` Built-In

You can also use the `tuple()` built-in to create a tuple from another sequence type, such as a string:

```
>>> tuple("Python")
('P', 'y', 't', 'h', 'o', 'n')
```

`tuple()` only accepts a single parameter, so you can't just list the values you want in the tuple as individual arguments. If you do, Python raises a `TypeError`:

```
>>> tuple(1, 2, 3)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    tuple(1, 2, 3)
TypeError: tuple expected at most 1 arguments, got 3
```

You will also get a `TypeError` if the argument passed to `tuple()` can't be interpreted as a list of values:

```
>>> tuple(1)
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    tuple(1)
TypeError: 'int' object is not iterable
```

The word **iterable** in the error message indicates that a single integer can't be **iterated**, which is to say that the integer data type doesn't contain multiple values that can be accessed one-by-one.

The single parameter of `tuple()` is optional, though, and leaving it out produces an empty tuple:

```
>>> tuple()
()
```

However, most Python programmers prefer to use the shorter `()` for creating an empty tuple.

Similarities Between Tuples and Strings

Tuples and strings have a lot in common. Both are sequence types with a finite length, support indexing and slicing, are immutable, and can be iterated over in a loop.

The main difference between strings and tuples is that the elements of tuples can be any kind of value you like, whereas strings can only contain characters.

Let's look at some of the parallels between strings in tuples in more depth.

Tuples Have a Length

Both strings and tuples have a **length**. The length of a string is the number of characters in it. The length of a tuple is the number of elements it contains.

Just like strings, the `len()` function can be used to determine the length of a tuple:

```
>>> numbers = (1, 2, 3)
>>> len(numbers)
3
```

Tuples Support Indexing and Slicing

Recall from Chapter 4 that you can access a character in a string using index notation:

```
>>> name = "David"
>>> name[1]
'a'
```

The index notation `[1]` after the variable `name` tells Python to get the character at index 1 in the string `"David"`. Since counting starts at `0`, the character at index 1 is the letter `"a"`.

Tuples also support index notation:

```
>>> values = (1, 3, 5, 7, 9)
>>> values[2]
5
```

Another feature that strings and tuples have in common is slicing. Recall that you can extract a substring from a string using slicing notation:

```
>>> name = "David"
>>> name[2:4]
"vi"
```

The slice notation `[2:4]` after the variable `name` creates a new string containing the characters in `name` starting at position `2` and up to, but not including, the character at position `4`.

Slicing notation also works with tuples:

```
>>> values = (1, 3, 5, 7, 9)
>>> values[2:4]
(5, 7)
```

The slice `values[2:4]` creates a new tuple containing the all integers in `values` starting at position `2` and up to, but not including, the integer at position `4`.

The same rules governing string slices also apply to tuple slices. You may want to take some time to review the slicing examples in Chapter 4 with some of your own examples of tuples.

Tuples Are Immutable

Like strings, tuples are immutable. This means you can't change the value of an element of a tuple once it has been created.

If you do try to change the value at some index of a tuple, Python will raise a `TypeError`:

```
>>> values[0] = 2
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    values[0] = 2
TypeError: 'tuple' object does not support item assignment
```

Note

Although tuples are immutable, there are some situations in which the values in a tuple can change.

These quirks and oddities are covered in depth in Real Python's Immutability in Python video course.

Tuples Are Iterable

Just like strings, tuples are **iterable**, so you can loop over them:

```
>>> vowels = ("a", "e", "i", "o", "u")
>>> for vowel in vowels:
...     print(vowel.upper())
...
A
E
I
O
U
```

The `for` loop in this example works just like the `for` loops you saw in Chapter 6 that loop over a `range()` of numbers.

On the first step of the loop, the value `"a"` is extracted from the tuple `vowels`. It is converted to an upper case letter using the `.upper()` string method you learned about in Chapter 4, and then displayed with `print()`.

The next step of the loop extracts the value `"e"`, converts it to upper case, and prints it. This continues for each of the values `"i"`, ”`o"`, and "u".

Now that you've seen how to create tuples and some of the basic operations they support, let's look at some common use cases.

Tuple Packing and Unpacking

There is a third, although less common, way of creating a tuple. You can type out a comma-separated list of values and leave off the parentheses:

```
>>> coordinates = 4.21, 9.29
>>> type(coordinates)
<class 'tuple'>
```

It looks like two values are being assigned to the single variable coordinates. In a sense, they are, although the result is that both values are **packed** into a single tuple. You can verify that coordinates is indeed a tuple with type().

If you can pack values into a tuple, it only makes sense that you can unpack them as well:

```
>>> x, y = coordinates
>>> x
4.21
>>> y
9.29
```

Here the values contained in the single tuple coordinates are **unpacked** into two distinct variables x and y.

By combining tuple packing and unpacking, you can make multiple variable assignments in a single line:

```
>>> name, age, occupation = "David", 34, "programmer"
>>> name
'David'
```

```
>>> age
34
>>> occupation
'programmer'
```

This works because first, on the right hand side of the assignment, the values `"David"`, `34`, and `"programmer"` are packed into a tuple. Then the values are unpacked into the three variable `name`, `age`, and `programmer`, in that order.

Note

While assigning multiple variables in a single line can shorten the number of lines in a program, you may want to refrain from assigning too many values in a single line.

Assigning more than two or three variable this way can make it difficult to tell which value is assigned to which variable name.

Keep in mind that the number of variable names on the left of the assignment expression must equal the number of values in the tuple on the right hand side, otherwise Python will rase a `ValueError`:

```
>>> a, b, c, d = 1, 2, 3
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    a, b, c, d = 1, 2, 3
ValueError: not enough values to unpack (expected 4, got 3)
```

The error message here tells you that the tuple on the right hand side doesn't have enough values to unpack into the four variable names.

Python also raises a `ValueError` if the number of values in the tuple exceeds the number of variable names:

```
>>> a, b, c = 1, 2, 3, 4
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
```

```
    a, b, c = 1, 2, 3, 4
ValueError: too many values to unpack (expected 3)
```

Now the error message indicates that there are too many values in the tuple to unpack into three variables.

Checking Existence of Values With `in`

You can check whether or not a value is contained in a tuple with the `in` keyword.

```
>>> vowels = ("a", "e", "i", "o", "u")
>>> "o" in vowels
True
>>> "x" in vowels
False
```

If the value to the left of `in` is contained in the tuple to the right of `in`, the result is `True`. Otherwise, the result in `False`.

Returning Multiple Values From a Function

One common use of tuples is to return multiple values from a single function.

```
>>> def adder_subtractor(num1, num2):
...     return (num1 + num2, num1 - num2)
...
>>> adder_subtractor(3, 2)
(5, 1)
```

The function `adder_subtractor()` has two parameters, `num1` and `num2`, and returns a tuple whose first element is the sum of the two numbers, and whose second element is the difference.

Strings and tuples are just two of Python's built-in sequence types. Both are immutable and iterable and can be used with index and slicing notation.

In the next section, you'll learn about a third sequence type with one very big difference from strings and tuples: mutability.

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 tuple literal named `cardinal_numbers` that holds the strings `"first"`, `"second"` and `"third"`, in that order.
2. Using index notation and `print()`, display the string at index `1` in `cardinal_numbers`.
3. Unpack the values in `cardinal_numbers` into three new strings named `position1`, `position2` and `position3` in a single line of code, then print each value on a separate line.
4. Create a tuple called `my_name` that contains the letters of your name by using `tuple()` and a string literal.
5. Check whether or not the character `"x"` is in `my_name` using the `in` keyword.
6. Create a new tuple containing all but the first letter in `my_name` using slicing notation.

Leave feedback on this section »

9.2 Lists Are Mutable Sequences

The `list` data structure is another sequence type in Python. Just like strings and tuples, lists contain items that are indexed by integers, starting with `0`.

On the surface, lists look and behave a lot like tuples. You can use index and slicing notation with lists, check for the existence of an element using `in`, and iterate over lists with a `for` loop.

Unlike tuples, however, lists are **mutable**, meaning you can change the value at an index even after the list has been created.

In this section, you will learn how to create lists and compare them with tuples.

Creating Lists

Just like tuples, you can create lists with **list literals** and the `list()` built-in. Lists, though, can also be created with from a string using the `.split()` string method. We'll look at each of these methods for creating lists.

A list literal looks almost exactly like a tuple literal, except that it is surrounded with square brackets (`[` and `]`) instead of parentheses:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> type(colors)
<class 'list'>
```

When you inspect a list, Python displays it as a list literal:

```
>>> colors
["red", "yellow", "green", "blue"]
```

Lists usually contain values of the same type but, just like tuples, this is not required. The list literal `["one", 2, 3.0]` is perfectly valid.

Just like `tuple()`, you can use `list()` to create a new `list` from any other sequence:

```
>>> list("red")
['r', 'e', 'd']
```

You can even create a list from a tuple:

```
>>> colors_tuple = ("red", "yellow", "green", "blue")
>>> colors_list = list(colors_tuple)
>>> colors_list
```

```
["red", "yellow", "green", "blue"]
```

Finally, if you have a string containing a list of words separated by commas, you can convert the string to a list of words with the string object's `.split()` method:

```
>>> groceries = "eggs, milk, cheese"
>>> grocery_list = groceries.split(", ")
>>> grocery_list
['eggs', 'milk', 'cheese']
```

The `.split()` method has a single string parameter `sep` and splits the string into a list of words using `sep` as the separator for each word.

In the above example, `sep` is set to the string `", "`, so anything in the string `groceries` that lies between two occurrences of the string `", "` (or one end of the string and an occurrence of `", "`) is considered its own word and returned as a list element. The characters matching the string `sep` are not returned as elements of the string.

If you had used the string `","` in `.split()` instead of `", "`, here's what you would get:

```
>>> groceries.split(",")
['eggs', ' milk', ' cheese']
```

Notice that in the above example, every word except the string `"eggs"` has a space in front of it. That's because this space wasn't a part of the `sep` argument of `.split()`, so it is returned as a part of a word.

You can use any string you like with the `.split()` method. Here's what you get if you split the `groceries` with the string `"s"`:

```
>>> groceries.split("s")
['egg', ', milk, chee', 'e']
```

No matter what, `.split()` always returns a list, even if you pass to it a string that doesn't exist in the string you are splitting:

```
>>> groceries.split("x")
['eggs, milk, cheese']
```

Since the `groceries` string doesn't contain the character `"x"` anywhere, the entire string is returned as a single element of a list.

Basic List Operations

Indexing and slicing operations work on lists the same way they do on tuples. The following code block illustrates the operations that lists and tuples have in common:

```
>>> numbers = [1, 2, 3, 4]

>>> # Access elements by index
>>> numbers[1]
2

>>> # Create a new list using slicing notation
>>> numbers[1:3]
[2, 3]

>>> # Check existence of an element
>>> "Bob" in numbers
False

>>> # Iterate over a list with a for loop
>>> for number in numbers:
...     if number % 2 == 0:
...         print(f"{number} is even")
...     else:
...         print(f"{number} is odd")
...
1 is odd
2 is even
3 is odd
4 is even
```

Notice that each of these operations looks exactly like the counterpart for tuples. In fact, without seeing the list literal assigned to the variable `numbers` on the first line, there would be no way to tell that `numbers` refers to a list and not a tuple!

This brings us to the major difference between lists and tuples: mutability.

Changing Elements in a List

Lists are **mutable**, which means that you can alter the elements in a list.

You can think of a list as a sequence of numbered slots. Each slot can hold a value, and every slot must be filled at all times, but you can swap out the value in a given slot with a new one whenever you want.

To do this swap, you use an assignment expression with the slot you want to change specified by index notation on the left hand side of the assignment operator and the new value on the right hand side:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors[0] = "burgundy"
```

The value at index `0`, which is `"red"` when `colors` is created, will now be `"burgundy"`:

```
>>> colors
['burgundy', 'yellow', 'green', 'blue']
```

You can change several values in a list at once by assigning a list to a to slice:

```
>>> colors[1:3] = ["orange", "magenta"]
>>> colors
['burgundy', 'orange', 'magenta', 'blue']
```

`colors[1:3]` selects the two values at positions `1` and `2` in the `colors` list. The two values `"orange"` and `"magenta"` are assigned these two slots.

Changing multiple values this way has some surprising results when the length of the list of new values doesn't match the length of the slice they are assigned to.

For instance, when you assign a list with three values to a slice with two values, the first two values are inserted in the same positions as the slice.

However, a new slot is created in the list and any remaining values in the original list are shifted to the right to make room for the third value:

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors[1:3] = ["orange", "magenta", "aqua"]
>>> colors
['red', 'orange', 'magenta', 'aqua', 'blue']
```

The values "orange" and "magenta" replace the original values "yellow" and "green" in colors at the indices 1 and 2. Then a new slot is created at index 4 and "blue" is assigned to this index. Finally, "aqua" is assigned to index 3.

When the length of the list being assigned to the slice is less than the length of the slice, the overall length of the original list is reduced:

```
>>> colors[1:4] = ["yellow", "green"]
>>> colors
['red', 'yellow', 'green', 'blue']
```

The values "yellow" and "green" replace the values "orange" and "magenta" in colors at the indices 1 and 2. Then the value at index 3 is replaced with the value "blue". Finally, the slot at index 4 is removed from colors entirely.

As these examples illustrate, lists are quite malleable. You can swap out existing values for new values, as well as increase or decrease the length of the list.

There are several list methods for performing different operations for adding and removing elements to a list. Let's take a look at these methods now.

List Methods For Adding and Removing Elements

Although you can add and remove elements by carefully assigning lists to a slice, this isn't a very natural way of performing certain operations.

List methods make adding and removing elements in a list much more natural. They also make your code much more readable!

We'll look at several methods, starting with how to insert a single value into a list.

`list.insert()`

The `list.insert()` method is used to insert a single new value into a list. It takes two parameters, and index `i` and a value `x`, and inserts the value `x` at index `i` in the list.

```
>>> colors = ["red", "yellow", "green", "blue"]
>>> # Insert "orange" into the second position
>>> colors.insert(1, "orange")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue']
```

There are a couple of important observations to make about this example.

The first observation applies to all list methods. To use them, you first write the name of the list you want to manipulate, followed by a dot (`.`) and then the name of the list method.

So, to use `insert()` on the `colors` list, you must write `colors.insert()`. This works just like string and number methods do.

Next, notice that when the value `"orange"` is inserted at the index `1`, the value `"yellow"` and all following values are shifted to the right.

If the value for the index parameter of `.insert()` is larger than the greatest index in the list, the value is inserted at the end of the list:

```
>>> colors.insert(10, "violet")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'violet']
```

Here the value `"violet"` is actually inserted at index `5`, even though `.insert()` was called with `10` for the index.

You can also use negative indices with `.insert()`:

```
>>> colors.insert(-1, "indigo")
>>> colors
['red', 'orange', 'yellow', 'green', 'blue', 'indigo', 'violet']
```

This insert `"indigo"` into the slot and index `-1` which is the last element of the list. The value `"violet"` is shifted to the right by one slot.

Important

When you `.insert()` an item to a list, you do not need to assign the result to the original list.

For example, the following code actually erases the colors list:

```
>>> colors = colors.insert(-1, "indigo")
>>> print(colors)
None
```

`.insert()` is said to alter `colors` **in place**. This is true for all list methods that do not return a value.

If you can insert a value at a specified index, it only makes sense that you can also remove an element at a specified index.

list.pop()

The `list.pop()` method takes one parameter, an index `i`, and removes the value from the list at that index. The value that is removed is returned by the method:

```
>>> color = colors.pop(3)
>>> color
'green'
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo', 'violet']
```

Here, the value `"green"` at index `3` is removed and and assigned to the variable `color`. When you inspect the `colors` list, you can see that the string `"green"` has indeed been removed.

Unlike `.insert()`, if you pass a parameter larger than the length of the list to `.pop()`, Python raises an `IndexError`:

```
>>> colors.pop(10)
Traceback (most recent call last):
  File "<pyshell#0>", line 1, in <module>
    colors.pop(10)
IndexError: pop index out of range
```

Negative indices also work with `.pop()`:

```
>>> colors.pop(-1)
'violet'
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo']
```

If you do not pass a value to `.pop()`, it removes the last item in the list:

```
>>> colors.pop()
'indigo'
>>> colors
['red', 'orange', 'yellow', 'blue']
```

This way of removing the final element, by calling `.pop()` with no specified index, is generally considered the most Pythonic.

`list.append()`

The `list.append()` method is used to append an new element to the end of a list:

```
>>> colors.append("indigo")
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo']
```

After calling `.append()`, the length of the list increases by one and the value `"indigo"` is inserted into the final slot. Note that `.append()` alters the list in place, just like `.insert()`.

`.append()` is equivalent to inserting an element at an index greater than the length of the list. The above example could also have been written as follows:

```
>>> colors.insert(len(colors), "indigo")
```

`.append()` is both shorter and more descriptive than using `.insert()` this way, and is generally considered the more Pythonic way of added an element to the end of a list.

`list.extend()`

The `list.extend()` method is used to add several new elements to the end of a list:

```
>>> colors.extend(["violet", "ultraviolet"])
>>> colors
['red', 'orange', 'yellow', 'blue', 'indigo', 'violet', 'ultraviolet']
```

`.extend()` takes a single parameter that must be an iterable type. The elements of the iterable are appended to the list in the same order that they appear in the argument passed to `.extend()`.

Just like `.insert()` and `.append()`, `.extend()` alters the list in place.

Typically, the argument passed to `.extend()` is another list, but it could also be a tuple. For example, the above example could be written as follows:

```
>>> colors.extend(("violet", "ultraviolet"))
```

The four list methods discussed in this section make up the most common methods used with lists. The following table serves to recap everything you have seen here:

List Method	Description
`.insert(i, x)`	Insert the value `x` at index `i`
`.append(x)`	Insert the value `x` at the end of the list
`.extend(iterable)`	Insert all the values of `iterable` at the end of the list, in order
`.pop(i)`	Remove and return the element at index `i`

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 list named `food` list with two elements `"rice"` and `"beans"`.
2. Append the string `"broccoli"` to `food` using `.append()`.
3. Add the string `"bread"` and `"pizza"` to `"food"` using `.extend()`.
4. Print the first two items in the `food` list using `print()` and slicing notation.
5. Print the last item in `food` using `print()` and index notation.
6. Create a list called `breakfast` from the string `"eggs, fruit, orange juice"` using the string `.split()` method.
7. Verify that `breakfast` has three items using `len()`.

Leave feedback on this section »

9.3 Nesting, Copying, and Sorting Tuples and Lists

Now that you have learned what tuples and lists are, how to create them, and some basic operations with them, let's look at three more concepts:

1. Nesting
2. Copying
3. Sorting

Nesting Lists and Tuples

Lists and tuples can contain values of any type. That means lists and tuples can contain lists and tuples as values. A **nested list**, or **nested tuple**, is a list or tuple that is contained as a value in another list or tuple.

For example, the following list has two values, both of which are other lists:

```
>>> two_by_two = [[1, 2], [3, 4]]

>>> # two_by_two has length 2
>>> len(two_by_two)
2

>>> # Both elements of two_by_two are lists
>>> two_by_two[0]
[1, 2]
>>> two_by_two[1]
[3, 4]
```

Since `two_by_two[1]` returns the list `[3, 4]`, you can use **double index notation** to access an element in the nested list:

```
>>> two_by_two[1][0]
3
```

First, Python evaluates `two_by_two[1]` and returns `[3, 4]`. Then Python evaluates `[3, 4][0]` and returns the first element `3`.

In very loose terms, you can think of a list of lists or a tuple of tuples as a sort of table with rows and columns.

The `two_by_two` list has two rows, `[1, 2]` and `[3, 4]`. The columns are made of of the corresponding elements of each row, so the first columns contains the elements `1` and `3`, and the second column contains the elements `2` and `4`.

This table analogy is only an informal way of thinking about a list of lists, though. For example, there is no requirement that all the lists in a list of lists have the same length, in which case this table analogy starts to break down.

Note

Readers interested in data analysis or scientific computing may recognize lists of lists as a sort of matrix of values.

While you can use the built in `list` and `tuple` types for matrices, better alternatives exist. To learn how to work with matrices in Python, check out Chapter 16.

Copying a List

Sometimes you need to copy one list into another list. However, you can't just reassign one list object to another list object, because you'll get this (possibly surprising) result:

```
>>> animals = ["lion", "tiger", "frumious Bandersnatch"]
>>> large_cats = animals
>>> large_cats.append("Tigger")
>>> animals
```

```
['lion', 'tiger', 'frumious Bandersnatch', 'Tigger']
```

In this example, you first assign the list stored in the `animals` variable to the variable `large_cats`, and then we add a new string to the `large_cats` list. But, when the contents of `animals` are displayed you can see that the original list has also been changed.

This is a quirk of object-oriented programming, but it's by design. When you say `large_cats = animals`, the `large_cats` and `animals` variables both refer to the same object.

A variable name is really just a reference to a specific location in computer memory. Instead of copying all the contents of the list object and creating a new list, `large_cats = animals` assigns the memory location referenced by `animals` to `large_cats`. That is, both variables now refer to the same object in memory, and any changes made to one will affect the other.

To get an independent copy of the `animals` list, you can use slicing notation to return a new list with the same values:

```
>>> animals = ["lion", "tiger", "frumious Bandersnatch"]
>>> large_cats = animals[:]
>>> large_cats.append("leopard")
>>> large_cats
['lion', 'tiger', 'frumious Bandersnatch', 'leopard']
>>> animals
["lion", "tiger", "frumious Bandersnatch"]
```

Since no index numbers are specified in the `[:]` slice, every element of the list is returned from beginning to end. The `large_cats` list now has the same elements as `animals`, and in the same order, but you can `.append()` items to it without changing the list assigned to `animals`.

If you want to make a copy of a list of lists, you can do so using the `[:]` notation you saw earlier:

```
>>> matrix1 = [[1, 2], [3, 4]]
>>> matrix2 = matrix1[:]
>>> matrix2[0] = [5, 6]
>>> matrix2
[[5, 6], [3, 4]]
>>> matrix1
[[1, 2], [3, 4]]
```

Let's see what happens when you change the first element of the second list in `matrix2`:

```
>>> matrix2[1][0] = 1
>>> matrix2
[[5, 6], [1, 4]]
>>> matrix1
[[1, 2], [1, 4]]
```

Notice that the second list in `matrix1` was also altered!

This happens because a list does not really contain objects themselves, but references to those objects in memory. When you make a copy of the list using the `[:]` notation, a new list is returned containing the same references as the original list. In programming jargon, this method of copying a list is called a **shallow copy**.

To make a copy of both the list and all of the elements it contains, you must use what is known as a **deep copy**. This method of copying is beyond the scope of this course. For more information on shallow and deep copies, check out the Shallow vs Deep Copying of Python Objects article on realpython.com.

Sorting Lists

Lists have a `.sort()` method that sorts all of the items in ascending order. By default, the list is sorted in alphabetical or numerical order, depending on the type of elements in the list:

```
>>> # Lists of strings are sorted alphabetically
>>> colors = ["red", "yellow", "green", "blue"]
>>> colors.sort()
>>> colors
['blue', 'green', 'red', 'yellow']

>>> # Lists of numbers are sorted numerically
>>> numbers = [1, 10, 5, 3]
>>> numbers.sort()
>>> numbers
[1, 3, 5, 10]
```

Keep in mind that `.sort()` sorts the list in place, so you don't need to assign it's result to anything.

Review Exercises

1. Create a tuple `data` that with two values. The first value should be the tuple `(1, 2)` and the second value should be the tuple `(3, 4)`.
2. Write a `for` loop that loops over `data` and prints the sum of each nested tuple. The output should look like this:

```
Row 1 sum: 3
Row 2 sum: 7
```

3. Create the following list `[4, 3, 2, 1]` and assign it to the variable `numbers`.
4. Create a copy of the `numbers` list using the `[:]` slicing notation.
5. Sort the `numbers` list in numerical order using the `.sort()` method.

Leave feedback on this section »

9.4 Challenge: List of lists

Define a function, `enrollment_stats()`, that takes, as an input, a list of lists where each individual list contains three elements: (a) the name of a university, (b) the total number of enrolled students, and (c) the annual tuition fees.

Sample list:

```
universities = [
    ['California Institute of Technology', 2175, 37704],
    ['Harvard', 19627, 39849],
    ['Massachusetts Institute of Technology', 10566, 40732],
    ['Princeton', 7802, 37000],
    ['Rice', 5879, 35551],
    ['Stanford', 19535, 40569],
    ['Yale', 11701, 40500]
]
```

This function should return two lists: the first containing all of the student enrollment values and the second containing all of the tuition fees.

Next, define a `mean()` and a `median()` function. Both functions should take a single list as an argument and return the mean and median from the values in each list.

Use the return values from `enrollment_stats()` as arguments for each function.

Challenge yourself by finding a way to sum all the values in a `list` *without using the built-in* `sum()` *function for calculating the mean.*

At some point you should calculate the total students enrolled and the total tuition paid.

Finally, output all values:

```
*************************
Total students:   77285
Total tuition:  $ 271905
```

```
Student mean:     11040
Student median:   10566

Tuition mean:   $ 38843
Tuition median: $ 39849
*************************
```

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

Leave feedback on this section »

9.5 Challenge: Wax Poetic

Write a script `poetry.py` that will generate a poem based on randomly chosen words and a pre-determined structure. When you are done, you will be able to generate poetic masterpieces such as the following in mere milliseconds:

```
A furry horse

A furry horse curdles within the fragrant mango
extravagantly, the horse slurps
the mango meows beneath a balding extrovert
```

All of the poems will have this same general structure, inspired by Clifford Pickover:

```
{A/An} {adjective1} {noun1}

{A/An} {adjective1} {noun1} {verb1} {preposition1} the {adjective2} {noun2}
{adverb1}, the {noun1} {verb2}
the {noun2} {verb3} {preposition2} a {adjective3} {noun3}
```

Your script should include a function `make_poem()` that returns a multiline string representing a complete poem. The main section of the code should simply print `make_poem()` to display a single poem. In order to get there, use the following steps as a guide:

1. First, you'll need a vocabulary from which to create the poem. Create several lists, each containing words pertaining to one part of speech (more or less); i.e., create separate lists for nouns, verbs, adjectives, adverbs, and prepositions. You will need to include at least three different nouns, three verbs, three adjectives, two prepositions and one adverb. You can use the sample word lists below, but feel free to add your own:
    - Nouns: "fossil", "horse", "aardvark", "judge", "chef", "mango", "extrovert", "gorilla"
    - Verbs: "kicks", "jingles", "bounces", "slurps", "meows", "explodes", "curdles"
    - Adjectives: "furry", "balding", "incredulous", "fragrant", "exuberant", "glistening"
    - Prepositions: "against", "after", "into", "beneath", "upon", "for", "in", "like", "over", "within"
    - Adverbs: "curiously", "extravagantly", "tantalizingly", "furiously", "sensuously"
2. Choose random words from the appropriate list using the `random.choice()` function, storing each choice in a new string. Select three nouns, three verbs, three adjectives, one adverb, and two prepositions. Make sure that none of the words are repeated. (Hint: Use a `while` loop to repeat the selection process until you get a new word.)
3. Plug the words you selected into the structure above to create a poem string using f-strings.
4. Bonus: Make sure that the "A" in the title and the first line is adjusted to become an "An" automatically if the first adjective begins with a vowel.

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

Leave feedback on this section »

9.6 Store Relationships in Dictionaries

One of the most useful data structures in Python is the **dictionary**.

In this section, you'll learn what a dictionary is, how dictionaries differ from lists and tuples, and how to define and use dictionaries in your own code.

What is a Dictionary?

In plain English, a dictionary is a book containing the definitions of words. Each entry in a dictionary has two parts: the word being defined, and its definition.

Python dictionaries, like lists and tuples, store a collection of objects. However, instead of storing objects in a sequence, dictionaries hold information in pairs of data called **key-value pairs**. That is, each object in a dictionary has two parts: a **key** and a **value**.

The **key** in a key-value pair is a unique name that identifies the **value** part of the pair. Comparing this to an English dictionary, the key is like the word being defined and the value is like the definition of the word.

For example, you could use a dictionary to store names of states and their capitals:

Key	Value
`"California"`	`"Sacramento"`
`"New York"`	`"Albany"`
`"Texas"`	`"Austin"`

In the table above, the keys of the dictionary are the names of the states, and the values of the dictionary are the names of the capitals.

The difference between an English dictionary and a Python dictionary is that the relationship between a key and its value is completely arbitrary. Any key can be assigned to any value.

For example, the following table of key-value pairs is valid:

Key	Value
1	"Sunday"
"red"	12:45pm
17	True

The keys in this table don't appear to be related to the values at all. The only relationship is that each key is assigned to its corresponding value by the dictionary.

In this sense, a Python dictionary is much more like a **map** than it is an English dictionary. The term map here comes from mathematics. It is used to describe a relation between two sets of values, not a geographical map.

In practice, it is this idea of dictionaries as a map that is particularly useful. Under this lens, the English dictionary is a special case of a map that relates words to their definitions.

So in summary, a Python dictionary is a data structure that relates a set of keys to a set of values. Each key is assigned a single value, which defines a relationship between the two sets.

Now that you have an idea what a dictionary is, let's see how to create dictionaries in Python code.

Creating Dictionaries

The following code creates a **dictionary literal** containing names of states and their capitals:

```
>>> capitals = {
  "California": "Sacramento",
  "New York": "Albany",
  "Texas": "Austin",
}
```

Notice that each key is separated from its value by a colon (`:`), each key-value pair is separated by a comma (`,`), and the entire dictionary is enclosed in curly braces (`{` and `}`).

You can also create a dictionary from a sequence of tuples using the `dict()` built-in:

```
>>> key_value_pairs = (
...     ("California", "Sacramento"),
...     ("New York", "Albany"),
...     ("Texas", "Austin"),
)
>>> capitals = dict(key_value_pairs)
```

When you inspect a dictionary, it is displayed as a dictionary literal, regardless of how it was created:

```
>>> capitals
{'California': 'Sacramento', 'New York': 'Albany', 'Texas': 'Austin'}
```

Note

If you happen to be following along with a Python version older than 3.6, then you will notice that the output dictionaries in the interactive window have a different order than the ones that appear in these examples.

Prior to Python 3.6, the order of key-value pairs in a Python dictionary was random. In later versions, the order of the key-value pairs is guaranteed to match the order in which they were inserted.

You can create an empty dictionary using either a literal or `dict()`:

```
>>> {}
{}
>>> dict()
{}
```

Now that we've created a dictionary, let's look at how you access its values.

Accessing Dictionary Values

To access a value in a dictionary, enclose the corresponding key in square brackets (`[` and `]`) at the end of dictionary or a variable name assigned to a dictionary:

```
>>> capital["Texas"]
'Austin'
```

The bracket notation used to access a dictionary value looks similar to the index notation used to get values from strings, lists, and tuples. However, dictionaries are a fundamentally different data structure than sequence types like lists and tuples.

To see the difference, let's step back for a second and notice that we could just as well define the `capitals` dictionary as a list:

```
>>> capitals_list = ["Sacramento", "Albany", "Austin"]
```

You can use index notation to get the capital of each of the three states from the `capitals` dictionary:

```
>>> capitals_list[0]  # Capital of California
'Sacramento'

>>> capitals_list[2]  # Capital of Texas
'Austin'
```

One nice thing about dictionaries is that they can be used to provide context to the values they contain. Typing `capitals["Texas"]` is easier to understand than `capitals_list[2]`, and you don't have to remember the order of data in a long list or tuple.

This idea of ordering is really the main difference between how items in a sequence type are accessed compared to a dictionary.

Values in a sequence type are accessed by index, which is an integer value expressing the order of items in the sequence.

On the other hand, items in a dictionary are accessed by a key, which doesn't define any kind of order, but just provides a label that can be used to reference the value.

Adding and Removing Values in a Dictionary

Like lists, dictionaries are mutable data structures. This means you can add and remove items from a dictionary.

Let's add the the capital of Colorado to the `capitals` dictionary:

```
>>> capitals["Colorado"] = "Denver"
```

First you use the square bracket notation with `"Colorado"` as the key, as if you were looking up the value. Then you use the assignment operator `=` to assign the value `"Denver"` to the new key.

When you inspect `capitals`, you see that a new key `"Colorado"` exists with the value `"Denver"`:

```
>>> capitals
{'California': 'Sacramento', 'New York': 'Albany', 'Texas': 'Austin',
'Colorado': 'Denver'}
```

Each key in a dictionary can only be assigned a single value. If a key is given a new value, Python just overwrites the old one:

```
>>> capitals["Texas"] = "Houston"
>>> capitals
```

```
{'California': 'Sacramento', 'New York': 'Albany', 'Texas': 'Houston',
'Colorado': 'Denver'}
```

To remove a from a dictionary, use the `del` keyword with the key for the value you want to delete:

```
>>> del capitals["Texas"]
>>> capitals
{'California': 'Sacramento', 'New York': 'Albany',
'Colorado': 'Denver'}
```

Checking the Existence of Dictionary Keys

If you try to access a value in a dictionary using a key that doesn't exist, Python raises a `KeyError`:

```
>>> capitals["Arizona"]
Traceback (most recent call last):
  File "<pyshell#1>", line 1, in <module>
    capitals["Arizona"]
KeyError: 'Arizona'
```

The `KeyError` is the most common error encountered when working with dictionaries. Whenever you see it, it means that an attempt was made to access a value using a key that doesn't exist.

You can check that a key exists in a dictionary using the `in` keyword:

```
>>> "Arizona" in capitals
False
>>> "California" in capitals
True
```

With `in`, you can first check that a key exists before doing something with the value for that key:

```
>>> if "Arizona" in capitals:
...     # Only print if the "Arizona" key exists
...     print(f"The capital of Arizona is {capitals['Arizona']}.")
```

It is important to remember that `in` only checks the existence of keys:

```
>>> "Sacramento" in capitals
False
```

Even though `"Sacramento"` is a value for the existing `"California"` key in `capitals`, checking for its existence returns `False`.

Iterating Over Dictionaries

Like lists and tuples, dictionaries are iterable. However, looping over a dictionary is a bit different than looping over a list or tuple.

When you loop over a dictionary with a `for` loop, you iterate over the dictionary's keys:

```
>>> for key in capitals:
...     print(key)
...
California
New York
Colorado
```

So, if you want to loop over the `capitals` dictionary and print "The capital of X is Y", where X is the name of the state and Y is the state's capital, you can do the following:

```
>>> for state in capitals:
        print(f"The capital of {state} is {capitals[state]}")


The capital of California is Sacramento
The capital of New York is Albany
The capital of Colorado is Denver
```

However, there is a slightly more succinct way to do this using the `.items()` dictionary method. `.items()` returns a list-like object containing tuples of key-value pairs. For example, `capitals.items()` returns a list of tuples of states and their corresponding capitals:

```
>>> capitals.items()
dict_items([('California', 'Sacramento'), ('New York', 'Albany'),
('Colorado', 'Denver')])
```

The object returned by `.items()` isn't really a list. It has a special type called a `dict_items`:

```
>>> type(capitals.items())
<class 'dict_items'>
```

You don't need to worry about what `dict_items` really is, because you usually won't work with it directly. The important thing to know about it is that you can use `.items()` to loop over a dictionary's keys and values simultaneously.

Let's rewrite the previous loop using `.items()`:

```
>>> for state, capital in capitals.items():
...     print(f"The capital of {state} is {capital}")

The capital of California is Sacramento
The capital of New York is Albany
The capital of Colorado is Denver
```

When you loop over `capitals.items()`, each iteration of the loop produces a tuple containing the state name and the corresponding capital city name. By assigning this tuple to `state, capital`, the components of the tuple are unpacked into the two variable `state` and `capital`.

Dictionary Keys and Immutability

In the `capitals` dictionary you've been working with throughout this section, each key is a string. However, there is no rule that says dictionary keys must all be of the same type.

For instance, you can add an integer key to `capitals`:

```
>>> capitals[50] = "Honolulu"
>>> capitals
{'California': 'Sacramento', 'New York': 'Albany',
'Colorado': 'Denver', 50: 'Honolulu'}
```

There is only one restriction on what constitutes a valid dictionary key. Only immutable types are allowed. This means, for example, that a list cannot be a dictionary key.

Consider this: what should happen if a list were used as a key in a dictionary and, somewhere later in the code, the list is changed?

Should the list be associated to the same value as the old list in the dictionary? Or should the value for the old key be removed from the dictionary all together?

Rather than make a guess about what should be done, Python raises an exception:

```
>>> capitals[[1, 2, 3]] = "Bad"
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unhashable type: 'list'
```

It might not seem fair that some types can be keys and others can't, but it's important that a programming language always has well-defined behavior. It should never make guesses about what the author intended.

For reference, here's a list of all the data types you've learned about so far that are valid dictionary keys:

Valid Dictionary Key Types
integers
floats
strings
booleans
tuples

Valid Dictionary Key Types

Unlike keys, dictionary values can be any valid Python type, including other dictionaries!

Nested Dictionaries

Just as you can nest lists inside of other lists, and tuples inside of other tuples, you can created nested dictionaries.

Let's alter the `capitals` dictionary to illustrate this idea. Instead of mapping state names to their capital cities, we'll create a dictionary that maps each state name to a dictionary containing the capital city and the state flower.

```
>>> states = {
...     "California": {
...         "capital": "Sacramento",
...         "flower": "California Poppy"
...     },
...     "New York": {
...         "capital": "Albany",
...         "flower": "Rose"
...     },
...     "Texas": {
...         "capital": "Austin",
...         "flower": "Bluebonnet"
...     },
... }
```

The value of each key is a dictionary:

```
>>> states["Texas"]
{'capital': 'Austin', 'flower': 'Bluebonnet'}
```

To get the Texas state flower, first get the value at the key `"Texas"`, and then the value at the key `"flower"`:

```
>>> states["Texas"]["flower"]
'Bluebonnet'
```

Nested dictionaries come up more often than you might expect. They are particularly useful when working with data transmitted over the web. Nested dictionaries are also great for modeling structured data, such as spreadsheets or relational databases.

Review Exercises

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

1. Create an empty dictionary named `captains`.
2. Using the square bracket notation, enter the following data into the dictionary, one item at a time:

   ```
   'Enterprise': 'Picard'
   'Voyager': 'Janeway'
   'Defiant': 'Sisko'
   ```

3. Write two `if` statements that check if `"Enterprise"` and `"Discovery"` exist as keys in the dictionary. Set their values to `"unknown"` if the key does not exist.
4. Write a `for` loop to display the ship and captain names contained in the dictionary. For example, the output should look something like this:

   ```
   The Enterprise is captained by Picard.
   ```

5. Delete `"Discovery"` from the dictionary.
6. Bonus: Make the same dictionary by using `dict()` and passing in the initial values when you first create the dictionary.

Leave feedback on this section »

9.7 Challenge: Capital City Loop

Review your state capitals along with dictionaries and `while` loops!

First, finish filling out the following dictionary with the remaining states and their associated capitals in a file called `capitals.py`.

```
capitals_dict = {
    'Alabama': 'Montgomery',
    'Alaska': 'Juneau',
    'Arizona': 'Phoenix',
    'Arkansas': 'Little Rock',
    'California': 'Sacramento',
    'Colorado': 'Denver',
    'Connecticut': 'Hartford',
    'Delaware': 'Dover',
    'Florida': 'Tallahassee',
    'Georgia': 'Atlanta',
}
```

Next, pick a random state name from the dictionary, and assign both the state and it's capital to two variables. You'll need to `import` the `random` module at the top of your program.

Then display the name of the state to the user and ask them to enter the capital. If the user answers, incorrectly, repeatedly ask them for the capital name until they either enter the correct answer or type the word "exit".

If the user answers correctly, display `"Correct"` and end the program. However, if the user exits without guessing correctly, display the correct answer and the word `"Goodbye"`.

Note

Make sure the user is not punished for case sensitivity. In other words, a guess of `"Denver"` is the same as `"denver"`. Do the same for exiting—`"EXIT"` and `"Exit"` should work the same as `"exit"`.

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

Leave feedback on this section »

9.8 How to Pick a Data Structure

In this chapter, you've learned about three data structures native to Python: lists, tuples, and dictionaries.

You might be wondering, "How do I know when to use which data structure?" It's a great question, and one many new Python programmers struggle with.

The type of data structure you use depends on the problem you are solving, and there is no hard and fast rule you can use to pick the right data structure every time. You'll always need to spend a little time thinking about the problem, and which structure works best for it.

Fortunately, there are some guidelines you can use to help you make the right choice. These are presented below:

Use a list when:

- Data has a natural order to it
- You will need to update or alter the data during the program
- The primary purpose of the data structure is iteration

Use a tuple when:

- Data has a natural order to it
- You **will not** need to update or alter the data during the program
- The primary purpose of the data structure is iteration

Use a dictionary when:

- The data is unordered, or the order does not matter
- You will need to update or alter the data during the program
- The primary purpose of the data structure is looking up values

Leave feedback on this section »

9.9 Challenge: Cats With Hats

You have 100 cats.

One day you decide to arrange all your cats in a giant circle. Initially, none of your cats have any hats on. You walk around the circle 100 times, always starting at the same spot, with the first cat (cat # 1). Every time you stop at a cat, you either put a hat on it if it doesn't have one on, or you take its hat off if it has one on.

1. The first round, you stop at every cat, placing a hat on each one.
2. The second round, you only stop at every second cat (#2, #4, #6, #8, etc.).
3. The third round, you only stop at every third cat (#3, #6, #9, #12, etc.).
4. You continue this process until you've made 100 rounds around the cats (e.g., you only visit the 100th cat).

Write a program that simply outputs which cats have hats at the end.

Note

This is not an easy problem by any means. Honestly, the code is simple. This problem is often seen on job interviews as it tests your ability to reason your way through a difficult problem. Stay calm. Start with a diagram, and then write pseudo code. Find a pattern. Then code!

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

Leave feedback on this section »

9.10 Summary and Additional Resources

In this chapter, you learned about three data structures: lists, tuples, and dictionaries.

Lists, such as `[1, 2, 3, 4]`, are mutable sequences of objects. You can interact with lists using various list methods, such as `.append()`, `.remove()`, and `.extend()`. Lists can be sorted using the `.sort()` method. You can access individual elements of a list using subscript notation, just like strings. Slicing notation also works with lists.

Tuples, like lists, are sequences of objects. The big difference between lists and tuples is that tuples are immutable. Once you create a tuple, it cannot be changed. Just like lists, you can access elements by index and using slicing notation.

Dictionaries store data as key-value pairs. The are not sequences, so you cannot access elements by index. Instead, you access elements by their key. Dictionaries are great for storing relationships, or when you need quick access to data. Like lists, dictionaries are mutable.

Lists, tuples and dictionaries are all iterable, meaning they can be looped over. You saw how to loop over all three of these structures using `for` loops.

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-9

Additional Resources

To learn more about lists, tuples, and dictionaries, check out the following resources:

- Lists and Tuples in Python
- Dictionaries in Python
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 10

Object-Oriented Programming (OOP)

OOP, or **O**bject-**O**riented **P**rogramming, is a method of structuring a program by bundling related properties and behaviors into individual **objects**.

Conceptually, objects are like components of a system. Think of a program as a factory assembly line of sorts. A system component at each step of the assembly line processes some material a little bit, ultimately transforming raw material into a finished product.

An object contains data, like the raw or pre-processed materials at each step on an assembly line, and behavior, like the action each assembly line component performs.

In this chapter, you will learn how to:

- Create a `class`, which is like a blueprint for creating object
- Use classes to create new objects
- Model systems with class inheritance

Let's get started!

Leave feedback on this section »

10.1 Define a Class

Primitive data structures—like numbers, strings, and lists—are designed to represent simple things, such as the cost of something, the name of a poem, and your favorite colors, respectively. What if you want to represent something much more complicated?

For example, let's say you wanted to track employees in an organization. You need to store some basic information about each employee, such as their name, age, position, and the year they started working.

One way to do this is to represent each employee as a list:

```
kirk = ["James Kirk", 34, "Captain", 2265]
spock = ["Spock", 35, "Science Officer", 2254]
mccoy = ["Leonard McCoy", "Chief Medical Officer", 2266]
```

There are a number of issues with this approach.

First, when you reference `kirk[0]` several lines away from where the `kirk` list is declared, will you remember that the 0th element of the list is the employee's name? What if not every employee has the same number of elements in the list?

Second, in the `mccoy` list above, the age is missing, so `mccoy[1]` will return `"Chief Medical Officer"` instead of Dr. McCoy's age.

A great way to make this type of code more manageable and more maintainable is to use **classes**.

Classes vs Instances

Classes are used to create user-defined data structures that contain data pertaining to some object. Often, classes also contain special function, called **methods**, that define behaviors and actions that an object can perform with its data.

In this chapter you'll create a `Dog` class that stores some basic information about a dog.

It's important to note that a class just provides structure. A class is a blueprint for how something should be defined. It doesn't actually provide any real content itself. The `Dog` class may specify that the name and age are necessary for defining a dog, but it will not actually state what a specific dog's name or age is.

While the class is the blueprint, an **instance** is an object built from a class that contains real data. An instance of the `Dog` class is not a blueprint anymore. It's an actual dog with a name, like Miles, who's four years old.

Put another way, a class is like a form or questionnaire. It defines the needed information. After you fill out the form, your specific copy is an instance of the class. It contains actual information relevant to you.

You can fill out multiple copies of a form to create many different instances, but without the form as a guide, you would be lost, not knowing what information is required. Thus, before you can create individual instances of an object, you must first specify what is needed by defining a class.

How to Define a Class

All class definitions start with the `class` keyword, which is followed by the name of the class and a colon. This is similar to the signature of a function, except that you don't need to add any parameters in parentheses. Any code that is indented below the class definition is considered part of the class's body.

Here is an example of a simple `Dog` class:

```python
class Dog:
    pass
```

The body of the `Dog` class consists of a single statement: the `pass` keyword. `pass` is often used as a place holder where code will eventually go. It allows you to run this code without throwing an error.

Note

Unlike functions and variables, the convention for naming classes in Python is to use CamelCase notation, starting with a capital letter. For example, a class for a specific breed of a dog, like the Jack Russell Terrier, would be written as `JackRussellTerrier`.

The `Dog` class isn't very interesting right now, so let's spruce it up a bit by defining some properties that all `Dog` objects should have. There are a number of properties that we can choose from, such as name, age, coat color, and breed. To keep things simple, we'll stick with just two for now: name and age.

To define the properties, or **instance attributes**, that all `Dog` objects must have, you need to define a special method called `.__init__()`. This method is run every time a new `Dog` object is created and tells Python what the initial **state**—that is, the initial values of the object's properties—of the object should be.

The first positional argument of `.__init__()` is always a variable that references the class instance. This variable is almost universally named `self`. After the `self` argument, you can specify any other arguments required to create an instance of the class.

The following updated definition of the `Dog` class shows how to write an `.__init__()` method that creates two instance attributes: `.name` and `.age`:

```
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Notice that the function signature—the part that starts with the `def` keyword—is indented four spaces. The body of the function is indented by eight spaces. This indentation is vitally important. It tells Python that the `.__init__()` method belongs to the `Dog` class.

Without the indentation, Python would treat `__init__()` as just another function.

Note

Functions that belong to a class are called **instance methods** because they belong to the instance of a class. For example, `list.append()` and `string.find()` are instance methods.

In the body of the `.__init__()` method, there are two statements using the `self` variable. The first line, `self.name = name`, creates a class attribute called `name` and assigns to it the value of the `name` variable that was passed to the `.__init__()` method. The second line creates an `age` attribute and assigns to it the value of the `age` argument.

This might look kind of strange. The `self` variable is referring to an instance of the `Dog` class, but we haven't actually created an instance yet. It is a place holder that is used to build the blueprint. Remember, the class is used to define the `Dog` data structure. It does not actually create any instances of individual dogs with specific names and ages.

While instance attributes are specific to each object, **class attributes** are the same for all instances—which in this case is all dogs. In the next example, a class attribute called `species` is created and assigned the value `"Canis familiaris"`:

```python
class Dog:
    # Class Attribute
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age
```

Class attributes are defined directly underneath the first line of the class and outside of any method definition. They must be assigned a value because they are created on a class instance without arguments to determine what their initial value should be.

You should use class attributes whenever a property should have the same initial value for all instances of a class. Use instance attributes for properties that must be specified before an instance is created.

Now that we have a `Dog` class, let's create some dogs!

Leave feedback on this section »

10.2 Instantiate an Object

Once a class has been defined, you have a blueprint for creating—also known as **instantiating**—new objects. To instantiate an object, you simple type the name of the class, in the original CamelCase, followed by parentheses containing any values that must be passed to the class's `.__init__()` method.

Let's take a look at an actual example. Open IDLE's interactive window and type the following:

```
>>> class Dog:
...     pass
...
```

This creates a new `Dog` class with no attributes and methods.

Next, instantiate a new `Dog` object:

```
>>> Dog()
<__main__.Dog object at 0x106702d30>
```

The output indicates that you now have a new `Dog` object at memory address `0x106702d30`. Note that the address you see on your screen will very likely be different from the address shown here.

Now let's instantiate another `Dog` object:

```
>>> Dog()
<__main__.Dog object at 0x0004ccc90>
```

The new `Dog` instance is located at a different memory address. This is because it is an entirely new instance, completely unique from the first `Dog` object you instantiated.

To see this another way, type the following:

```
>>> a = Dog()
>>> b = Dog()
>>> a == b
False
```

Two new `Dog` objects are created and assigned to the variables `a` and `b`. When `a` and `b` are compared using the `==` operator, the result is `False`. For user defined classes, the default behavior of the `==` operator is to compare the memory addresses of two objects and return `True` if the address is the same and `False` otherwise.

What this means is that even though the `a` and `b` object are both instances of the `Dog` class and have the exact same attributes and methods—namely, no attributes or methods, in this case—`a` and `b` represent two distinct objects in memory.

Note

The default behavior of the `==` operator can be overridden. How this is done is outside the scope of this book.

If you would like more information on how to customize the behavior of your classes, check out Real Python's Operator and Function Overloading in Custom Python Classes tutorial.

You can use the `type()` function to determine an object's class:

```
>>> type(a)
<class '__main__.Dog'>
```

Of course, even though both `a` and `b` are distinct `Dog` instances, they have the same type:

```
>>> type(a) == type(b)
True
```

Class and Instance Attributes

Let's look at a slightly more complex example using the `Dog` class we defined with `.name` and `.age` instance attributes:

```
>>> class Dog:
...     species = "Canis familiaris"
...     def __init__(self, name, age):
...         self.name = name
...         self.age = age
...
>>> buddy = Dog("Buddy", 9)
>>> miles = Dog("Miles", 4)
```

After declaring the new `Dog` class, two new instances are created—one `Dog` whose name is Buddy and is nine years old, and another named Miles who is four years old.

Does anything look a little strange about how the `Dog` objects are instantiated? The `.__init__()` method takes three arguments, so why are only two arguments specified in the example instead of three?

When you instantiate a `Dog` object, Python creates a new instance and passes it to the first argument of `.__init__()`. This happens for you behind the scenes, so you don't have to worry about it.

You can access instance attributes by using **dot notation**:

```
>>> buddy.name
'Buddy'
>>> buddy.age
9

>>> miles.name
```

```
'Miles'
>>> miles.age
4
```

Class attributes are accessed the same way:

```
>>> buddy.species
'Canis familiaris'
```

One of the biggest advantages of using classes to organize data is that instances are guaranteed to have the attributes you expect:

```
>>> buddy.species == miles.species
True
```

Both `buddy` and `miles` have the `.species` attribute. Contrast this to the method of using lists to represent similar data structures that you saw at the beginning of the previous section. With a class you no longer have to worry that an attribute may be missing.

Both instance and class attributes can be modified dynamically:

```
>>> buddy.age = 10
>>> buddy.age
10

>>> miles.species = "Felis silvestris"
>>> miles.species
'Felis silvestris'
```

In this example, the `.age` attribute of the `buddy` object is changed to `10`. Then the `.species` attribute of the `miles` object is changed to `"Felis silvestris"`, which is the species of the household cat. That makes Miles a pretty strange dog, but it is valid Python!

The important takeaway here is that custom objects are mutable by default. Recall that an object is mutable if it can be altered dynamically. For example, lists and dictionaries are mutable, but strings and tuples are not—they are immutable.

Now that you know the difference between a class and an instance, how to create instances and set class and instance attributes, the next step is to look at instance methods in more detail.

Instance Methods

Instance methods are functions defined inside of a class. This means that they only exist within the context of the object itself and cannot be called without referencing the object. Just like `.__init__()`, the first argument of an instance method is always `self`:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    # Instance method
    def description(self):
        return f"{self.name} is {self.age} years old"

    # Another instance method
    def speak(self, sound):
        return f"{self.name} says {sound}"
```

In this example, two new instance methods are defined: `.description()` and `.speak()`. The `.description()` method returns a string displaying the name and age of the dog, and `.speak()` takes one argument called `sound` and returns a string containing the dog's name and the sound the dog makes.

Let's see how instance methods work in practice. To avoid typing out the whole class in the interactive window, you can save the modified `Dog` class in a script in IDLE and run it. Then open the interactive window and type the following to see instance methods in action:

```
>>> miles = Dog("Miles", 4)

>>> miles.description()
'Miles is 4 years old'

>>> miles.speak("Woof Woof")
'Miles says Woof Woof'

>>> miles.speak("Bow Wow")
'Miles says Bow Wow'
```

The `.description()` method defined in the above `Dog` class returns a string containing information about the `Dog` instance `miles`. When writing your own classes, it is a good idea to have a method that returns a string containing useful information about an instance of the class. However, `.description()` isn't the most Pythonic way of doing this.

When you create a `list` object, you can use the `print()` function to display a string that looks like the list:

```
>>> names = ["Fletcher", "David", "Dan"]
>>> print(names)
['Fletcher', 'David', 'Dan']
```

Let's see what happens if we try and `print()` the `miles` object:

```
>>> print(miles)
<__main__.Dog object at 0x00aeff70>
```

When you `print(miles)`, you get a somewhat cryptic looking message telling you that `miles` is a `Dog` object located at some memory address.

You can specify what should be printed by defining a special instance method called `.__str__()` on the `Dog` class. Let's change `.description()` to `.__str__()` in the `Dog` class:

```
class Dog:
    # Class attributes and other methods ommitted...
    # Replace description with __str__
    def __str__(self):
        return f"{self.name} is {self.age} years old"
```

Now when you `print(miles)` you get much friendlier output:

```
>>> miles = Dog("Miles", 4)
>>> print(miles)
'Miles is 4 years old'
```

Note

> Methods like `.__str__()` are commonly called **dunder methods** because they begin and end with double underscores. There are a number of dunder methods available that allow your classes to work well with other Python language features.
>
> Dunder methods are powerful and are an important part of mastering OOP in Python, but we won't go into detail here. For more information, you are encouraged to checkout Operator and Function Overloading in Custom Python Classes as well as Chapter 4 of Python Tricks: The Book.

You should now have a pretty good idea of how to create a class that stores some data and provides some methods to interact with that data and define behaviors for an object.

In the next section, you'll see how to take your knowledge one step further and create classes from other classes. But first, 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. Modify the `Dog` class to include a third instance attribute called

`coat_color` that stores the color of the dog's coat as a string. Store your new class in a script and test it out by adding the following code at the bottom of the script:

```python
philo = Dog("Philo", 5, "brown")
print(f"{philo.name}'s coat is {philo.coat_color}.")
```

The output of your script should be:

```
Philo's coat is brown.
```

2. Create a `Car` class with two instance attributes: `.color`, which stores the name of the car's color as a string, and `.mileage`, which stores the number of miles on the car as an integer. Then instantiate two `Car` object—a blue car with 20,000 miles, and a red car with 30,000 miles, and print out their colors and mileage. Your output should look like the following:

```
The blue car has 20,000 miles.
The red car has 30,000 miles.
```

3. Modify the `Car` class with an instance method called `.drive()` that takes a number as an argument and adds that number to the `.mileage` attribute. Test that your solution works by instantiating a car with 0 miles, then call `.drive(100)` and print the `.mileage` attribute to check that it is set to `100`.

Leave feedback on this section »

10.3 Inherit From Other Classes

Inheritance is the process by which one class takes on the attributes and methods of another. Newly formed classes are called **child classes**, and the classes that child classes are derived from are called **parent classes**.

Child classes can override and extend the attributes and methods of parent classes. In other words, child classes inherit all of the parent's attributes and methods but can also specify different attributes and methods that are unique to themselves, or even redefine methods from their parent class.

The concept of object inheritance can be thought of sort of like genetic inheritance, even though the analogy isn't perfect.

For example, you may have inherited your hair color from your mother. It's an attribute you were born with. You may decide that you want to color you hair purple. Assuming your mother doesn't have purple hair, you have just **overridden** the hair color attribute you inherited from your mom.

You also inherit, in a sense, your language from your parents. If your parents speak English, then you will also speak English. One day, you may decide to learn a second language, like German. In this case you are **extending** attributes, because you have added an attribute that your parents do not have.

The `object` Class

The most basic type of class is an `object`, which generally all other classes inherit from as their parent. When you define a new class, Python 3 implicitly uses `object` as the parent class, so the following two definitions are equivalent:

```python
class Dog(object):
    pass

# In Python 3, this is the same as:

class Dog:
    pass
```

The inheritance from `object` is stated explicitly in the first definition by putting `object` in between parentheses after the `Dog` class name. This is the same pattern used to create child classes from your own custom classes.

Note

In Python 2 there's a distinction between new-style and old-style classes. We won't cover this distinction, because it doesn't apply to Python 3. Just know that in Python 3, there is an `object` class that all classes inherit from, even though you don't have to explicitly state that in your code.

Let's see how and why you might create child classes from a parent class.

Dog Park Example

Pretend for a moment that you are at a dog park. There are many dogs of different breeds at the park, all engaging in various dog behaviors.

Suppose now that you want to model the dog park with Python classes. The `Dog` class you wrote in the previous section can distinguish dogs by name and age, but not by breed.

You could modify the `Dog` class by adding a `.breed` attribute:

```
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age, breed):
        self.name = name
        self.age = age
        self.breed = breed
```

The instance methods defined earlier are omitted here because they aren't important for this discussion.

Now, to model the dog park, you could instantiate a bunch of different dogs:

```
>>> miles = Dog("Miles", 4, "Jack Russell Terrier")
>>> buddy = Dog("Buddy", 9, "Dachshund")
>>> jack = Dog("Jack", 3, "Bulldog")
>>> jim = Dog("Jim", 5, "Bulldog")
```

Each breed of dog has slightly different behaviors. For example, bulldogs have a low bark that sounds like “woof” but dachshunds have a higher pitched bark that sounds more like “yap”.

Using just the `Dog` class, you must supply a string for the `sound` argument of the `.speak()` method every time you call it on a `Dog` instance:

```
>>> buddy.speak("Yap")
'Buddy says Yap'

>>> jim.speak("Woof")
'Jim says Woof'

>>> jack.speak("Woof")
'Jack says Woof'
```

Passing a string to every call to `.speak()` is repetitive and inconvenient. What’s worse, the string representing the sound each `Dog` instance makes depends on the `.breed` attribute, but there is nothing stopping you, or someone using the `Dog` class you have created, from passing any string they wish.

You can simplify the experience of working with the `Dog` class by creating a child class for each breed of dog. This allows you to extend the functionality each child class inherits, including specifying a default argument for `.speak()`.

Parent Classes vs Child Classes

Let’s create a child class for each of the three breeds mentioned above: Jack Russell Terrier, Dachshund, and Bulldog.

For reference, here is the full definition of the `Dog` class:

```python
class Dog:
    species = "Canis familiaris"

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def __str__(self):
        return f"{self.name} is {self.age} years old"

    def speak(self, sound):
        return f"{self.name} says {sound}"
```

Remember, to create a child class, you create new class with its own name and then put the name of the parent class in parentheses. The following creates three new child classes of the `Dog` class:

```python
class JackRussellTerrier(Dog):
    pass

class Dachshund(Dog):
    pass

class Bulldog(Dog):
    pass
```

With the child classes defined, you can now instantiate some dogs of specific breeds:

```python
>>> miles = JackRussellTerrier("Miles", 4)
>>> buddy = Dachshund("Buddy", 9)
>>> jack = Bulldog("Jack", 3)
>>> jim = Bulldog("Jim", 5)
```

Instances of child classes inherit all of the attributes and methods of the parent class:

```
>>> miles.species
'Canis familiaris'

>>> buddy.name
'Buddy'

>>> print(jack)
Jack is 4 years old

>>> jim.speak("Woof")
'Jim says Woof'
```

To determine which class a given object belongs to, you can use the built-in `type()` function:

```
>>> type(miles)
<class '__main__.JackRussellTerrier'>
```

What if you wanted to determine if `miles` is also an instance of the `Dog` class? You can do this with the built-in `isinstance()` function:

```
>>> isinstance(miles, Dog)
True
```

Notice that `isinstance()` takes two arguments, an object and a class. In the example above, `isinstance()` checks if `miles` is an instance of the `Dog` class and returns `True`.

All of the `miles`, `buddy`, `jack` and `jim` objects are instances of the `Dog` class, but `miles` is not an instance of the `Bulldog` class, and `jack` is not an instance of the `Dachshund` class:

```
>>> isinstance(miles, Bulldog)
False

>>> isinstance(jack, Dachshund)
False
```

More generally, all objects created from a child class are instances of the parent class, although they may not be instances of other child classes.

Now that you've got some child classes created for some different breeds of dogs, let's give each breed its own sound.

Extending the Functionality of a Parent Class

At this point, we have four classes floating around: a parent class—`Dog`—and three child classes—`JackRussellTerrier`, `Dachshund` and `Bulldog`. All three child classes inherit every attribute and method from the parent class, including the `.speak()` method.

Since different breeds of dogs have slightly different barks, we want to provide a default value for the `sound` argument of their respective `.speak()` methods. To do this, we need to override the `.speak()` method in the class definition for each breed. To override a method defined on the parent class, you define a method with the same name on the child class.

Let's see what this looks like for the `JackRussellTerrier` class:

```python
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return f"{self.name} says {sound}"
```

The `.speak()` method is now defined on the `JackRussellTerrier` class with the default argument for `sound` set to `"Arf"`. Now you can call `.speak()` and a `JackRussellTerrier` instance without passing an argument to `sound`:

```python
>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles says Arf'
```

Sometimes dogs make different barks, so if Miles gets angry and growls, you can still call `.speak()` with a different sound:

```
>>> miles.speak("Grrr")
'Miles says Grrr'
```

One advantage of class inheritance is that changes to the parent class will automatically propagate to their child classes. This occurs as long as the attribute or method being changed isn't overridden in the child class.

For example, let's say you decide to change the string returned by `.speak()` in the `Dog` class:

```
class Dog:
    # Other attributes and methods omitted...

    def speak(self, sound):
        return f"{self.name} barks: {sound}"
```

Now, when you create a new `Bulldog` instance named `jim`, the result of `jim.speak("Woof")` will be `'Jim barks: Woof'` instead of `'Jim says Woof'`:

```
>>> jim = Bulldog("Jim", 5)
>>> jim.speak("Woof")
'Jim barks: Woof'
```

However, calling `.speak()` on a `JackRussellTerrier` instance won't show the new style of output:

```
>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles says Arf'
```

Sometimes it make sense to completely override a method from a parent class. But in this instance, we don't want the `JackRussellTerrier` class to lose any changes that might be made to the formatting of the output string of `Dog.speak()`.

To do this, you still need to define a `.speak()` method on the `JackRussellTerrier` class. But instead of explicitly defining the output string, you need to call the `Dog` class's `.speak()` method *inside* of the child class's `.speak()` method and make sure to pass to it the whatever is passed to `sound` argument of `JackRussellTerrier.speak()`.

You can access the parent class from inside a method of a child class by using the `super()` function. Here's how you could re-write the `JackRussellTerrier.speak()` method using `super()`:

```python
class JackRussellTerrier(Dog):
    def speak(self, sound="Arf"):
        return super().speak(sound)
```

When you call `super().speak(sound)` inside of `JackRussellTerrier`, Python searches the parent class, `Dog`, for a `.speak()` method and calls it with the variable `sound`. Now, when you call `miles.speak()`, you will see output reflecting the new formatting in the `Dog` class:

```
>>> miles = JackRussellTerrier("Miles", 4)
>>> miles.speak()
'Miles barks: Arf'
```

Important

In the above examples, the **class hierarchy** is very simple: the `JackRussellTerrier` class has a single parent class—`Dog`.

In many real world examples, the class hierarchy can get quite complicated with one class inheriting from a parent class, which inherits from another parent class, which inherits from another parent class, and so on.

The `super()` function does much more than just search the parent class for a method or an attribute. It traverses the entire class hierarchy for a matching method or attribute. If you aren't careful, `super()` can have surprising results.

In this section, you learned how to make new classes from existing classes utilizing an OOP concept called **inheritance**. You saw how to check if an object is an instance of a class or parent class using the isinstance() function. Finally, you learned how to extend the functionality of a parent class by using super().

In the next section you will bring together everything you have learned by using classes to model a farm. Before you tackle the assignment, check your understanding with the review exercises below.

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 GoldenRetriever class that inherits from the Dog class. Give the sound argument of the GoldenRetriever.speak() method a default value of "Bark". Use the following code for your parent Dog class:

   ```
   class Dog:
       species = "Canis familiaris"

       def __init__(self, name, age):
           self.name = name
           self.age = age

       def __str__(self):
           return f"{self.name} is {self.age} years old"

       def speak(self, sound):
           return f"{self.name} says {sound}"
   ```

2. Write a Rectangle class that must be instantiated with two attributed: length and width. Add a .area() method to the class that returns the area (length * width) of the rectangle. Then write a Square class that inherits from the Rectangle class and that is instantiated with a single attribute called side_length. Test your Square class by instantiating a Square with a side_length of 4. Calling the .area() method should return 16.

Leave feedback on this section »

10.4 Challenge: Model a Farm

In this assignment, you'll create a simplified model of a farm. As you work through this assignment, keep in mind that there are a number of correct answers.

The focus of this assignment is less about the Python class syntax and more about software design in general, which is highly subjective. This assignment is intentionally left open-ended to encourage you to think about how you would organize your code into classes.

Before you write any code, grab a pen and paper and sketch out a model of your farm, identifying classes, attributes, and methods. Think about inheritance. How can you prevent code duplication? Take the time to work through as many iterations as you feel are necessary.

The actual requirements are open to interpretation, but try to adhere to these guidelines:

1. You should have at least four classes: the parent `Animal` class, and then at least three child animal classes that inherit from `Animal`.
2. Each class should have a few attributes and at least one method that models some behavior appropriate for a specific animal or all animals—such as walking, running, eating, sleeping, and so on.
3. Keep it simple. Utilize inheritance. Make sure you output details about the animals and their behaviors.

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

Leave feedback on this section »

10.5 Summary and Additional Resources

In this chapter you learned about object-oriented programming (OOP) in Python, which is a programing paradigm that is not specific to Python. Most of the modern programming languages—such as Java, C#, and C++—follow OOP principles.

You saw how to define a class, which is a sort of "blueprint" for an object, and how to instantiate an object from a class. You also learned about attributes, which correspond to properties of an object, and methods, which correspond to behaviors and actions of an object.

Finally, you learned how inheritance works by creating child classes from a parent class. You saw how to reference a method on a parent class using `super()`, and how to check if an object inherits from some class using `isinstance()`.

OOP is a big and sometimes difficult topic. Some programmers consider OOP a foundational part of modern programming, but this viewpoint isn't without its criticisms.

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-10

Additional Resources

You've seen the basics of OOP, but there is so much more to learn! Continue your journey with the following resources:

1. Official Python documentation
2. OOP Articles on Real Python

3. Recommended resources on realpython.com

Leave feedback on this section »

Chapter 11

File Input and Output

Programs require data—sometimes lots of data! There are many different ways to store and retrieve data, but one of the most common is on a computer's file system.

Working with files is an essential skill for any programmer. Typical file operations might include saving program output to a text file, cleaning up a spreadsheet so that each column is formatted properly, or removing large files from a folder on your hard drive.

Whatever tasks you need to do with files, you can use Python to automate them!

In this chapter, you will learn how to:

- Read and write text files
- Deal with file paths
- Work with comma-separated value (CSV) files

Let's dive in!

Leave feedback on this section »

11.1 Read and Write Simple Files

So far, you've written programs that take input either from the program itself or from the user. If you need to work with a lot of data, these methods are problematic. In many real-world applications, the input is read from files. Python has the tools you need to work with files built into the language.

In this section, you will learn how to both read and write data to and from a file with Python.

Writing Output to a File

To read or write "raw" text files (the sort of file that you could use in a basic text editor because it contains no formatting or extra information), you use the general-purpose `open()` function. When you `open()` a file, the first thing you have to determine is if you want to read from it or write to it.

Let's start by creating a new text file and writing some data into it:

```
output_file = open("hello.txt", "w")
```

In this example, you've passed two arguments to the `open()` function. The first argument is a string that represents the actual name of the file we want to create: `hello.txt`. The second argument specifies the purpose, or **mode**, for opening the file. In this case, you passed `"w"`, which stands for "write," to open the file in write mode.

The `open()` function returns a `file` object that has been saved in the variable `output_file`. To write a line of text into the file, use the `.writelines()` method. Here is a simple script that writes a line of text into the `hello.txt` file:

```
output_file = open("hello.txt", "w")
output_file.writelines("This is my first file.")
output_file.close()
```

Make sure you know where you are saving this script before you run it. Right now, hello.txt will be created in the same folder as the script. That's because you only supplied a file name to the function, and not a path.

Note

You should always use the close() method to close any file that you have open() once you're completely done with the file. Python will eventually close any open files when you exit the program, but not closing files yourself can still cause unexpected problems.

This is because Python often buffers file output, meaning that it might save a bunch of commands you've written (without running them right away), then run them all in a big batch later on to make the process run faster.

This could result in something like the following unwanted situation. You write output to a file, then open that file up in a text editor to view the output. Since you didn't .close() the file, and IDLE is still running, the file is completely blank even though Python is planning to write output to the file before it is closed.

After running this script, you should see a new file named hello.txt appear in the same folder as your script. Open the file to check that it contains the line you wrote.

The .writelines() method can also take a list of lines to be written all at once. Without deleting the existing hello.txt file, run the following script:

```
output_file = open("hello.txt", "w")

lines_to_write = [
    "This is my file.",
    "There are many like it,",
    "but this one is mine.",
]

output_file.writelines(lines_to_write)
output_file.close()
```

Now open `hello.txt`. What happened?

There are two valuable lessons to take away from this example. First, when you `open()` a file in write mode, any existing content in the file is deleted. Deleting content from an important file in this manner is a common mistake, even for experienced Python developers.

Second, the strings in the `lines_to_write` list are written as a continuous line with no separating whitespace, like this:

```
This is my file.There are many like it,but this one is mine.
```

It is important to remember that `.writelines()` behaves this way, because the name `.writelines()` implies that each "line" will be written on a separate line in the file, even though this is not the case.

To write each string on a separate line, you must include a **newline character** by inserting `\n` at the beginning of the strings you would like to be written on new lines:

```
output_file = open("hello.txt", "w")

lines_to_write = [
    "This is my file.",
    "\nThere are many like it,",
    "\nbut this one is mine.",
]

output_file.writelines(lines_to_write)
output_file.close()
```

Now when you open `hello.txt`, you see this:

```
This is my file.
There are many like it,
but this one is mine.
```

In addition to “write” mode, the `open()` function can open a file in “append mode” by passing `"a"` to its second argument in place of `"w"`.

Append mode retains existing content and writes new content at the end of the file. Be careful, though, because you still need to insert `\n` to indicate that the new lines should be written on a new line in the file.

The following script opens `hello.txt` and appends the string “NON SEQUITUR” on a new line at the end of the file:

```
output_file = open("hello.txt", "a")
output_file.writelines("\nNON SEQUITUR")
output_file.close()
```

If you open `hello.txt`, you should now see the following text in it:

```
This is my file.
There are many like it,
but this one is mine.
NON SEQUITUR
```

Reading From a File

Now that you have written to a file, let’s see how to read its contents. You might be able to guess at how this can be done:

```
input_file = open("hello.txt", "r")
print(input_file.readlines())
input_file.close()
```

This time, `"r"` is passed as the second argument of `open()` to open the file in “read” mode. Then the `.readlines()` method is used to return each line of the file, which are displayed like so:

```
['This is my file.\n', 'There are many like it,\n',
 'but this one is mine.\n', 'NON SEQUITUR']
```

The above example illustrates that `.readlines()` returns a list of lines read from the file. Each string in the list contains one "line," including the newline character, which you can see printed in each string in the list.

This means that you can loop over this list with a `for` loop:

```python
input_file = open("hello.txt", "r")

for line in input_file.readlines():
    print(line)

input_file.close()
```

The output of the code above looks like this:

```
This is my file.

There are many like it,

but this one is mine.

NON SEQUITUR
```

Notice that there is an extra line in the output between each line read from the file. This happens because `print()` outputs a newline character by default after printing its argument.

To suppress this default behavior, you can specify a different output with the `end` parameter of the `print()` function:

```python
input_file = open("hello.txt", "r")

for line in input_file.readlines():
    print(line, end="")  # Output an empty string when done printing


input_file.close()
```

This outputs the following:

```
This is my file.
There are many like it,
but this one is mine.
NON SEQUITUR
```

Note

You can specify any string you'd like with `end`. For example, setting `end="!"` prints an exclamation point every time the `print()` function is done executing.

While `end=""` is probably the most common use case, specifying some other string can be useful for debugging purposes or determining where lines are separated in a file.

In addition to `.readlines()`, which returns a list of all lines in the file, you can read lines one at a time with the `.readline()` method. Python keeps track of where you are in the file for as long as you have it open, returning the next available line each time `.readline()` is called.

When `.readline()` reaches the end of the file, it returns an empty string. You can use this fact to write a `while` loop to read each line in the `hello.txt` file:

```
input_file = open("hello.txt", "r")


line = input_file.readline()  # Read the first line
while line != "":
    print(line, end="")
    line = input_file.readline()  # Read the next line
```

```
input_file.close()
```

At first glance, you may wonder why you would read the contents of a file with `.readline()` and not `.readlines()`, because the former requires more code. There are a couple of different reasons you might choose to do this. For example, `.readline()` is useful if you only need to read up to a certain point in the file.

You have just seen that `.readline()` returns an empty string once all content of a file has been read. Attempting to read with `.readline()` after reading the whole file always results in an empty string no matter how many times it is called.

This is because Python does not “reset” the read position until you have closed the file and re-opened it. The `.readlines()` method exhibits similar behavior.

For example, consider the output of the following script:

```
input_file = open("hello.txt", "r")

print("First pass:")
for line in input_file.readlines():
    print(line, end="")

print("\n\nSecond pass:")
# This loop doesn't do anything!
for line in input_file.readlines():
    print(line, end="")

input_file.close()
```

Running that produces this output:

```
First pass:
This is my file.
There are many like it,
but this one is mine.
```

```
NON SEQUITUR

Second pass:
```

Nothing is printed after “Second pass:” because `.readlines()` returns an empty list in the second `for` loop. At the end of this section, you will see how to manually reset the read position so that you can “fix” the above script using the `.seek()` method.

That said, if you need to pass over a file more than once, it is often easiest to do so by storing the list returned by `.readlines()` in a variable:

```python
input_file = open("hello.txt", "r")

lines_in_file = input_file.readlines()

print("First pass:")
for line in lines_in_file:
    print(line, end="")

print("\n\nSecond pass:")
for line in lines_in_file:  # Print the file contents a second time
    print(line, end="")

input_file.close()
```

There is an additional shortcut that can be helpful in organizing code when working with files: using Python’s `with` keyword:

```python
with open("hello.txt", "r") as input_file:
    for line in input_file.readlines():
        print(line)
```

Compare this code carefully to the previous examples. The first line does several things for you.

First, it opens `hello.txt` in “read” mode and stores the object returned by `open()` in the variable `input_file`. It then begins a new block of code where you can use the my_input_variable as you normally would.

What makes with special, though, is that when the following code block is exited, the file is closed for you automatically without having to call .close().

The with keyword makes managing file operations much simpler, and the resulting code is often much easier to read than the equivalent code that does not use with.

You can name multiple variables in a with statement to open multiple files at once. For instance, the following script reads in from hello.txt and writes its contents out to a new file hi.txt:

```
with open("hello.txt", "r") as source, open("hi.txt", "w") as dest:
    for line in source.readlines():
        dest.write(line)
```

Again, this takes care of all the clean-up work for you, closing both files once the block of code inside the with statement has terminated.

Note

Practically speaking, there's an easier way to accomplish this particular task; the shutil module includes many helpful functions including copy(), which can be used to copy an entire file into a new location.

The rest of the material in this section is conceptually more complicated and usually isn't necessary for most basic file reading/writing tasks.

Feel free to skim this remaining material for now and come back to it if you ever find that you need to read or write to a file in a way that involves specific parts of lines rather than taking entire lines from a file one by one.

Traversing a File with .seek()

To visit a specific part of a file, you can use the .seek() method to jump a particular number of characters into the file. For instance:

```
input_file = open("hello.txt", "r")
print("Line 0 (first line):", input_file.readline())

input_file.seek(0)  # Jump back to beginning
print("Line 0 again:", input_file.readline())
print("Line 1:", input_file.readline())

input_file.seek(8)  # Jump to character at index 8
print("Line 0 (starting at 9th character):", input_file.readline())

input_file.close()
```

The .seek() method can be challenging to understand at first. It may be helpful to run the above script and following along with the output as you read the code.

When you pass a single number to .seek(), it moves the read position to the character in the file with that index number, regardless of where you currently are in the file. So .seek(0) always gets you back to the beginning of the file, and .seek(8) the read position at the character at index position 8.

Opening a File for Both Reading and Writing

It is possible to open a file for both reading and writing. However, this is not always a good idea. Keeping track of where you are in a particular file using seek() to decide which pieces you want to read or write can be difficult.

If you need to do this, though, you can specify the mode "r+" to allow for both reading and writing, or "ra+" to both read and append to an existing file. Since writing and appending changes the characters in the file, however, you must perform a new seek() whenever switching modes from writing to reading.

Review Exercises

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

1. Read in the raw text file `poem.txt` from the Chapter 11 practice files and display each line by looping over them individually, then close the file; we'll discuss using file paths in the next section, but for now, you can save your script in the same folder as the text file
2. Repeat the previous exercise using the `with` keyword so that the file is closed automatically after you're done looping through the lines
3. Write a text file `output.txt` that contains the same lines as `poem.txt` by opening both files at the same time (in different modes) and copying the original file over line-by-line; do this using a loop and closing both files, then repeat this exercise using the `with` keyword
4. Re-open `output.txt` and append an additional line of your choice to the end of the file on a new line

Leave feedback on this section »

11.2 Working With Paths in Python

Chances are that you don't want to limit yourself to using the same folder as your script for all your files all the time. To get access to different directories, you can add them to the file name to specify a full path. For instance, you could pass the following fictitious path to `open()`:

```
input_file = open("C:/test/useless text files/hello.txt", "r")
```

Notice how only forward slashes are used in the path string. This method of substituting forward slashes works fine even in Windows, where the operating system's default is to use backslashes to separate directories.

The point of this is to avoid the “**escape character**” problem where Python would have treated a backslash and the character following it as a pair of special characters instead of reading them along with the rest of the string normally.

The backslash is called an “escape character” because it lets Python know that the backslash and the character following it should be read as a pair to represent a different character. For instance, `\n` would be interpreted as a newline character and `\t` represents a “tab” character.

Another way to get around this problem is to put a lowercase `r` just before a string, without a space, like so:

```
path = r"C:\My Documents\useless text files\hello.txt"
```

This creates a “raw” string that is read in exactly as it is typed, meaning that backslashes are only ever read as actual backslash characters and won’t be combined with any other characters to create special characters.

The `os` Module

To do anything more advanced with file structures, you can use the os module, which exposes various functions related to the operating system. The first thing that you will need to do is `import os` into your code.

If you are used to working in the command line, the `os` module gives you much of the same basic functionality that will probably already be somewhat familiar. For instance, the `rmdir()` function deletes a directory and the `mkdir()` function creates a new directory:

Soon you’ll see how to manipulate and interact with the included example files in various ways. Although there are many different ways to set your scripts up correctly for accessing files, we’ll follow a simple pattern in this course.

Whenever we need to write a script that makes use of an example file in the course folder, we will start with something like the following code:

```
import os

path = "C:/Real Python/python-basics-exercises"
```

The first line imports the os module so that you can work with its functions in your code. The second line stores a string representing the path to the course materials.

You should replace the string that gets assigned to the variable with a string that represents the location at which you've saved the main course materials folder (which is named Real Python/python-basics-exercises by default, but might not be saved directly onto your C: drive).

This way, you will only ever have to specify which folders inside of the course folder you want to access instead of typing it out each time you want to access a sample file. You will then join this path to the rest of each file location using the os.path.join() function, as you'll see below.

Many of the exercises in this chapter ask you to store and output file in a directory called "Output." You could create this directory using your operating systems graphical user interface, but let's see how to do this using the os module.

The mkdir() function creates a new directory at a specified path. For example, the following script creates a new directory called My Directory in the current working directory:

```
import os

os.mkdir("My Directory")
```

The above script creates the My Directory folder in the same directory that the script is stored in. You don't have to create the folder in the same directory as the script, however. For example, if you want to create a My Directory directory in the python-basics-exercises folder, you can use os.path.join() like this:

```
import os

path = "C:/Real Python/python-basics-exercises"
os.mkdir(os.path.join(path, "My Directory"))
```

In the above example, os.path.join() is used to create the full folder path by passing the two parts of the path as string arguments to this function. The os.path.join() function concatenates the two strings, making sure that the right number of slashes is included in between the two parts.

Instead of using os.path.join(), you could have simply concatenated path and "My Directory" by using a plus sign and adding an extra forward slash between the two strings like this:

```
os.mkdir(path + "/My Directory")
```

However, os.path.join() comes with the added benefit of Python automatically adding any slashes between the two path strings necessary to create a valid path. This is why it's a good idea to get into the habit of using this function to join path names together.

Sometimes you will retrieve part of a path name through your code and not know ahead of time if it includes an extra slash or not, and in these cases os.path.join() will be a necessity.

Note

If you're using Python 3.4+, you also have the option to work with object-oriented paths using pathlib. For an introduction to using pathlib, see the Python 3's pathlib Module: Taming the File System tutorial on realpython.com.

To remove a directory, use the os.rmdir() function. This function takes a string argument, just like os.mkdir() that represents the path to the folder you want to delete. The following script removes the "My Directory” folder from the “python-basics-exercises” directory:

```
import os

path = "C:/Real Python/python-basics-exercises"
os.rmdir(os.path.join(path, "My Directory"))
```

Now that you’ve seen some of the basics let’s start modifying files. We’ll use the following example: suppose you have copied an entire directory to a new location on your hard drive and want to rename every `*.gif` file in a folder to `*_backup.gif`.

To get a list of the files in the folder, you can use the `os.listdir()` function, which returns a list of file names found in the provided directory. You can use the string method `.endswith()` to check the file extension of each file name. Finally, use `os.rename()` to rename each file:

```
import os

path = "C:/Real Python/python-basics-exercises" \
       "/ch11-file-input-and-output/practice_files/images"

for file_name in os.listdir(path):
    if file_name.lower().endswith(".gif"):
        full_path = os.path.join(path, file_name)
        new_file_name = full_path[:-4] + "_backup.gif"
        os.rename(full_path, new_file_name)
```

Since `.endswith()` is case-sensitive, you must first convert `file_name` to lowercase using the `.lower()` string method. The `.lower()` method returns a string, so you can stack one method on top of another in the same line.

Subscripts are used to replace the file extension in the line `new_file_name = full_file_name[:-4]`. This trims the last four characters (the “.gif”) off of the full file name. Then the `"_backup.gif"` string is added to the end of the string.

Finally, the os.rename() function is used to rename the file. It takes two arguments, the first being the full original file name and the second being the new file name.

While looping over the paths in the list returned by os.listdir() gets the job done, Python provides a more efficient method for searching files and directors in its glob module. Let's take a look at that now.

The glob Module

The glob module contains tools to help match patterns of file names. For example, the glob.glob() function takes a string that uses "wildcard" characters, then returns a list of all possible matches.

For example, if you provide the file name pattern "*.txt" then you will be able to find any file names that match the ".txt" extension at the end:

```
>>> import glob
>>> glob.glob("*.txt")
['file1.txt', 'notest.txt', 'test.txt']
```

Or, as a more elaborate example, you can use this to batch-rename a list of files at once. The following script renames all .gif files in the given folder so that their filenames end in _backup.gif:

```
import glob
import os

path = "C:/Real Python/python-basics-exercises" \
       "/ch11-file-input-and-output/practice_files/images"

possible_files = os.path.join(path, "*.gif")

for file_name in glob.glob(possible_files):
    full_path = os.path.join(path, file_name)
    new_file_name = full_path[:-4] + "_backup.gif"
    os.rename(full_path, new_file_name)
```

When you provide a string with the full file path and an * to glob(), it returns a glob list of all possible GIF images in that particular directory.

Note

You may be wondering what the deal is with the glob module's funny name. It comes from the practice of calling file searching on UNIX bash systems "globbing."

You can also use glob() to search through subfolders. For instance, if you want to search for all of the PNG files that are in folders inside of the images/ folder, you can use the string pattern "*/*.png".:

```
import glob
import os

path = "C:/Real Python/python-basics-exercises" \
       "/ch11-file-input-and-output/practice_files/images"

possible_files = os.path.join(path, "*/*.png")

for file_name in glob.glob(possible_files):
    print(file_name)
```

Adding the string "*/*.png" to the path tells glob() to search for any files ending in .png that are inside of folders that can have any name (the first *). The forward slash used to separate the last folder tells glob() to only search in subfolders of the images/ directory.

Another special pattern-matching character that can be included in a glob pattern is a ? to stand for any one single character. For instance, searching for anything matching ??.gif returns GIF files that have a name that is exactly two characters long.

You can also specify ranges to search over by putting them in square brackets. The pattern [0-9] matches any single number from 0 through 9, and the pattern [a-z] matches any single letter. For instance, if you want to search for any GIF files that have the name "image" followed specifically by two digits, you would pass the pattern image[0-9][0-9].gif to glob().

Checking the Existence of Files and Folders

In previous examples, you saw how listdir() can be used to get a list of all files *and folders* in a given directory. If you need to alter every file in the images/ folder, you need to be careful not to alter folder names as well.

You can check whether or not a path is a file or a folder with os.path.isfile() and os.path.isdir(), both of which return True or False. For instance, the following script appends the string "folder" to the end of each folder name inside the images/ directory but not affect any of the files in images/:

```python
import os

path = "C:/Real Python/python-basics-exercises" \
       "/ch11-file-input-and-output/practice_files/images"

files_and_folders = os.listdir(path)

for folder_name in files_and_folders:
    full_path = os.path.join(path, folder_name)
    if os.path.isdir(full_path):
        os.rename(full_path, full_path + " folder")
```

Here os.path.isdir() is used to check if folder_name is actually a folder or not. In this case, it's either a valid path to a folder or a valid path to a file, but passing any string that isn't a valid folder path to os.path.isdir() returns False.

Another related function that can be especially useful for deciding whether or not a particular file needs to be created for the first time is `os.path.exists()`, which returns `True` or `False` depending on whether the file or folder specified already exists or not.

Traversing a Directory Structure

Sometimes you need to deal with more complicated directory structures. For example, you might need to get all the files in all subfolders of a particular directory.

You can do this with `os.walk()`. This function returns all the possible combinations of folder, subfolders, and file names as tuples that represent the paths to reach every file anywhere in a named root directory.

For example, the following displays every file in the `images/` folder and all of its subfolders:

```python
import os

path = "C:/Real Python/python-basics-exercises" \
       "/ch11-file-input-and-output/practice_files/images"

for current_folder, subfolders, file_names in os.walk(path):
    for file_name in file_names:
        print(os.path.join(current_folder, file_name))
```

The call to `os.walk()` creates an object that Python can loop over. At each step of the loop, it returns a different tuple that includes:

1. A particular folder
2. A list of the subfolders within that folder, and
3. A list of files within that folder.

In the above example, tuple unpacking is used in the outer `for` loop to loop over every possible combination of `(current_folder, subfolders, file_names)`, where `current_folder` may represent the `images/` folder or one of its subfolder.

In this case, you don't need anything in the `subfolders` list since looping through each of the `file_names` and joining them to `current_folder` gives the full path to every file.

In this section, you saw some of the most common cases involving functions from the `os` module. That said, there are many additional functions belonging to both the os module and the os.path module that can be used in various ways for accessing and modifying files and folders.

In the assignment after the review exercises, you'll practice a couple more of these functions: deleting files and folders by passing them to the `os.remove()` function and getting the size of a file in bytes by passing it to the `os.path.getsize()` function.

Review Exercises

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

1. Display the full paths of all of the files and folders in the `images/` folder by using `os.listdir()`.
2. Display the full paths of any `*.png` files in the `images/` folder by using `glob.glob()`.
3. Rename any `*.png` files in the `images/` folder and its subfolders to `*_backup.png` by using `os.walk()`; in case you mess things up beyond repair, there is a copy of the `images/` folder in the backup folder.
4. Make sure that your last script worked by using `os.path.exists()` to check that the renamed files now exist (by providing `os.path.exists()` with the full path to each of these files).
5. Create a folder called "Output" in your current working directory, and add a new `python.txt.` file to it containing the string "I was put here by Python!"

Leave feedback on this section »

11.3 Challenge: Use Pattern Matching to Delete Files

Write a script `remove_files.py` that looks in the Chapter 11 `practice_files` folder named `little pics` as well all of its subfolders. The script should use `os.remove()` to delete any JPG file found in any of these folders if the file is less than 2 KB (2,000 bytes) in size.

You can supply the `os.path.getsize()` function with a full file path to return the file's size in bytes. Check the contents of the folders before running your script to make sure that you delete the correct files.

You should only end up removing the files named "to be deleted.jpg" and "definitely has to go.jpg"—although you should only use the file extensions and file sizes to determine this.

If you mess up and delete the wrong files, there is a folder named `backup` that contains an exact copy of the `little pics` folder and all its contents so that you can copy these contents back and try again.

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

Leave feedback on this section »

11.4 Read and Write CSV Data

The types of files you likely deal with in everyday life are often more complicated than plain text files. To modify the contents of these files (rather than just copy, rename or delete them), you need some more complex machinery.

One common way of storing text data is in CSV files. "CSV" stands for Comma-Separated Value because each entry in a row of data is usually separated from other entries by a comma. For example, the contents of the file named `wonka.csv` in the Chapter 11 `practice_files` folder look like this:

```
First name,Last name,Reward
Charlie,Bucket,"golden ticket, chocolate factory"
Veruca,Salt,squirrel revolution
Violet,Beauregarde,fruit chew
```

The first line of the file contains three column names: “First name,” “Last name,” and “Reward.” Each line represents a row of data, including the first row, which is a “header” row that tells us what each entry represents. The entries appear in the same order for each row, with each entry separated from others by commas.

Notice that “golden ticket, chocolate factory” is in quotes. This is because it contains a comma, but this comma isn’t meant to separate one entry from another. There is no set standard for how to write out CSV files, and this particular file was created with Microsoft Excel, which added the quotation marks around the entry containing a comma.

Note

If you open the `wonka.csv` practice file, it will most likely be opened automatically by Excel, OpenOffice Calc, LibreOffice Calc, or a similar program. All of these programs can read and write CSV data, which is one reason why this format is so useful.

As long as you don’t need to track the characteristics of a data file such as formatting and colors, it’s usually easiest to export data to a CSV file before working with the data from a script. CSV files can also be useful for importing or exporting data from systems such as SQL databases.

Python has a built-in csv module that makes it nearly as easy to read and write CSV files as any other sort of text file.

Reading Data From a CSV File

Let’s start with a basic example that reads from the `wonka.csv` file and display its contents:

```
import csv
import os

path = "C:/Real Python/python-basics-exercises" \
       "/ch11-file-input-and-output/practice_files"

with open(os.path.join(path, "wonka.csv"), "r") as my_file:
    reader = csv.reader(my_file)
    for row in reader:
        print(row)
```

After opening the `wonka.csv` file with `open()`, a CSV file reader object is created by passing `my_file` to the `csv.reader()` function. Note that the `my_file` object must be passed to `csv.reader()`, *not* the file path.

Next, a `for` loop is used to loop over the rows of data in the `reader` object, and each row is displayed with `print()`:

```
['First name', 'Last name', 'Reward']
['Charlie', 'Bucket', 'golden ticket, chocolate factory']
['Veruca', 'Salt', 'squirrel revolution']
['Violet', 'Beauregarde', 'fruit chew']
```

To get a single row of data from the CSV reader object, you can use the `next()` function. This function is usually used to skip over a row of "header" data.

For instance, to read in and store all the information in `wonka.csv`, except the first row, add the line `next(reader)` after opening the CSV file. Then loop through the remaining rows as usual.

If you know what fields to expect from the CSV ahead of time, you can even unpack them from each row into new variables in a single step:

```
import csv
import os
```

```python
path = "C:/Real Python/python-basics-exercises" \
       "/ch11-file-input-and-output/practice_files"

with open(os.path.join(path, "wonka.csv"), "r") as my_file:
    reader = csv.reader(my_file)
    next(reader)
    for first_name, last_name, reward in reader:
        print(f"{first_name} {last_name} got: {reward}")
```

After skipping the first header row with the `next()` function, the three values in each row are assigned to the three separate variables `first_name`, `last_name` and `reward`. These variables are then used to print the contents of each row with a formatted string. This produces the following output:

```
['First name', 'Last name', 'Reward']
Charlie Bucket got: golden ticket, chocolate factory
Veruca Salt got: squirrel revolution
Violet Beauregarde got: fruit chew
```

> **Note**
>
> The first line of this output is generated by the call to the `next()` function. This only happens when running the code in the interactive window. If you save the code to a script and run it, you will not see the list of column headers.

The commas in CSV files are called **delimiters** because they are the character used to separate, or delimit, different pieces of the data.

Sometimes a CSV file uses a different character as a delimiter, especially if there are many commas already contained in the data. For instance, the following example reads data from the `tabbed wonka.csv`, which uses tabs instead of commas as delimiters and looks like this:

```
First name  Last name   Reward
Charlie     Bucket      golden ticket, chocolate factory
```

```
Veruca      Salt        squirrel revolution
Violet      Beauregarde fruit chew
```

You can read files like this using the `csv` module just as easily as CSV files, but you'll need to specify what character to use as the delimiter:

```python
import csv
import os

path = "C:/Real Python/python-basics-exercises" \
       "/ch11-file-input-and-output/practice_files"

with open(os.path.join(path, "tabbed wonka.csv"), "r") as my_file:
    reader = csv.reader(my_file, delimiter="\t")
    next(reader)
    for row in reader:
        print(row)
```

In the above example, the special character `\t`, which represents the "tab" character, is assigned it to the `delimiter` argument of the `csv.reader()` function.

Writing Data to a CSV File

Writing CSV files is accomplished using the `csv.writer()` function. Just as rows of data are read from CSV files as lists of strings, you first need to structure the rows to be written as lists of strings. These lists can then be written to file with the `.writerow()` method, as in the following example:

```python
import csv
import os

path = "C:/Real Python/python-basics-exercises" \
       "/ch11-file-input-and-output/practice_files"

with open(os.path.join(path, "movies.csv"), "w") as my_file:
    writer = csv.writer(my_file)
    writer.writerow(["Movie", "Rating"])
    writer.writerow(["Rebel Without a Cause", "3"])
    writer.writerow(["Monty Python's Life of Brian", "5"])
    writer.writerow(["Santa Claus Conquers the Martians", "0"])
```

First, a new file `movies.csv` is opened in write mode by passing `"w"` to the second argument of `open()`. Individual rows are then written to the CSV file using the `.writerow()` method of the CSV file writer object.

You also use the `.writerows()` method, which takes a list of rows, to write all the rows in a single line:

```
import csv
import os

path = "C:/Real Python/python-basics-exercises" \
       "/ch11-file-input-and-output/practice_files"

ratings = [["Movie", "Rating"],
           ["Rebel Without a Cause", "3"],
           ["Monty Python's Life of Brian", "5"],
           ["Santa Claus Conquers the Martians", "0"]]

with open(os.path.join(path, "movies.csv"), "w") as my_file:
    writer = csv.writer(my_file)
    writer.writerows(ratings)
```

Special Purpose Tools

Python's `csv` module provides a simple yet powerful interface for working with CSV files. Sometimes, though, you need to work with files destined to be opened by a user with a particular program.

There are a number of packages designed to interact with Microsoft Excel documents (although they all have limitations), including xlrd and xlwt for reading and writing basic Excel files, openpyxl for manipulating Excel 2010 files, and XlsxWriter for creating *.xlsx* files from scratch.

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 reads in the data from the CSV file `pastimes.csv` located in the Chapter 11 practice files folder, skipping over the header row.
2. Display each row of data (except for the header row) as a list of strings.
3. Add code to your script to determine whether or not the second entry in each row (the “Favorite Pastime”) converted to lower-case includes the word “fighting” using the string methods `.find()` and `.lower()`.
4. Use the list `.append()` method to add a third column of data to each row that takes the value “Combat” if the word “fighting” is found and takes the value “Other” if neither word appears.
5. Write out a new CSV file `categorized pastimes.csv` to the Output folder with the updated data that includes a new header row with the fields “Name,” “Favorite Pastime,” and “Type of Pastime.”

Leave feedback on this section »

11.5 Challenge: Create a High Scores List

Write a script `high_scores.py` that reads in a CSV file of users’ scores and displays the highest score for each person. The file you will read is named `scores.csv` and is located in the Chapter 11 `practice_files` folder.

You should store the high scores as values in a dictionary with the associated names as dictionary keys. This way, as you read in each row of data. If the name already has a score associated with it in the dictionary, you can compare these two scores and decide whether or not to replace the “current” high score in the dictionary.

Use the `sorted()` function on the dictionary’s keys to display an ordered list of high scores, which should match this output:

```
Empiro 23
L33tH4x 42
LLCoolDave 27
MaxxT 25
Misha46 25
O_O 22
johnsmith 30
red 12
tom123 26
```

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

Leave feedback on this section »

11.6 Challenge: Split a CSV file

Write a script that takes three required command line arguments—`input_file`, `output_file`, and the `row_limit`. From those arguments, split the input CSV into multiple files based on the `row_limit` argument.

Arguments:

1. `-i`: input file name
2. `-o`: output file name
3. `-r`: row limit to split

Default settings:

1. `output_path` is the current directory

2. headers are displayed on each split file
3. the default delimiter is a comma

Example usage:

```
$ # Split csv by every 100 rows
$ python csv_split.py -i input.csv -o output -r 100
```

Before you start coding, stop for a minute and read over the directions again. If you're having trouble following them, take some notes. What makes this assignment so tricky is that it has many moving pieces. However, if you can break them down into manageable chunks, then the process will be much easier. Let's look at it together.

1. You first need to grab the command line arguments. I recommend using the argparse library for this. Once obtained, you should validate the arguments to ensure that (a) the input file exists and (b) the number of rows in the input file is greater than the row limit to split. Make sure that each of your functions does only one thing. Think about how many functions you need for this first step.
2. If the validation passes, the program should continue. If not, the program ends, displaying an error message.
3. Next, you need to split up the CSV file into separate "chunks" based on the `row_limit` argument. In other words, if your input CSV file has 150 rows (minus the header) and the `row_limit` is set to 50, then there should be three chunks (and when you create your output CSV files, each chunk will have 50 rows + the header). There are many ways to create each chunk. In this example, it's probably easiest to create a separate `list` for each chunk containing the appropriate # of rows from the input CSV file.
4. You need to have separate output files (one for each chunk) that have some sort of naming convention that makes sense. You could use a timestamp. Alternatively, you could add the chunk number to each filename—for example, `output-file-name_chunk-number.csv`. Each file must have a `.csv` extension as well as the headers. Add each chunk to the appropriate output file.

5. Finally, output information to the user indicating the file name and the # of rows for each chunk. Format this in an appropriate, legible manner.

Try this out on your own before looking at the answer. You should be able to get through the first two steps on your own. The remaining steps are a bit more difficult. If you found this assignment easy, try to add additional functionality to your program, such as the ability to include or exclude the headers from each file, splitting the input CSV by the column # (or name) instead of by row.

Note

Need a hint? Here are some recommendations for how to break down steps 3, 4, and 5:

In step 3, you should open the CSV file and create a list of lists where each list is a row in the spreadsheet. Remove the header and save it, since you'll need to add it to each chunk.

Use a `for` loop to loop through the list of lists and create a chunk that contains the rows from a starting row number to an ending row number.

The ending row number is the `row_limit`. If there are not enough rows left—that is, the rows remaining is less than the `row_limit`—then make sure to add all remaining rows to the final chunk.

Steps 4 and 5 are best kept in the same `for` loop. Create a new, unique output file name, add the headers to each chunk, then add each chunk to the file. Output the information to the user.

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

Leave feedback on this section »

11.7 Summary and Additional Resources

In this chapter, you learned how to work with files in Python. You can open a file for both reading and writing with the `open()` function, which takes two arguments: a string containing the path to the file to be opened, and a string containing the mode to open the file in, such as `"r"` for read mode and `"w"` for write mode.

You also learned how to work with file paths using the `os` module. You can join parts of paths together using the `os.path.join()` function, and create and delete directories using `os.mkdir()` and `os.rmdir()`. You can iterate over files in a directory and its subdirectories using the `glob` module.

Finally, you learned how to work with comma separated value (CSV) files, a standard file type for storing categorized data, using the `csv` module. You saw how to read data in a CSV file using `csv.reader()`, which returns each row in the CSV file as a list, and how to write rows of data to a CSV file using `csv.writer()`.

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-11

Additional Resources

To get even more practice working with files, check out these resources:

- Reading and Writing Files in Python (Guide)
- Working With Files in Python
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 12

Installing Packages With Pip

Up to this point, you have been working within the bounds of the **Python standard library**. In the remaining half of this course, you will work with various toolkits, or **packages**, that are not included with Python by default.

Many languages come with, or offer as a separate download, a **package manager** that automates the process of installing, upgrading, and removing third-party packages. Python is no exception.

The *de facto* package manager for Python is called `pip`. Historically, `pip` had to be downloaded and installed separately from Python, but, as of Python 3.4, it is included with most distributions of the language.

In this chapter, you will learn:

- How to install and manage third-party packages with `pip`
- What the benefits—and risks—of third-party packages are

Let's go!

Leave feedback on this section »

12.1 Install a Third-Party Package With Pip

Most likely, `pip` was included when you installed Python. To determine whether or not you have `pip` installed on your machine, run the following in a terminal:

```
$ pip3 --version
```

> **Important**
>
> On Windows, the above command most likely returns an error, and you should instead run:
>
> ```
> $ pip --version
> ```
>
> > **Important**
> >
> > Remember, the `$` is just the command line prompt, so you do not need to type that in.
>
> On macOS and Linux, where system Python may be a version of Python 2, it is important to always invoke `pip` using the `pip3` command. Otherwise, installing packages with the `pip` command puts them in your Python 2 environment.

You should see output that tells you the version number and where `pip` is installed:

```
pip 10.0.1 from c:usersdaveaappdatalocalprogramspythonpython37
libsite-packagespip (python 3.7)
```

Note

If you see a different version number than the one shown in the above output, that's okay. If you see a smaller version number, you can upgrade `pip` with the following command on Windows:

```
$ python -m pip install --upgrade pip
```

Or, on macOS and Linux:

```
$ pip3 install --upgrade pip
```

If your operating system tells you that `pip3` is an unrecognized command, then `pip` did not come with your Python distribution for some reason. This usually indicates a deeper issue with your Python installation.

You may want to review the how to install Python in Chapter 2.

List Your Installed Packages

You can use `pip` to show you which packages you have installed in your environment. Let's take a peek at what is currently available. Try typing the following into your terminal:

```
$ pip3 list
```

If you haven't already installed any packages into your environment, which will be the case if you started this course with a fresh Python 3.7 installation, you should see something like the following:

```
Package    Version
---------- -------
pip        10.0.1
setuptools 39.0.1
```

As you can see, there isn't much here. You see `pip` itself listed, because `pip` is a package. You may also see `setuptools`. This is a package used to create your own Python package distributions.

Whenever you install a package with `pip`, it will show up in this list. You can always use `pip3 list` to see which packages, and which version of each package, you currently have installed in your environment.

Install the `requests` Package

Let's install your first Python package! For this exercise, you will install the `requests` package, which is one of the most popular Python packages ever created. In your terminal, type the following:

```
$ pip3 install requests
```

While `pip` is installing the `requests` package, you will see a bunch of output:

```
Collecting requests
  Downloading https://.../requests-2.19.1-py2.py3-none-any.whl (91kB)
    100% |................................| 92kB 1.9MB/s
Collecting chardet<3.1.0,>=3.0.2 (from requests)
  Downloading https://.../chardet-3.0.4-py2.py3-none-any.whl (133kB)
    100% |................................| 143kB 9.1MB/s
Collecting certifi>=2017.4.17 (from requests)
  Downloading https://.../certifi-2018.4.16.py3-none-any.whl (150kB)
    100% |................................| 153kB 5.0MB/s
Collecting idna<2.8,>=2.5 (from requests)
  Downloading https://.../idna-2.7-py2.py3-none-any.whl (58kB)
    100% |................................| 61kB 12.8MB/s
Collecting urllib3<1.24,>=1.21.1 (from requests)
  Downloading https://.../urllib3-1.23-py2.py3-none-any.whl (133kB)
    100% |................................| 143kB 16.5MB/s
Installing collected packages: chardet, certifi, idna,
urllib3, requests
Successfully installed certifi-2018.4.16 chardet-3.0.4 idna-2.7
  requests-2.19.1 urllib3-1.23
```

Note

The formatting of the above output has been altered so that it fits nicely on the page, so the output that you see may look slightly different.

Notice that `pip` first tells you that it is "Collecting requests." You will see the URL that `pip` is using to install the package from, as well as a progress bar indicating the progress of the download.

After that, you will see that `pip` installs four more packages: `chardet`, `certifi`, `idna` and `urllib3`. These packages are **dependencies** of `requests`. That means that `requests` requires these packages to be installed in order for it to work properly.

Once `pip` is done installing `requests` and its dependencies, run `pip3 list` in your terminal again. You should now see the following list:

```
$ pip3 list
Package    Version
---------- ---------
certifi    2018.4.16
chardet    3.0.4
idna       2.7
pip        10.0.1
requests   2.19.1
setuptools 39.0.1
urllib3    1.23
```

Show Package Details

Now that you have installed the `requests` package, you can use `pip` to tell you a little bit more about what the package is:

```
$ pip3 show requests
Name: requests
Version: 2.19.1
```

```
Summary: Python HTTP for Humans.
Home-page: http://python-requests.org
Author: Kenneth Reitz
Author-email: me@kennethreitz.org
License: Apache 2.0
Location: /home/david/stuff/python/venv/lib/python3.7/site-packages
Requires: urllib3, chardet, idna, certifi
Required-by:
```

The `pip3 show` command displays some information about a package you have installed, including the author's name and email, and a home page you can navigate to in your internet browser to learn more about what the package does.

The `requests` package is used for making HTTP requests from a Python program. It is extremely useful in a variety of domains, and is a requirement of a large number of other Python packages.

Uninstall the `requests` Package

If you can install a package with `pip`, it only makes sense that you can also uninstall a package. Let's uninstall the requests package now.

To uninstall `requests`, type the following into your terminal:

```
$ pip3 uninstall requests
```

Important

If you already have projects that use `requests` or one of its dependencies, you may not want to run the commands in the remainder of this section.

You will immediately see the following prompt:

```
Uninstalling requests-2.19.1:
  Would remove:
    /home/david/venv/lib/python3.7/site-packages/requests.dist-info/*
    /home/david/venv/lib/python3.7/site-packages/requests/*
Proceed (y/n)?
```

Before `pip` actually removes anything from your computer, it asks for your permission first. How considerate!

Type `y` and press `Enter` to continue. You should then see the following message confirming that `requests` was removed:

```
Successfully uninstalled requests-2.19.1
```

Take a look at your package list again:

```
$ pip3 list
Package    Version
---------- ---------
certifi    2018.4.16
chardet    3.0.4
idna       2.7
pip        10.0.1
setuptools 39.0.1
urllib3    1.23
```

Notice that `pip` uninstalled `requests`, but it didn't remove any of its dependencies! This behavior is a feature, not a bug.

Imagine that you have installed several packages into your environment with `pip`, some of which share dependencies. If `pip` uninstalled a package *and* its dependencies, it would render any other package requiring those dependencies unusable!

For now, though, go ahead and remove the remaining packages by running `pip3 uninstall` for each:

```
$ pip3 uninstall certifi
$ pip3 uninstall chardet
$ pip3 uninstall idna
$ pip3 uninstall urllib3
```

Note

You can also uninstall all four packages with a single command:

```
$ pip3 uninstall certifi chardet idna urllib3
```

When you are done, verify that everything has been removed by running `pip3 list` again. You should see the same list of packages you saw when you first started:

```
Package    Version
---------- -------
pip        10.0.1
setuptools 39.0.1
```

Python's ecosystem of third-party packages are one of its greatest strengths. These packages allow Python programmers to be highly productive and create full-featured software much more quickly than can be done in, say, a language like C++.

That said, using third-party packages in your code introduces several concerns that must be addressed with care and responsibility. You'll learn about some of the pitfalls associated with third-party packages in the next section.

Leave feedback on this section »

12.2 The Pitfalls of Third-Party Packages

The beauty of third-party packages is that they give you the ability to add functionality to your project without having to implement everything from scratch. This offers massive boosts in productivity.

But with great power comes great responsibility. As soon as you include someone else's package in your project, you are placing an enormous amount of trust in those responsible for developing and maintaining the package.

By using a package you did not develop, you lose control over certain aspects of your project. In particular, the maintainers of a package may release a new version that introduces changes that are incompatible with the version you use in your project.

By default, `pip` installs the latest release of a package, so if you distribute your code to some one else and they install a newer version of a package required by your project, they may not be able to run your code.

This presents a significant headache, for both the end user and yourself. Fortunately, Python comes with a fix for this all-to-common problem: virtual environments.

A virtual environment creates an isolated and, most importantly, reproducible environment that you can use to develop a project. The environment can contain a specific version of Python, as well as specific versions of your project's dependencies.

When you distribute your code to someone else, they can reproduce this environment and be confident that, barring any issues with their own Python installation and operating system, they can run your code without error.

Do you want to learn more about managing your projects dependencies? Check out Real Python's Managing Python Dependencies With Pip and Virtual Environments course. In it you will learn how to:

- Install, use, and manage third-party Python packages with the "pip" package manager on Windows, macOS, and Linux, in more detail than presented here.
- Isolate project dependencies with so-called virtual environments to avoid version conflicts in your Python projects.
- Apply a complete 7-step workflow for finding and identifying qual-

ity third-party packages to use in your own Python projects (and justifying your decisions to your team or manager.)

- Set up repeatable development environments and application deployments using the "pip" package manager and requirements files.

Managing Python Dependencies With Pip and Virtual Environments is a great next step when you have completed this book.

Leave feedback on this section »

12.3 Summary and Additional Resources

In this chapter, you learned how to install third-party packages using Python's package manager `pip`. You saw several useful `pip` commands, including `pip install`, `pip list`, `pip show` and `pip uninstall`.

You also learned about some of the pitfalls associated with third party packages. Not every package that is downloadable with `pip` is a good choice for your project. Since you do not have control over the code in the package you install, you must trust that the package is safe and will work well for the users of your program.

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-12

Additional Resources

To learn more about managing third-party packages, you can check out these resources:

- Managing Python Dependencies Course
- Python Virtual Environments: A Primer
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 13

Creating and Modifying PDF Files

PDF files have become a sort of necessary evil these days. Despite their frequent use, PDFs can be difficult to work with.

Fortunately, the Python ecosystem has some great packages for reading, manipulating, and creating PDF files!

In this chapter, you will learn how to:

- Read text from a PDF
- Rotate and crop pages in a PDF file
- Create a PDF file from scratch

Let's get started!

Leave feedback on this section »

13.1 Work With the Contents of a PDF File

In this section, you will learn how to read the contents of a PDF file using the PyPDF2 package. Before you can do that, though, you need to install PyPDF2 with `pip`:

```
$ pip3 install PyPDF2
```

When the installation is complete, you can verify it by running the following in your terminal:

```
$ pip3 show PyPDF2
Name: PyPDF2
Version: 1.26.0
Summary: PDF toolkit
Home-page: http://mstamy2.github.com/PyPDF2
Author: Mathieu Fenniak
Author-email: biziqe@mathieu.fenniak.net
License: UNKNOWN
Location: c:\users\david\python37\lib\site-packages
Requires:
Required-by:
```

Pay particular attention to the version information. At the time of writing, the latest version of PyPDF2 is 1.26.0. By default, `pip` installs the latest version of a package available on pypi.org.

If you see a different version number—especially if the version is labeled 2.0.0 or higher—there is no guarantee that the code examples in this chapter will work properly.

Note

If you want to make sure that you are using version 1.26.0, you can run:

```
$ pip3 install PyPDF2==1.26.0
```

This tells `pip` to install version 1.26.0 rather than the latest version available. If you have an different version installed, running the command above will remove the old version and install version 1.26.0.

Now that you have PyPDF2 installed let's start working with some PDF files!

Open a PDF File

Let's get started by reading in some basic information from a sample PDF file. For this example, we will use the first couple of chapters of Jane Austen's *Pride and Prejudice*.

Note

The sample PDF used in the following examples can be found in the `\ch13-interact-with-pdf-files\practice_files` directory of the `python-basics-exercises` folder. The file is called `Pride and Prejudice.pdf`.

If you do not have the exercise solutions and practice files downloaded, you can obtain a copy here.

As a last resort, you can download a full PDF of "Pride and Prejudice" free of charge from Project Gutenberg.

The primary interface for reading information from a PDF file with PyPDF2 is through the `PdfFileReader` object. To use it, you need to import it from the `PyPDF2` package. Open IDLE's interactive window and type the following:

```
>>> from PyPDF2 import PdfFileReader
```

Now that you have the PdfFileReader class imported from the PyPDF2 package, you can open a PDF file for reading by passing a string containing the path to a PDF file to the PdfFileReader constructor:

```
>>> file_path = "C:/python-basics-exercises/" \
...             "ch13-interact-with-pdf-files/" \
...             "practice_files/Pride and Prejudice.pdf"
>>> input_pdf = PdfFileReader(file_path)
```

Note

In the above example, you may need to change the file_path to an appropriate path for your system.

Read the Document Meta-Data

Now that you have the file open as a PdfFileReader instance, you can start to gather some information about it. For example, the PdfFileReader.getNumPages() method returns the number of pages contained in the PDF file:

```
>>> input_pdf.getNumPages()
234
```

Note

You may have noticed that the `.getNumPages()` method does not follow the naming convention for variables in function names laid out by PEP 8. Remember, PEP 8 is a set of guidelines. As far as Python is concerned, writing your function names in camel case (`camelCase`) is perfectly acceptable.

The PyPDF2 package has its roots all the way back in 2005, a mere four years after the PEP 8 guidelines were released. At that time, many programmers using Python were migrating from other languages, such as Java, where camel case was the *de facto* style for function names.

When you work with Python, you will undoubtedly encounter code bases that break the PEP 8 recommendation for snake case (`snake_case`).

You can also read some basic information, called **meta-data**, about the document with the `.getDocumentInfo()` method:

```
>>> document_info = input_pdf.getDocumentInfo()
>>> document_info
{'/Title': 'Pride and Prejudice, by Jane Austen',
 '/Author': 'Chuck',
 '/Creator': 'Microsoft® Office Word 2007',
 '/CreationDate': 'D:20110812174208',
 '/ModDate': 'D:20110812174208',
 '/Producer': 'Microsoft® Office Word 2007'}
```

The object by `.getDocumentInfo()` is looks like a dictionary, but it is actually a special object whose type is `DocumentInformation`:

```
>>> type(document_info)
<class 'PyPDF2.pdf.DocumentInformation'>
```

`DocumentInformation` objects have attributes that allow you to access the information they contain. For instance, to get the document's title, you use the `.title` attribute:

```
>>> document_info.title
'Pride and Prejudice, by Jane Austen'
```

The `.author` attribute returns the name of the author:

```
>>> document_info.author
'Chuck'
```

Who is Chuck?

The information contained in the `DocumentInformation` object is whatever is stored in the PDF file's meta-data, and may not reflect, for example, the actual author of the content. In this case, Chuck is probably the PDF file's creator.

Extract Text From a Page

Recall that the `input_pdf` file has 243 pages. Each page has an index between 0 and 242. To extract the text from a page in the PDF, pass the page's index to the `PdfFileReader.getPage()` method:

```
>>> page0 = input_pdf.getPage(0)
```

`.getPage()` returns a `PageObject`:

```
>>> type(page0)
<class 'PyPDF2.pdf.PageObject'>
```

You can extract the page's text with the `PageObject.extractText()` method:

```
>>> page0.extractText()
'\n \nThe Project Gutenberg EBook of Pride and Prejudice, by Jane
Austen\n \n\nThis eBook is for the use of anyone anywhere at no cost
and with\n \nalmost no restrictions whatsoever.  You may copy it,
give it away or\n \nre\n-\nuse it under the terms of the Project
```

```
Gutenberg License included\n \nwith this eBook or online at
www.gutenberg.org\n \n \n \nTitle: Pride and Prejudice\n \n
\nAuthor: Jane Austen\n \n \nRelease Date: August 26, 2008
[EBook #1342]\n\n[Last updated: August 11, 2011]\n \n \nLanguage:
Eng\nlish\n \n \nCharacter set encoding: ASCII\n \n \n***
START OF THIS PROJECT GUTENBERG EBOOK PRIDE AND PREJUDICE ***\n \n
\n \n \n \nProduced by Anonymous Volunteers, and David Widger\n
\n \n \n \n \n \n \nPRIDE AND PREJUDICE \n \n \nBy Jane
Austen \n \n\n \n \nContents\n \n'
```

So, once you have a `PdfFileReader` object, there are two steps to extracting the text: first, get a `PageObject` with `PdfFileReader.getPage()`, and then extract the text with `PageObject.extractText()`. If that seems a little convoluted, you can do the entire operation in a single line by **chaining** the method calls together, like this:

```
>>> page0_text = input_pdf.getPage(0).extractText()
```

You can chain these together because Python understands that each item invoked to the right of a `.` applies to the result of evaluating everything to the left of it. Python executes each part of the expression `input_pdf.getPage(0).extractText()` from left-to-right.

So, first Python executes `input_pdf.getPage(0)`, which returns a `PageObject`. Then Python executes `.extractText()` on this `PageObject` and returns the text on the page as a string.

Note

The output in the example above has been formatted to fit better on this page. The output you see in the interactive window may look different.

Every `PdfFileReader` object has a `.pages` attribute that returns the a Python list of `PageObject` objects. You can iterate over this list with a `for` loop and do something with each page in the PDF. For example, you can print the text on each like this:

```
>>> for page in input_pdf.pages:
...     print(page.extractText())
...
```

Let's put everything you've learned together and write a program that extracts all of the text from the `Pride and Prejudice.pdf` file and saves it to a `.txt` file.

Putting It All Together

Open a new script window in IDLE. Type out the script below:

```python
from PyPDF2 import PdfFileReader

# Change the path below to the correct path for your system.
path = "C:/python-basics-exercises/ch13-interact-with-pdf-files/" \
       "practice_files/Pride and Prejudice.pdf"

# 1
input_pdf = PdfFileReader(path)
title = input_pdf.getDocumentInfo().title
num_pages = input_pdf.getNumPages()

# 2
with open("Pride and Prejudice.txt", "w") as output_file:
    output_file.write(f"{title}\n")
    output_file.write(f"Number of pages: {num_pages}\n\n")

    # 3
    for page in input_pdf.pages:
        text = page.extractText()
        output_file.write(text)
```

Let's break that down.

1. First, you assign a new `PdfFileReader` instance to the `input_pdf` variable. The next two lines get the document title and the total number of pages and assign these values to the `title` and `num_pages` vari-

ables, respectively.

2. Next, the script opens a new file `Pride and Prejudice.txt` in `"w"`, or "write," mode using the `with` statement you learned in Chapter 11. The new file is assigned to the `output_file` variable. The first two lines in the `with` block write the title and number of pages of the PDF to the text file.
3. Finally, The `for` loop iterates over each page in the PDF. At each step in the loop, the `page` variable is assigned to a `PageObject` containing the data for the next page in the PDF file. Inside of the loop's body, the text from each page is extracted with `page.extracteText()` and written to the `output_file`.

When you save and run the script above, a new file called `Pride and Prejudice.txt` is created in your current working directory that contains the full text of the `Pride and Prejudice.pdf` document. Check it out!

Extract a Portion of a PDF

Sometimes you need to extract a portion of an existing PDF and save it to a new PDF file. To write to a new PDF file, you use the `PdfFileWriter` object from `PyPDF2`.

In the interactive window again, start by importing both the `PdfFileReader` and `PdfFileWriter` objects:

```
>>> from PyPDF2 import PdfFileReader, PdfFileWriter
```

Next, open the `Pride and Prejudice.pdf` file, just like you did before. Remember, you may need to change the file path so that it works on your system.

```
>>> file_path = "C:/python-basics-exercises/" \
...             "ch13-interact-with-pdf-files/" \
...             "practice_files/Pride and Prejudice.pdf"
>>> input_pdf = PdfFileReader(file_path)
```

Now that you have the input file open, you can create a new `PdfFileWriter` instance:

```
>>> output_pdf = PdfFileWriter()
```

In this example, you will get the first page of the `input_pdf`, which is the cover page of the `Pride and Prejudice.PDF` file, and then add this page to the `output_pdf`. You already know how to get the first page of a PDF—just use `.getPage(0)`. To add a page to a PDF, use the `PdfFileWriter.addPage()` method. Type the following into the interactive window:

```
>>> cover_page = input_pdf.getPage(0)
>>> output_pdf.addPage(cover_page)
```

The `PageObject` returned by `input_pdf.getpage(0)` is assigned to the `cover_page` variable, which is then passed to `output_pdf.addPage()` to add the cover page to the output PDF. However, you haven't created a new PDF file on your system yet. Right now, the new file exists only in memory.

To save the new PDF file to disk, you need to open a new output file with the `open()` function in `"wb"`, or "write binary," mode. Just like you have to read a PDF in `"rb"` mode, you must write a PDF in `"wb"` mode. Once you have the output file open, you can use the `.write()` method to write the contents of `output_pdf` with the `PdfFileWriter.write()` method:

```
>>> with open("portion.pdf", "wb") as output_file:
...     output_pdf.write(output_file)
```

You now have a new PDF file saved in your current working directory with the name `portion.pdf` that contains the cover page of the `Pride and Prejudice.pdf` file. Pretty cool!

> **Note**
>
> Do not confuse the `.write()` method of a `PdfFileWriter()` object with the `.write()` method of a `file` object created with the `open()` function.
>
> For example, the statement `output_file.write(output_pdf)` will *not* write the contents of `output_pdf` to the `output_file`.

In the next section, you will learn how to manipulate PDF files by rotating and splitting pages. Before moving on, make sure you are comfortable with the techniques in this section. To help reinforce what you have learned, give the following review exercises a shot.

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 opens the file named `The Whistling Gypsy.pdf` from the Chapter 13 practice files directory, then displays the title, author, and total number of pages in the file.
2. Extract the full text of `The Whistling Gypsy.pdf` into a `.txt` file.
3. Save a new version of `The Whistling Gypsy.pdf` that does not include the cover page.

Leave feedback on this section »

13.2 Manipulate PDF Files

Besides extracting text and pages from a PDF file, other common operations with PDF files is rotating, cropping, and merging pages. While you can perform these operations manually with software, this isn’t always a practical solution. Suppose you need to crop the pages in 10,000 PDF files. You can automate these tasks with PyPDF2!

In this section, you'll learn how to rotate, crop and merge pages in a PDF file programmatically using the PyPDF2 package.

Rotating Pages

Let's start by learning how to rotate pages. This is a surprisingly common problem! For this example, we'll use the `ugly.pdf` file in the `python-basics-exercises/ch13-interact-with-pdf-files/practice_files` folder. If you haven't already, you can download this from the supporting material downloads page.

The `ugly.pdf` file contains a lovely version of Hans Christian Andersen's *The Ugly Duckling*, except that every odd-numbered page is rotated counterclockwise by ninety degrees. To correct this problem, you need to use the `PageObject.rotateClockwise()` method. This method takes an integer argument, in degrees, and rotates the page clockwise by that many degrees. For example, `.rotateClocwise(90)` rotates a PDF page clockwise by 90 degrees.

To rotate every other page in the `ugly.pdf` fileTo do so, you need to loop over every page in the PDF and rotate every page counterclockwise by 90 degrees. Open a new script window in IDLE and typo out the following:

```
import os

from PyPDF2 import PdfFileReader, PdfFileWriter

path = "C:/python-basics-exercises/" \
       "ch13-interact-with-pdf-files/practice_files"

# 1
input_file_path = os.path.join(path, "ugly.pdf")
input_pdf = PdfFileReader(input_file_path)
output_pdf = PdfFileWriter()

# 2
```

```python
num_pages = input_pdf.getNumPages()
for n in range(0, num_pages):
    page = input_pdf.getPage(n)
    if n % 2 == 0:
        page.rotateClockwise(90)
    output_pdf.addPage(page)

# 3
output_file_path = os.path.join(
    path, "output/The Conformed Duckling.pdf"
)
with open(output_file_path, "wb") as output_file:
    output_pdf.write(output_file)
```

Parts 1 and 3 of the above script open the `ugly.pdf` file for reading and output the result to a new file `The Conformed Duckling.pdf`. There is nothing new in this code, so let's focus on the `for` loop in section 2.

This loop starts off simple enough by iterating over the numbers 0 to one less than the total number of pages in `ugly.pdf`. In the body of the `for` loop, a new `Page` object is created at each step using the `.getPage()` method.

Next, an `if` statement is used to determine if `page_num` is even by checking whether or not `page_num % 2 == 0` is `True` or `False`. If it is `True`, then the page is rotated clockwise with the `.rotateClockwise()` method. Finally, the page object is added to the output PDF (2c).

Hold on, now! Why are the even-numbered pages being rotated if it is the *odd* numbered pages that are rotated incorrectly in the PDF?

Remember: the `for` loop starts with the number 0, and `.getPage(0)` returns the *first* page of the PDF! So even numbers in the loop correspond to odd-numbered pages in the PDF. Confusing, right? With practice, this mismatch becomes easier to deal with mentally.

Cropping Pages

With PyPDF2, you can also crop a page in a PDF file. This allows you to split PDF pages into multiple pages, or save partial sections of pages into their own PDF.

For example, open up the `half and half.pdf` file located in the `python-basics-exercises/ch13-interact-with-pdf-files/practice_files` folder. Each page in this PDF has two columns. You can use PyPDF2's cropping functionality to split each page of this PDF into two pages, one for each column.

To do that, you'll need to make use of the `PageObject.mediaBox` attribute. A `mediaBox` represents a rectangular area that defines the boundaries of a page. Before you jump into splitting the PDF, let's use IDLE's interactive window to explore the `mediaBox`:

```
>>> from PyPDF2 import PdfFileReader

>>> path = "C:/python-basics-exercises/" \
...        "ch13-interact-with-pdf-files/" \
...        "practice_files/half and half.pdf"

>>> input_pdf = PdfFileReader(path)
>>> page = input_pdf.getPage(0)
>>> page.mediaBox
RectangleObject([0, 0, 792, 612])
```

Notice that the `.mediaBox` property returns a `RectangleObject`. This is an object defined in the `PyPDF2` package and represents a rectangular area on the page.

You'll notice a list of four numbers in the output `RectangleObject([0, 0, 792, 612])`. The first two numbers in the list are the coordinates of the lower left corner of the rectangle. The third number is the width of the rectangle, and the fourth number represents the height of the rectangle.

A RectangleObject has four attributes that return the coordinates of the rectangle's corners. You can use these to find the coordinates of each corner of the mediaBox:

```
>>> page.mediaBox.lowerLeft
(0, 0)

>>> page.mediaBox.lowerRight
(792, 0)

>>> page.mediaBox.upperLeft
(0, 612)

>>> page.mediaBox.upperRight
(792, 612)
```

Each of these properties returns a tuple containing the coordinates of the specified corner. You can access individual coordinates with square brackets, just like you would any other Python tuple:

```
>>> page.mediaBox.upperRight[0]
792

>>> page.mediaBox.upperRight[1]
612
```

You can alter the coordinates of a mediaBox by assigning a new tuple to one of its properties:

```
>>> page.mediaBox.upperLeft = (0, 480)
>>> page.mediaBox.upperLeft
(0, 480)
```

When you change the .upperLeft coordinates, the .upperRight attribute adjust automatically so that a rectangular shape is preserved:

```
>>> page.mediaBox.upperRight
(792, 480)
```

Now that you have seen how the `mediaBox` works, you can use it to crop and split the pages in the `half and half.pdf` file. The following script handles the cropping, splitting, and saving the new PDF into a file called `The Little Mermaid.pdf`. The code is a little tricky, so take your time typing it out into a new script in IDLE. Each numbered section of the code is explained after the code block.

```python
import copy
import os

from PyPDF2 import PdfFileReader, PdfFileWriter

path = "C:/python-basics-exercises/" \
       "ch13-interact-with-pdf-files/practice_files"

# 1
input_file_path = os.path.join(path, "half and half.pdf")
input_pdf = PdfFileReader(input_file_path)
output_pdf = PdfFileWriter()

# 2
for page_num in range(0, input_pdf.getNumPages()):
    page_left = input_pdf.getPage(page_num)
    page_right = copy.copy(page_left)
    # Calculate the new coordinates for the upper-right
    # corner of the page_left and the upper-left
    # corner of page_right
    upper_right = page_left.mediaBox.upperRight
    new_coords = (upper_right[0]/2, upper_right[1])
    # Crop and add left-side page to the ouput file
    page_left.mediaBox.upperRight = new_coords
    output_pdf.addPage(page_left)
    # Crop and add right-side page to the output file
    page_right.mediaBox.upperLeft = new_coords
    output_pdf.addPage(page_right)
```

```python
# 3
output_file_path = os.path.join(path, "output/The Little Mermaid.pdf")
with open(output_file_path, "wb") as output_file:
    output_pdf.write(output_file)
```

Sections 1 and 2 of the above script handle opening and closing the input and output files for reading and writing. These operations should be familiar to you by now. The `for` loop in section 2 is new, so let's discuss what's going on there in more detail.

The `for` loop iterates over the numbers 0 to one less than the total number of pages in the `half and half.pdf` file. The first two lines in the `for` block get the next page in the PDF and make a copy of it. The `page_left` variable is assigned to the original page in the PDF, and the `page_right` page is assigned to a copy of `page_left` using the `copy()` function from the `copy` module. `page_right` is now a new independent `PageObject` that you can manipulate independently of `page_left`.

To crop the `page_left` and `page_right`, you'll have to do some math—but don't worry, nothing too crazy! You need to calculate the corners of each half-page so that you can crop out the side of the page you don't want.

You want to crop the page into two halves, right down the center of the page. If you think about each half of the page as rectangle, the upper-right-hand corner of the left side of the page is the same as the upper-left-hand corner of the right side of the page.

The x-coordinate (with the x-axis being the axis along the width of the page) is 1/2 of the width of the of the entire page. The width of the whole page is the x-coordinate of the upper-right-hand corner of `page_left.mediaBox`.

To calculate the coordinates of the new upper-right hand corner of `page_left`, you first get the original upper-right-hand corner using `page_left.mediaBox.upperRight`, and assign the tuple returned to the `upper_right` variable. Then a new tuple, with x-coordinate `upper_right[0]/2` and y-coordinate `upper_right[1]`, is assigned to the `new_coords` variable.

Next, the left side of the page is cropped by setting the `.upperRight` attribute of `page_left.mediaBox` to `new_coords` and then adding `page_left` to `output_pdf`. Then the right side of the page is cropped by setting `page_right.mediaBox.upperLeft` to `new_coords` and adding `page_right` to `output_pdf`.

Note

PDF files are a bit unusual in how they save page orientation. Depending on how the PDF was created, it might be the case that your axes are switched.

For instance, a standard "portrait" document that has been converted into a landscape PDF might have the x-axis represented vertically while the y-axis is horizontal. Likewise, the corners would all be rotated by 90 degrees. That is, the upper left corner would appear on the upper right or the lower left, depending on the file's rotation.

It's always a good idea to do some initial testing to make sure that you are using the correct corners and axes.

Merging Pages

Another common operation with PDFs is adding a header or a watermark to each page in an existing PDF document. You can do this with `PyPDF2` by merging two PDF pages into one.

For this example, you'll add a watermark to each page of the `The Emperor.pdf` file in the `practice_files` subfolder of the `python-basics-exercises/ch13-interact-with-pdf-files/` directory. The same folder contains a `top secret.pdf` file with a transparent background that you can use as the watermark.

To merge the `top secret.pdf` file to each page of `The Emperor.pdf`, use the `PageObject.mergePage()` method. The following script shows how to do this:

```python
import os

from PyPDF2 import PdfFileReader, PdfFileWriter

path = "C:/python-basics-exercises/" \
       "ch13-interact-with-pdf-files/practice_files"

# 1
input_file_path = os.path.join(path, "The Emperor.pdf")
input_pdf = PdfFileReader(input_file_path)
output_pdf = PdfFileWriter()

# 2
watermark_file_path = os.path.join(path, "top secret.pdf")
watermark_pdf = PdfFileReader(watermark_file_path)
watermark_page = watermark_pdf.getPage(0)

# 3
for page in input_pdf.pages:
    page.mergePage(watermark_page)
    output_pdf.addPage(page)

# 4
output_file_path = os.path.join(path, "output/New Suit.pdf")
with open(output_file_path, "wb") as output_file:
    output_pdf.write(output_file)
```

Sections 1 and 2 of the above script open the input and watermark PDF files for reading and create a `PdfFileWrite()` instance for writing to the output file. The `for` loop in section 3 loops through each page in the input file, merges each page with the first page of the watermark file, and then adds the merged page to the output file. Finally, in section 4, the new PDF is saved to a file called `New Suit.pdf`.

Password Protecting Your PDF Files

Sometimes PDF files are password protected. With the `PyPDF2` package, you can work with encrypted PDF files, as well as add password protection to existing PDFs.

To encrypt a PDF file, you use the `.encrypt()` method of a `PdfFileWriter()` instance. The `.encrypt()` method takes a single string `password` as an argument. For example, the following script creates a new password-protected copy of the `New Suit.pdf` file created in the previous script.

```python
import os

from PyPDF2 import PdfFileReader, PdfFileWriter

path = "C:/python-basics-exercises/" \
            "ch13-interact-with-pdf-files/practice_files"

input_file_path = os.path.join(path, "output/New Suit.pdf")
input_pdf = PdfFileReader(input_file_path)
output_pdf = PdfFileWriter()

for page in input_pdf.pages:
    output_pdf.addPage(page)

# Add password to the output PDF file
output_pdf.encrypt("SuperSecret")

output_file_path = os.path.join(path, "output/New Suit Encrypted.pdf")
with open(output_file_path, "wb") as output_file:
    output_pdf.write(output_file)
```

Now, when you open `New Suit Encrypted.pdf`, you are asked to enter a password. You must type “SuperSecret” into the prompt to open the file.

When you work with password-protected files programmatically, you need to decrypt them before you can access any of the contents. Try running the following script:

```python
import os

from PyPDF2 import PdfFileReader

path = "C:/python-basics-exercises/" \
       "ch13-interact-with-pdf-files/practice_files"

input_file_path = os.path.join(path, "output/New Suit Encrypted.pdf")
input_pdf = PdfFileReader(input_file_path)

for page in input_pdf.pages:
    text = page.extractText()
    print(text)
```

When you save and run the above program, you get a nasty looking traceback in IDLE's interactive window for a `PDFReadError` that tells you the file has not been decrypted:

```
Traceback (most recent call last):
  File "C:\realpython\read_top_secret.py", line 10, in <module>
    for page in input_pdf.pages:
  File "C:\...\PyPDF2\utils.py", line 159, in __getitem__
    len_self = len(self)
  File "C:\...\PyPDF2\utils.py", line 150, in __len__
    return self.lengthFunction()
  File "C:\...\PyPDF2\pdf.py", line 1150, in getNumPages
    raise utils.PdfReadError("File has not been decrypted")
PyPDF2.utils.PdfReadError: File has not been decrypted
```

> **Note**
>
> You may notice that the error above occurs on line 10 of the script, which is the first line of the `for` loop. That means you can successfully open a password-protected PDF without knowing the password, but you can not access any of its data.

To successfully read from the PDF, you must first decrypt the file with the `.decrypt()` method by passing to it the password as a string:

```python
import os

from PyPDF2 import PdfFileReader

path = "C:/python-basics-exercises/" \
       "ch13-interact-with-pdf-files/practice_files"

input_file_path = os.path.join(path, "output/New Suit Encrypted.pdf")
input_pdf = PdfFileReader(input_file_path)

# Decrypt the PDF file
input_pdf.decrypt("SuperSecret")

for page in input_pdf.pages:
    text = page.extractText()
    print(text)
```

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 opens the file named `Walrus.pdf` from the Chapter 13 practice files. Use the password “IamtheWalrus” to decrypt the file.

2. Rotate every page in this input file counter-clockwise by 90 degrees.

3. Split each page in half vertically, such that every column appears on its a separate page, and output the results as a new PDF file in the Output folder.

Leave feedback on this section »

13.3 Challenge: Add a Cover Sheet to a PDF File

In the `python-basics-exercises/ch13-interact-with-pdf-files/practice-files/` folder, there are two PDF files called `The Emperor.pdf` and `Emperor cover sheet.pdf`.

Write a script `cover_the_emperor.py` that appends `The Emperor.pdf` to the end of `Emperor cover sheet.pdf` and saves the resulting PDF to a file called `The Covered Emperor.pdf`.

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

Leave feedback on this section »

13.4 Create PDF Files

The PyPDF2 package is great for reading and modifying existing PDF files, but you can't use it to create a new PDF file. In this section, you will use the ReportLab toolkit, which has a free, open-source version, to generate PDF files from scratch.

This is not meant to be an exhaustive introduction to ReportLab. Instead, this section aims to give you a taste of what is possible. For more examples, checkout the ReportLab's code snippet page, where you can find tons of code that you can use in your own projects.

Install `reportlab`

To get started, you need to install ReportLab with `pip`:

```
$ pip3 install reportlab
```

You can verify the installation with `pip show`:

```
$ pip show reportlab
Name: reportlab
Version: 3.5.10
Summary: The Reportlab Toolkit
Home-page: http://www.reportlab.com/
Author: Andy Robinson, Robin Becker, the ReportLab team
        and the community
Author-email: reportlab-users@lists2.reportlab.com
License: BSD license (see license.txt for details),
         Copyright (c) 2000-2018, ReportLab Inc.
Location: c:usersdaveastuffvenvlibsite-packages
Requires: pillow
Required-by:
```

Notice that the latest version of `reportlab` at the time of writing is 3.5.10.

Create a PDF

The following script creates a new PDF file `hello.pdf` that contains the string `"Hello World"`:

```python
from reportlab.pdfgen import canvas

c = canvas.Canvas("hello.pdf")
c.drawString(100, 100, "Hello World")
c.save()
```

The script you used to make the `hello.pdf` file is pretty simple, but let's break it down a little bit to understand better what is going on.

The first thing to notice is that you do not import anything directly from `reportlab` but instead from the `reportlab.pdfgen` module. This module contains objects useful for generating PDFs. In particular, the `canvas` sub-module contains the `Canvas` class that is used to create a blank PDF page that you can write to.

The first line of code in the script assigns to the variable `c` a new `Canvas` object that will be created at the path `"hello.pdf"`. Since this is a relative path, the new file will be created in your current working directory. You could instead supply a different relative or absolute path to create the file wherever you would like.

The canvas's `.drawString()` method is used to write text to the canvas. The `.drawString()` method takes three arguments: the number of points from the left margin, the number of points from the bottom of the page, and the text to be written. A point is defined as 1/72 of an inch, so in this case, the string `"Hello World"` gets printed about 1.4 inches from the left and 1.4 inches from the bottom of the page.

The fact that the position of the text is measured from the bottom of the page and not the top may seem strange at first. It may help to think of the first two arguments of `.drawString` as x- and y-coordinates, with the origin (0, 0) located at the bottom-left corner of the page.

Finally, the canvas's `.save()` method is called to save the file. Once you have run the script, you can navigate to your current working directory in your file explorer and open your newly created PDF file! Pretty cool, huh?

In ReportLab, the default page size is A4, but you can change this. The following script does the same thing as the previous script, but saves the result to a letter-sized PDF:

```python
from reportlab.pdfgen import canvas
from reportlab.lib.pagesize import letter


c = canvas.Canvas("hello.pdf", pagesize=letter)
```

```
c.drawString(100, 100, "Hello World")
c.save()
```

Working with points to position text can be confusing. ReportLab provides an interface for using different units. For example, the following script writes `"Hello World"` precisely 2.3 inches from the left side of the page and 6 inches from the bottom:

```
from reportlab.pdfgen import canvas
from reportlab.lib.pagesizes import letter
from reportlab.lib.units import inch

xmargin = 3.2 * inch
ymargin = 6 * inch

c = canvas.Canvas("hello_again.pdf", pagesize=letter)
c.drawString(xmargin, ymargin, "Hello World")

c.save()
```

The other units available are: millimeters, `mm`; centimeters, `cm`; and `pica`.

Draw a Table

Now let's look at something a bit more complicated. PDFs are commonly used to create business reports, and business executives love tables of data. The following script draws a table with 5 columns and 8 rows in a new PDF file:

```
from reportlab.pdfgen import canvas
from reportlab.lib.pagesizes import letter
from reportlab.lib.units import inch
from reportlab.lib import colors
from reportlab.platypus import Table

xmargin = 3.2 * inch
```

```
ymargin = 6 * inch

c = canvas.Canvas("tps_report.pdf", pagesize=letter)

# 1
data = [['#1', '#2', '#3', '#4', '#5'],
        ['10', '11', '12', '13', '14'],
        ['20', '21', '22', '23', '24'],
        ['30', '31', '32', '33', '34'],
        ['20', '21', '22', '23', '24'],
        ['20', '21', '22', '23', '24'],
        ['20', '21', '22', '23', '24'],
        ['20', '21', '22', '23', '24']]

# 2
t = Table(data)

# 3
t.setStyle([('TEXTCOLOR', (0,0), (4,0), colors.red)])

# 4
t.wrapOn(c, xmargin, ymargin)
t.drawOn(c, xmargin, ymargin)

c.save()
```

The above program works as follows:

1. The data for the table is stored as a list of lists and assigned to the `data` variable.

2. Then a new `Table` object is create from `data` and is assigned to the variable `t`.

3. Next, the header of the table is colored read using the `.setStyle()` method. You'll notice that the argument to this method is a list containing a single tuple with four elements: the string `"TEXTCOLOR"`, which tells `setStyle()` to set the color of the text, two tuples `(0,0)`

and `(4, 0)` which define a range of cells to be colored, and finally `colors.red`, which defines the text color.

4. After setting the text color of the header, the `.wrapOn()` and `.drawOn()` methods are used to write the table to the canvas. Finally, the `.save()` method is used to save the PDF file.

You can learn more about tables in ReportLab in Chapter 7 of the ReportLab reference docs.

Leave feedback on this section »

13.5 Summary and Additional Resources

In this chapter, you learned how to interact with PDF files using the `PyPDF2` and `reportlab` libraries.

With `PyPDF2`, you can read PDF files using the `PdfFileReader` object. This object stores some of the PDF file's metadata as attributes like `.title` and `.author`. You saw how to extract text from a PDF and write that to a text file, how to extract portions of a PDF to another PDF, and how to split and rotate PDF pages.

You learned how to create PDF files from scratch using the `reportlab` library. You saw how to write strings to a blank PDF page and create tables. `reportlab` is a mature and commonly used library to create PDF reports—a useful skill in a variety of disciplines.

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-13

Additional Resources

More information about working with PDF files can be found in the `PyPDF2` and `reportlab` documentation:

- `PyPDF2` documentation
- `reportlab` documentation
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 14

Working With Databases

In Chapter 11 you learned how to store and retrieve data from files using Python. Another common way to store data is in a database.

A **database** is a structured system for storing data. It could be made up of several CSV files organized into directories, or something more elaborate.

Python comes with a light-weight SQL database called SQLite that is perfect for learning how to work with databases.

In this chapter, you will learn:

- How to create a SQLite database
- How to store and retrieve data from a SQLite database
- What packages are commonly used to work with other databases

Important

Some experience with SQL will be helpful when reading this chapter. If you want to learn more about SQL, check out the resources on Real Python.

Let's dig in!

Leave feedback on this section »

14.1 An Introduction to SQLite

There are numerous SQL databases, and some are better suited to particular purposes than others. One of the simplest, most lightweight SQL databases is SQLite, which runs directly on your machine and comes bundled with Python automatically.

In this section, you will learn how to use the `sqlite3` package to create a new database and store and retrieve data.

SQLite Basics

There are four basic steps to working with SQLite:

1. Import the `sqlite3` package
2. Connect to an existing database, or create a new database
3. Execute SQL statements on the database
4. Close the database connection

Let's get started by exploring these four steps in IDLEs interactive window. Open IDLE and type the following:

```
>>> import sqlite3
>>> connection = sqlite3.connect("test_database.db")
```

The `sqlite3.connect()` function is used to connect to, or create, a database. When you execute `.connect("test_database.db")`, Python searches for an existing database called `"test_database.db"`. If no database with that name is found, a new one is created in the current working directory. To create a database in a different directory, you must specify the full path in the argument to `.connect()`.

Note

If you want to create a one-time-use database while you're testing code or playing around with table structures, you can use the special name `":memory:"` to create the database in temporary memory, like so:

```
connection = sqlite3.connect(":memory:")
```

The `.connect()` function returns a `sqlite3.Connection` object, which you can verify with the `type()` function:

```
>>> type(connection)
<class 'sqlite3.Connection'>
```

`Connection` objects represent the connection between your program and the database. They have several attributes and methods that can be used to interact with the database. To store and retrieve data, you need a `Cursor` object, which can be obtained with the `Connection.cursor()` function:

```
>>> cursor = connection.cursor()
>>> type(cursor)
<class 'sqlite3.Cursor'>
```

The `sqlite3.Cursor` object is your gateway to interacting with the database. Using a `Cursor`, you can create database tables, execute SQL statements, and fetch query results.

Note

The term **cursor** in database jargon usually refers to an object that is used to fetch results from a database query one row at a time. Although `sqlite3.Cursor` objects are used for this operation, they also do much more than is typically expected from a cursor. This is one important distinction to keep in mind when you use other databases besides SQLite.

Let's use the SQLite `datetime` function to get the current local time:

```
>>> query = "SELECT datetime('now', 'localtime');"
>>> cursor.execute(query)
<sqlite3.Cursor object at 0x000001A27EB85E30>
```

To get the current time, you first build a SQL statement with the correct syntax. In this case, `"SELECT datetime('now', 'localtime');"` is the statement we need, and it is assigned to the `query` variable. This returns the current time using the local time zone settings on your machine. Then the query is executed using the `cursor.execute()` method.

Note that `.execute()` returns a `Cursor` object, but we didn't assign this to a new variable. That's because `.execute()` alters the state of `cursor` and also returns the `cursor` object itself. This might look kind of strange, but it allows you to chain multiple `Cursor` methods together on a single line.

You might be wondering where the time returned by the `datetime` function is. To get the query results, use the `cursor.fetchone()` method. `.fetchone()` returns a tuple containing the first row of results:

```
>>> cursor.fetchone()
('2018-11-20 23:07:21',)
```

Since `.fetchone()` returns a tuple, you need to unpack the tuple elements to get the string containing the date and time information. Here's how you can do this by chaining the `.execute()` and `.fetchone()` methods:

```
>>> time = cursor.execute(query).fetchone()[0]
>>> time
'2018-11-20 23:09:45'
```

Finally, to close the database connection, use the `connection.close()` method:

```
>>> connection.close()
```

Using `with` to Manage Your Database Connection

Recall from Chapter 11 that you can use a `with` statement with the `open()` function to open the file and then automatically close the file once the `with` block has executed. The same pattern applies to SQLite database connections and is the recommended way to open a database connection.

Here's the `datetime` example from above using a `with` statement to manage the database connection:

```
>>> with sqlite3.connect("test_database.db") as connection:
...     cursor = connection.cursor()
...     query = "SELECT datetime('now', 'localtime');"
...     time = cursor.execute(query).fetchone()[0]
...
>>> time
'2018-11-20 23:14:37'
```

In this example, the `connection` variable is assigned to the `Connection` object returned by `sqlite3.connect()` in the `with` statement. The code in the `with` block gets a new `Cursor` object using `connection.cursor()`, and then gets the current time with the `Cursor` object's `.execute()` and `.fetchone()` methods.

Managing your database connections in a `with` statement has many advantages. The resulting code is often cleaner and shorter than code written without a `with` statement. Moreover, any changes made to the database are saved automatically, as you'll see in the next example.

Working With Database Tables

You don't usually want to create a whole database just to get the current time. Databases are used to store and retrieve information. To store data in a database, you need to create a table and write some values to it.

Let's create a table called `People` with three columns: `FirstName`, LastName, and Age. The SQL query to create this table looks like this:

```
CREATE TABLE People(FirstName TEXT, LastName TEXT, Age INT);
```

Notice that FirstName and LastName have the word TEXT next to them, and Age is next to the word INT. This tells SQLite that values in the FirstName and LastName columns are text values, and values in the Age column are integers.

Once the table is created, you can populate it with some data using the INSERT INTO SQL command. The following query inserts the values Ron, Obvious, and 42 in the FirstName, LastName, and Age columns, respectively:

```
INSERT INTO People VALUES('Ron', 'Obvious', 42);
```

Note

Note that the string 'Ron' and 'Obvious' are delimited with single quotation marks. This makes them valid Python strings as well, but more importantly, only strings delimited with single quotes are valid SQLite strings.

When you write SQL queries as strings, you need to make sure that they are delimited with double quotation marks so that you can use single quotation marks inside of the Python strings to delimit SQLite strings.

SQLite is not the only SQL database that follows the single quote convention. Keep an eye out for this whenever you work with any SQL database.

Let's walk through how to execute these statements and save the changes to the database. First, we'll do it without using a with statement. Save and run the following script:

```
import sqlite3
```

```
connection = sqlite3.connect("test_database.db")
cursor = connection.cursor()
cursor.execute(
    """CREATE TABLE People(
           FirstName TEXT,
           LastName TEXT,
           Age INT
       );"""
)
cursor.execute(
    """INSERT INTO People VALUES(
           'Ron',
           'Obvious',
           42
       );"""
)
connection.commit()
connection.close()
```

First, you get a `Connection` object with `sqlite3.connect()` and assign it to the `connection` variable. A `Cursor` object is created with `connection.cursor()` and used to execute the two SQL statements for creating the `People` table and inserting some data.

The SQL statement in both `.execute()` methods have been written using triple quote strings so that we can format the SQL nicely. SQL ignores whitespace, so we can get away with this here and improve the readability of the Python code.

Finally, `connection.commit()` is used to save the data to the database. **Commit** is database jargon for saving data. If you do not run `connection.commit()`, no `People` table is created.

After the script runs, `test_database.db` has a `People` table with one row in it. You can verify this in the interactive window:

```
>>> connection = sqlite3.connect("test_database.db")
>>> cursor = connection.cursor()
>>> cursor.execute("SELECT * FROM People;")
<sqlite3.Cursor object at 0x000001F739DB6650>
>>> cursor.fetchone()
('Ron', 'Obvious', 42)
```

Next, let's look at the same script written using a `with` statement to manage the database connection. Before you can do anything, though, you need to delete the `People` table so that we can recreate it. Type the following into the interactive window to remove the `People` table from the database:

```
>>> cursor.execute("DROP TABLE People;")
<sqlite3.Cursor object at 0x000001F739DB6650>
>>> connection.commit()
>>> connection.close()
```

Now save and run the following script:

```
import sqlite3

with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.execute(
        """CREATE TABLE People(
            FirstName TEXT,
            LastName TEXT,
            Age INT
        );"""
    )
    cursor.execute(
        """INSERT INTO People VALUES(
            'Ron',
            'Obvious',
            42
        );"""
    )
```

Notice that not only is there no `connection.close()`, you also don't have to type `connection.commit()`. That's because any changes made to the database are automatically committed when the `with` block is done executing. This is another advantage to using a `with` statement to manage your database connection.

Executing Multiple SQL Statements

If you want to run more than one SQL statement at a time, you have a couple of options. One simple option is to use the `.executescript()` cursor method and give it a string that represents a full SQL script. Although semicolons separate lines of SQL code, it's common to pass a multiline string for readability. The following script does the same thing as the script you wrote at the beginning of this section:

```
import sqlite3

with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.executescript(
        """DROP TABLE IF EXISTS People;
           CREATE TABLE People(
                FirstName TEXT,
                LastName TEXT,
                Age INT
            );
            INSERT INTO People VALUES(
                'Ron',
                'Obvious',
                '42'
            );"""
    )
```

You can also execute many similar statements by using the `.executemany()` method and supplying a tuple of tuples, where each inner tuple supplies the information for a single command. For instance, if you have a lot of people's information to insert into our `People` table, you can save this information in the following tuple of tuples:

```
people_values = (
    ("Ron", "Obvious", 42),
    ("Luigi", "Vercotti", 43),
    ("Arthur", "Belling", 28)
)
```

You can then insert all of these people at once in a single line of code:

```
cursor.executemany("INSERT INTO People VALUES(?, ?, ?)", people_values)
```

Here, the question marks act as place-holders for the tuples in `people_values`. This is called a **parameterized statement**. You may notice some similarity to this and formatting strings with the `.format()` string method you learned about in Chapter 4.

Avoid Security Issues With Parametrized Statements

For security reasons, especially when you need to interact with a SQL table based on the user input, you should *always* use parameterized SQL statements. This is because the user could potentially supply a value that looks like SQL code and causes your SQL statement to behave in unexpected ways. This is called a SQL injection attack and, even if you aren't dealing with a malicious user, it can happen entirely by accident.

For instance, suppose you want to insert a person into the `People` table based on user-supplied information. You might initially try something like the following:

```
import sqlite3
```

```python
# Get person data from user
first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")
age = int(input("Enter your age: "))

# Execute insert statement for supplied person data
with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.execute(
        f"INSERT INTO People Values('{first_name}', '{last_name}', {age});"
    )
```

What if the user's name includes an apostrophe? Try adding Flannery O'Connor to the table, and you'll see that she breaks the code. This is because the apostrophe gets mixed up with the single quotes in the line, making it appear to the database that the SQL code ends earlier than expected.

In this case, the code only causes an error, which is bad enough. In some cases, though, bad input can corrupt an entire table. Many other hard-to-predict cases can break SQL tables, and even delete portions of your database. To avoid this, you should always use parameterized statements.

The following script does the same thing as the script above, but uses a parametrized statement to insert the user input into the database:

```python
import sqlite3

first_name = input("Enter your first name: ")
last_name = input("Enter your last name: ")
age = int(input("Enter your age: "))
data = (first_name, last_name, age)

with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.execute("INSERT INTO People VALUES(?, ?, ?);", data)
```

You can update the content of a row by using a parametrized SQL UPDATE statement. For instance, if you want to change the Age associated with someone already in our People table, you could use the following:

```
cursor.execute(
    "UPDATE People SET Age=? WHERE FirstName=? AND LastName=?;",
    (45, 'Luigi', 'Vercotti')
)
```

Retrieving Data

Of course, inserting and updating information in a database isn't all that helpful if you can't fetch that information from the database. To fetch data from a database, you can use the .fetchone() and .fetchall() cursor methods. These are similar to the .readline() and .readlines() methods for reading lines from a file. .fetchone() returns a single row from query results, while .fetchall() retrieves all of the results of a query at once.

The following script illustrates how to use .fetchall():

```
import sqlite3

values = (
    ("Ron", "Obvious", 42),
    ("Luigi", "Vercotti", 43),
    ("Arthur", "Belling", 28),
)

with sqlite3.connect("test_database.db") as connection:
    cursor = connection.cursor()
    cursor.execute("DROP TABLE IF EXISTS People")
    cursor.execute(
        """CREATE TABLE People(
               FirstName TEXT,
               LastName TEXT,
               Age INT
        );"""
    )
    cursor.executemany("INSERT INTO People VALUES(?, ?, ?);", values)

    # Select all first and last names from people over age 30
    cursor.execute(
        "SELECT FirstName, LastName FROM People WHERE Age > 30;"
    )
    for row in cursor.fetchall():
        print(row)
```

In the script above, you first drop the `People` table to destroy the changes made in the previous examples in this section. Then you create the `People` table and insert several values into it. Next, a `SELECT` statement is executed that returns the first and last names of all people over the age of 30.

Finally, `.fetchall()` returns the results of a query as a list of tuples, where each tuple contains the data from a single row in the query results. The output of the script looks like this:

```
('Ron', 'Obvious')
('Luigi', 'Vercotti')
```

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 new database with a table named `Roster` that has three fields: `Name`, `Species` and `IQ`. The `Name` and `Species` columns should be text fields, and the `IQ` column should be an integer field.

2. Populate your new table with the following values:

Name	Species	IQ
Jean-Baptiste Zorg	Human	122
Korben Dallas	Meat Popsicle	100

Name	Species	IQ
Ak’not	Mangalore	-5

3. Update the Species of Korben Dallas to be Human.
4. Display the names and IQs of everyone in the table classified as Human.

Leave feedback on this section »

14.2 Libraries for Working With Other SQL Databases

If you have a particular type of SQL database that you’d like to access through Python, most of the basic syntax is likely to be identical to what you just learned for SQLite. However, you’ll need to install an additional package to interact with your database since SQLite is the only built-in option.

There are many SQL variants and corresponding Python packages available. A few of the most commonly used and reliable open-source alternatives to SQLite are:

- pyodbc, which connects to ODBC (Open Database Connection) databases, such as Microsoft SQL Server
- psycopg2, which connects to the PostgreSQL database
- PyMySQL, which connects to MySQL databases

One difference between SQLite and other databases—besides the actual syntax of the SQL code, which changes slightly with most flavors of SQL—is that most databases require a username and password to connect. Check the documentation for the particular package you want to use to for the syntax for making a database connection.

The SQLAlchemy package is another popular option for working with databases. SQLAlchemy is an object-relational mapping, or ORM, that uses an object-oriented paradigm to build database queries. It can be configured to connect to a variety of databases. The object-oriented approach allows you to make queries without writing and raw SQL statements.

Leave feedback on this section »

14.3 Summary and Additional Resources

In this chapter, you learned how to interact with the SQLite database that comes with Python. SQLite is a small and light SQL database that can be used to store and retrieve data in your Python programs. To interact with SQLite in Python, you must import the `sqlite3` module.

To work with an SQLite database, you first need to connect to existing database, or create a new database, with the `sqlite3.connect()` function, which returns a `Connection` object. Then you can use the `Connection.cursor()` method to get a new `Cursor` object.

`Cursor` objects are used to execute SQL statements and retrieve query results. For example, `Cursor.execute()` and `Cursor.executescript()` are used to execute SQL queries. You can retrieve query results using the `Cursor.fetchone()` and `Cursor.fetchall()` methods.

Finally, you learned about several third-party packages that you can use to connect to other SQL databases, including `psycopg2`, which is used to connect to PostgreSQL databases, and `pyodbc` for Microsoft SQL Server. You also learned about the SQLAlchemy library, which provides a standard interface for connecting to a variety of SQL databases.

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-14

Additional Resources

Here are some more resources on working with databases:

- `pyodbc` Getting Started
- `psycopg` Documentation
- SQLAlchemy Tutorial
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 15

Interacting With the Web

The Internet hosts perhaps the greatest source of information—and misinformation—on the planet.

Many disciplines, such as data science, business intelligence, and investigative reporting, can benefit enormously from collecting and analyzing data from websites.

Web scraping is the process of collecting and parsing raw data from the web, and the Python community has come up with some pretty powerful web scraping tools.

In this chapter, you will learn how to:

- Parse website data using string methods and regular expressions
- Parse website data using an HTML parser
- Interact with forms and other website components

Important

Some experience with **HTML**, short for **H**yper**T**ext **M**arkup **L**anguage—will be helpful when reading this chapter. To learn more about HTML, check out the resources on Real Python.

Let's go!

Leave feedback on this section »

15.1 Scrape and Parse Text From Websites

Collecting data from websites using an automated process is known as web scraping. Some websites explicitly forbid users from scraping their data with automated tools like the ones you will create in this chapter. Websites do this for either of two possible reasons:

1. The site has a good reason to protect its data. For instance, Google Maps doesn't let you to request too many results too quickly.
2. Making many repeated requests to a website's server may use up bandwidth, slowing down the website for other users and potentially overloading the server such that the website stops responding entirely.

Important

You should always check a website's acceptable use policy before scraping its data to see if accessing the website by using automated tools is a violation of its terms of use. Legally, web scraping against the wishes of a website is very much a gray area.

Please be aware that the following techniques may be illegal when used on websites that prohibit web scraping.

Let's start by grabbing all of the HTML code from a single webpage. We'll take a straightforward page that's been set up just for practice:

```python
from urllib.request import urlopen

url = "http://olympus.realpython.org/profiles/aphrodite"
html_page = urlopen(url)
```

```
html_text = html_page.read().decode("utf-8")


print(html_text)
```

This displays the following result for us, which represents the full HTML of the page just as a web browser would see it:

```
<html>
<head>
<title>Profile: Aphrodite</title>
</head>
<body bgcolor="yellow">
<center>
<br><br>
<img src="/static/aphrodite.gif" />
<h2>Name: Aphrodite</h2>
<br><br>
Favorite animal: Dove
<br><br>
Favorite color: Red
<br><br>
Hometown: Mount Olympus
</center>
</body>
</html>
```

Calling `urlopen()` will cause the following error if Python cannot connect to the Internet:

```
URLError: <urlopen error [Errno 11001] getaddrinfo failed>
```

If you provide an invalid web address that can't be found, you will see the following error, which is equivalent to the "404" page that a browser would load:

```
HTTPError: HTTP Error 404: Not Found
```

Now we can scrape specific information from the webpage using **text**

parsing techniques. Text parsing involves looking through the full string of text and grabbing only the pieces that are relevant to us.

For instance, if we wanted to get the title of the webpage (in this case, "Profile: Aphrodite"), we could use the string `find()` method to search through the text of the HTML for the `<title>` tags and parse out the actual title using index numbers:

```python
from urllib.request import urlopen

url = "http://olympus.realpython.org/profiles/aphrodite"
page = urlopen(url)
html = page.read().decode('utf-8')

start_tag = "<title>"
end_tag = "</title>"
start_index = html.find(start_tag) + len(start_tag)
end_index = html.find(end_tag)

print(html[start_index:end_index])
```

Running this script displays the HTML code limited to only the text in the title:

```
Profile: Aphrodite
```

Of course, this worked for a simple example, but HTML in the real world can be much more complicated and far less predictable. For a small taste of the "expectations versus reality" of text parsing, visit /profiles/poseidon and view the HTML source code.

The HTML for the `/profiles/poseidon` page looks similar to the `/profiles/aphrodite` page, but there is a small difference. The opening `<title >` tag has an extra space in it before the closing `>` character. Re-run the script you used to parse the title from the `profiles/aphrodite` page, but this time set the `url` variable to `http://olympus.realpython.org/profiles/poseidon`.

Instead of just seeing the text `Profile: Poseidon`, you get the following:

```
<head>
<title >Profile: Poseidon
```

The modified script doesn't find the beginning of the `<title>` tag correctly because of that pesky space before the closing `>`. So, `html.find(end_tag)` returns -1 because the exact string `<title>` wasn't found anywhere. When `-1` is added to `len(start_tag)`, which is 7, the `start_index` variable gets the assigned the value `6`.

The 6th character of the `html_text` string is the beginning `<` of the `<head>` tag. This means that `html[start_index:end_index]` returns all of the HTML starting with `<head>` and ending just before `</title>`.

These sorts of problems can occur in countless unpredictable ways. A more reliable alternative than using `find()` is to use **regular expressions**. Regular expressions—or "regex" for short—are strings that can be used to determine whether or not text matches a particular pattern.

Note

Regular expressions are not particular to Python. They are a general programming concept that can be used with a wide variety of programming languages. Regular expressions use a language all of their own that is notoriously difficult to learn but incredibly useful once mastered.

Python provides built-in support for regular expressions through the `re` module. Just as Python uses the backslash character as an "escape character" for representing special characters that can't simply be typed into strings, regular expressions use many different "special" characters (called **meta-characters**) that are interpreted as ways to signify different types of patterns.

For instance, the asterisk character, `*`, stands for "zero or more" of whatever came just before the asterisk. In the following example, the `re.findall()` function is used to find any text within a string that matches a given regular expression. The first argument of `re.findall()` is the regular expression that you want to match, and the second argument is the string to test:

```
>>> import re

>>> re.findall("ab*c", "ac")
['ac']

>>> re.findall("ab*c", "abcd")
['abc']

>>> re.findall("ab*c", "acc")
['ac']

>>> re.findall("ab*c", "abcac")
['abc', 'ac']

>>> re.findall("ab*c", "abdc")
[]
```

Our regular expression, `ab*c`, matches any part of the string that begins with an “a,” ends with a “c,” and has zero or more of “b” in between the two. The `re.findall()` function returns a list of all matches. If no matches are found, an empty list is returned.

Note that the matching is case-sensitive. If you want to match this pattern regardless of upper-case or lower-case differences, you can pass a third argument with the value `re.IGNORECASE`, which is a specific variable stored in the `re` module:

```
>>> re.findall("ab*c", "ABC")
[]

>>> re.findall("ab*c", "ABC", re.IGNORECASE)
['ABC']
```

You can use a period `.` to stand for any single character in a regular expression. For instance, we could find all the strings that contain the letters “a” and “c” separated by a single character as follows:

```
>>> re.findall("a.c", "abc")
['abc']

>>> re.findall("a.c", "abbc")
[]

>>> re.findall("a.c", "ac")
[]

>>> re.findall("a.c", "acc")
['acc']
```

Putting the term `.*` inside of a regular expression stands for any character repeated any number of times. For instance, `"a.*c"` can be used to find every substring that starts with `"a"` and ends with `"c"`, regardless of which letter—or letters—are in-between:

```
>>> re.findall("a.*c", "abc")
['abc']

>>> re.findall("a.*c", "abbc")
['abbc']

>>> re.findall("a.*c", "ac")
['ac']

>>> re.findall("a.*c", "acc")
['acc']
```

Often, you use the `re.search()` function to search for a particular pattern inside a string. This function is somewhat more complicated than `re.findall()` because it returns an object called a `MatchObject` that stores different “groups” of data. This is because there might be matches inside of other matches, and `re.search()` returns every possible result.

The details of the MatchObject object are irrelevant here. For now, just know that calling the .group() method on a MatchObject will return the first and most inclusive result, which in most instances is just what you want. For instance:

```
>>> match_results = re.search("ab*c", "ABC", re.IGNORECASE)
>>> match_results.group()
'ABC'
```

There is one more function in the re module that is useful for parsing out text. The re.sub() function, which is short for “substitute,” allows you to replace text in a string that matches a regular expression with new text (sort of like the .replace() method). The arguments passed to re.sub() are the regular expression, followed by the replacement text, followed by the string. For example:

```
>>> string = "Everything is <replaced> if it's in <tags>."
>>> string = re.sub("<.*>", "ELEPHANTS", string)
>>> string
'Everything is ELEPHANTS.'
```

Perhaps that wasn’t quite what you expected to happen.

The re.sub() function uses the regular expression "<.*>" to find and replace everything in between the first < and last >, which is most of the string. This is because Python’s regular expressions are **greedy**, meaning that they try to find the longest possible match when characters like * are used.

Alternatively, you can use the non-greedy matching pattern *?, which works the same way as * except that it matches the shortest possible string of text:

```
>>> string = "Everything is <replaced> if it's in <tags>."
>>> string = re.sub("<.*?>", "ELEPHANTS", string)
>>> string
"Everything is ELEPHANTS if it's in ELEPHANTS."
```

Armed with all this knowledge, let's now try to parse out the title from http://olympus.realpython.org/profiles/dionysus, which includes this rather carelessly written line of HTML:

```
<TITLE >Profile: Dionysus</title  / >
```

The `.find()` method would have a difficult time dealing with the inconsistencies here, but with the clever use of regular expressions, you can handle this code easily:

```
import re
from urllib.request import urlopen

url = "http://olympus.realpython.org/profiles/dionysus"
page = urlopen(url)
html = page.read().decode("utf-8")

pattern = "<title.*?>.*?</title.*?>"
match_results = re.search(pattern, html, re.IGNORECASE)
title = match_results.group()
title = re.sub("<.*?>", "", title) # Remove HTML tags

print(title)
```

Let's take a closer look at the first regular expression in the `pattern` string by breaking it down into three parts—`<title.*?>`, `.*?`, and `</title.*?>`.

1. **`<title.*?>`**—This pattern matches the opening `<TITLE  >` tag in `html`. The `<title` part of the pattern matches with `<TITLE` because `re.search()` is called with `re.IGNORECASE`, and `.*?>` matches any text after `<TITLE` up to the first instance of `>`.

2. **`.*?`**—This pattern matches all text after the opening `<TITLE  >` non-greedily, stopping at the first match for `</title.*?>`.

3. **`</title.*?>`**—The only difference between this pattern and the first one is the `/` character, so this matches the closing `</title  / >` tag in `html`.

The second regular expression, the string `"<.*?>"` also uses the non-greedy `.*?` to match all the HTML tags in the `title` string. By replacing any matches with `""`, the `re.sub()` function removes all of the tags returns only the text.

Regular expressions are a powerful tool when used correctly. This introduction barely scratches the surface. You can learn more about regular expressions and how to use them in the Python Regular Expression HOWTO section of the Python documentation.

> **Note**
>
> Web scraping can be tedious. No two websites are organized the same way, and HTML is often messy. Moreover, websites change over time. Web scrapers that work today are not guaranteed to work next year—or next week, for that matter!

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 grabs the full HTML from the page http://olympus.realpython.org/profiles/dionysus
2. Use the string `.find()` method to display the text following "Name:" and "Favorite Color:" (not including any leading spaces or trailing HTML tags that might appear on the same line).
3. Repeat the previous exercise using regular expressions. The end of each pattern should be a "<" (the start of an HTML tag) or a newline character, and you should remove any extra spaces or newline characters from the resulting text using the string `.strip()` method.

Leave feedback on this section »

15.2 Use an HTML Parser to Scrape Websites

Although regular expressions are great for pattern matching in general, sometimes it’s easier to use an HTML parser that is explicitly designed for parsing out HTML pages. There are many Python tools written for this purpose, but the Beautiful Soup library is a good one to start with.

To install Beautiful Soup, you can run the following in your terminal:

```
$ pip3 install beautifulsoup4
```

Run `pip show` to see the details of the package you just installed:

```
$ pip3 show beautifulsoup4
Name: beautifulsoup4
Version: 4.6.3
Summary: Screen-scraping library
Home-page: http://www.crummy.com/software/BeautifulSoup/bs4/
Author: Leonard Richardson
Author-email: leonardr@segfault.org
License: MIT
Location: c:\realpython\venv\lib\site-packages
Requires:
Required-by:
```

In particular, notice that the latest version at the time of writing in 4.6.3.

Once you have Beautiful Soup installed, you can now import the `bs4` module and pass a string of HTML to `BeautifulSoup` to begin parsing:

```
from bs4 import BeautifulSoup
from urllib.request import urlopen

url = "http://olympus.realpython.org/profiles/dionysus"
```

```
page = urlopen(url)
html = page.read().decode("utf-8")
soup = BeautifulSoup(html, "html.parser")
```

This scripts does three things:

1. The URL `http://olympus.realpython.org/profiles/dionysus` is opened using the `urlopen()` function from the `urllib.request` module.
2. The HTML from the page is read as a string and assigned to the `html` variable.
3. A `BeautifulSoup` object is created and assigned to the `soup` variable.

The `BeautifulSoup` object assigned to `soup` is created with two arguments. The first argument is the HTML to be parsed and the second argument, the string `"html.parser"`, tells the object which parser to use behind the scenes. `"html.parser"` represents Python's built-in HTML parser.

Save and run the above script in IDLE. When it is finished running, you can use the `soup` variable in the interactive window to parse the content of `html` in various ways.

For example, `BeautifulSoup` objects have a `.get_text()` method that can be used to extract all of the text from the document and remove any HTML tags automatically.

Type the following code into IDLE's interactive window:

```
>>> print(soup.get_text())



Profile: Dionysus




```

```

Name: Dionysus

Hometown: Mount Olympus

Favorite animal: Leopard

Favorite Color: Wine
```

There are a lot of blank lines in this output. These are the result of newline characters in the HTML document's text. You can remove these with the string `.replace()` method, if you need to.

Often, you only need to get specific text from an HTML document. Using Beautiful Soup to extract the text first and then using the `.find()` string method is *sometimes* easier than working with regular expressions.

However, sometimes the HTML tags themselves are the elements that point out the data you want to retrieve. For instance, perhaps you want to retrieve the URLs for all the images on the page. These links are contained in the `src` attribute of `<img>` HTML tags. In this case, you can use the `find_all()` method to return a list of all instances of that particular tag:

```
>>> soup.find_all("img")
[<img src="/static/dionysus.jpg"/>, <img src="/static/grapes.png"/>]
```

This returns a list of all `<img>` tags in the HTML document. The objects in the list look like they might be strings representing the tags, but they are actually instances of the `Tag` object provided by Beautiful Soup. `Tag` objects provide a simple interface for working with the information they contain.

Let's explore this a little by first unpacking the `Tag` objects from the list:

```
>>> image1, image2 = soup.find_all("img")
```

Each `Tag` object has a `.name` property that returns a string containing the HTML tag type:

```
>>> image1.name
'img'
```

You can access the HTML attributes of the `Tag` object by putting their name in-between square brackets, just as if the attributes were keys in a dictionary.

For example, the `<img src="/static/dionysus.jpg"/>` tag has a single attribute `src` with the value `dionysus.jpg`. Likewise, and HTML tag such as the link `<a href="https://realpython.com" target="_blank">` has two attributes, `href` and `target`.

To get the source of the images in the Dionysus profile page, you access the `src` attribute using the dictionary notation mentioned above:

```
>>> image1["src"]
'/static/dionysus.jpg'

>>> image2["src"]
'/static/grapes.png'
```

Certain tags in HTML documents can be accessed by properties of the `Tag` object. For example, to get the `<title>` tag in a document, you can use the `.title` property:

```
>>> soup.title
<title>Profile: Dionysus</title>
```

If you look at the source of the Dionysus profile by navigating to the URL http://olympus.realpython.org/profiles/dionysus, right-clicking on the page, and selecting “View Page Source,” you will notice that the `<title>` tag as written in the document looks like this:

```
<title >Profile: Dionysus</title/>
```

Beautiful Soup automatically cleans up the tags for you by removing the extra space in the opening tag and the extraneous / in the closing tag.

You can also retrieve just the string in the title tag with the .string property of the Tag object:

```
>>> soup.title.string
'Profile: Dionysus'
```

One of the more useful features of Beautiful Soup is the ability to search for specific kinds of tags whose attributes match certain values. For example, if we want to find all of the <img> tags that have a src attribute equal to the value /static/dionysus.jpg, you can provide the following additional argument to the .find_all() method:

```
>>> soup.find_all("img", src="/static/dionysus.jpg")
[<img src="/static/dionysus.jpg"/>]
```

This example is somewhat arbitrary, and the usefulness of this technique may not be apparent from the example. If you spend some time browsing various websites and viewing their page source, you'll notice that many websites have extremely complicated HTML structure.

When scraping data from websites, you are often interested in particular parts of the page. By spending some time looking through the HTML document, you can identify tags with unique attributes that can be used to extract the data you need.

Then, instead of relying on complicated regular expressions or using .find() to search through the document, you can directly access the particular tag you are interested in and extract the data you need.

In some cases, you may find that Beautiful Soup does not offer the functionality you need. The lxml library is somewhat trickier to get started with but offers far more flexibility than Beautiful Soup for parsing HTML documents. You may want to check it out once you are comfortable with using Beautiful Soup.

Note

HTML parsers like Beautiful Soup can save you a lot of time and effort when it comes to locating specific data in webpages. However, sometimes HTML is so poorly written and disorganized that even a sophisticated parser like Beautiful Soup can't interpret the HTML tags properly.

In this case, you're often left to your own devices (namely, `.find()` and regex) to try to parse out the information you need.

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 grabs the full HTML from the page http://olympus.realpython.org/profiles
2. Parse out a list of all the links on the page using Beautiful Soup by looking for HTML tags with the name `a` and retrieving the value taken on by the `href` attribute of each tag.
3. Get the HTML from each of the pages in the list by adding the full path to the file name, and display the text (without HTML tags) on each page using Beautiful Soup's `.get_text()` method.

Leave feedback on this section »

15.3 Interact With HTML Forms

The `urllib` module you have been working with so far this chapter is well suited for requesting the contents of a webpage. Sometimes, though, you need to interact with a webpage to obtain the content you need. For example, you might need to submit a form or click on a button to display hidden content.

The Python standard library does not provide a built-in means for working with web pages interactively, but many third-party packages are available from PyPI. Among these, MechanicalSoup is a popular and relatively simple package to use.

In essence, Mechanical Soup installs what is known as a **headless browser**, which is a web browser with no graphical user interface. This browser is controlled programmatically via a Python script.

You can install Mechanical Soup with `pip3` in your terminal:

```
$ pip3 install MechanicalSoup
```

You can now view some details about the package with `pip3 show`:

```
$ pip3 show mechanicalsoup
Name: MechanicalSoup
Version: 0.10.0
Summary: A Python library for automating interaction with websites
Home-page: https://mechanicalsoup.readthedocs.io/
Author: UNKNOWN
Author-email: UNKNOWN
License: MIT
Location: c:\realpython\venv\lib\site-packages
Requires: requests, beautifulsoup4, six, lxml
Required-by:
```

In particular, notice that the latest version at the time of writing is 0.10.0.

Note

You may need to close and restart your IDLE session for MechanicalSoup to load and be recognized after it's been installed.

To get started, let's write a script that creates a new `Browser` instance with Mechanical Soup and retrieves a webpage:

```
import mechanicalsoup


browser = mechanicalsoup.Browser()
url = "http://olympus.realpython.org/login"
page = browser.get(url)
```

If you save and run the above script, you can then access the page variable in IDLE's interactive window, which will be useful for following along with the rest of this section.

The page variable now stores various information returned by the web server. For example, you can access the HTML of the webpage through the .soup property:

```
>>> page.soup
```

This will print out the following HTML:

```
<html>
<head>
<title>Log In</title>
</head>
<body bgcolor="yellow">
<center>
<br/><br/>
<h2>Please log in to access Mount Olympus:</h2>
<br/><br/>
<form action="/login" method="post" name="login">
Username: <input name="user" type="text"/><br/>
Password: <input name="pwd" type="password"/><br/><br/>
<input type="submit" value="Submit"/>
</form>
</center>
</body>
</html>
```

The /login page accessed by the above script has a <form> on it with <input> elements for a username and password.

You should open this page in a browser and look at it yourself before moving on. Try typing in a random username and password combination. If you guessed incorrectly, the message “Wrong username or password!” is displayed at the bottom of the page.

However, if you provide the correct login credentials (username “zeus” and password “ThunderDude”), you are redirected to the /profiles page.

In the next example, you will see how to use Mechanical Soup to fill out and submit this form using Python!

The important section of HTML code is the login form—that is, everything inside the <form> tags. The <form> on this page has the name attribute set to login. This form contains two <input> elements, one named user and the other named pwd. The third <input> element is the “Submit” button.

Now that you know the underlying structure of the login form, as well as the credentials needed to log in, let’s take a look at a script that fills the form out and submits it:

```python
import mechanicalsoup

# 1
browser = mechanicalsoup.Browser()
url = "http://olympus.realpython.org/login"
login_page = browser.get(url)
login_html = login_page.soup

# 2
form = login_html.select("form")[0]
form.select("input")[0]["value"] = "zeus"
form.select("input")[1]["value"] = "ThunderDude"
```

```
# 3
profiles_page = browser.submit(form, login_page.url)
```

After saving and running the script, you can confirm that you successfully logged in by typing the following into the interactive window:

```
>>> profiles_page.url
'http://olympus.realpython.org/profiles'
```

Let's break down the above example.

1. In the first part of the script, a `Browser` instance is created and used to request the http://olympus.realpython.org/login page. The HTML content of the page is assigned to the `login_html` variable using the `.soup` property.
2. The next section handles filling out the form. The first step is to retrieve the `<form>` element itself from the page's HTML. `login_html.select("form")` returns a list of all `<form>` elements on the page. Since the page has only one `<form>` element, you can access the form by retrieving the 0th element of the list. The next two lines select the username and password inputs and set their value to `"zeus"` and `"ThunderDude"`, respectively.
3. Finally, the form is submitted with the `browser.submit()` method. Notice that two arguments are passed to this method, the `form` object and the URL of the `login_page`, which is accessed via `login_page.url`.

In the interactive window, you confirmed that the submission successfully redirected to the `/profiles` page. If something had gone wrong, the value of `profiles_page.url` would still be .

Note

We are always being encouraged to use long passwords with many different types of characters in them, and now you know the main reason: automated scripts like the one we just designed can be used by hackers to "brute force" logins by rapidly trying to log in with many different usernames and passwords until they find a working combination.

Besides this being highly illegal, almost all websites these days lock you out and report your IP address if they see you making too many failed requests, so don't try it!

Now that we have the `profiles_page` variable set let's see how to programmatically obtain the URL for each link on the `/profiles` page.

To do this, you use the `.select()` method again, this time passing the string `"a"` to select all of the `<a>` anchor elements on the page:

```
>>> links = profiles_page.soup.select("a")
```

Now you can iterate of each link and print the `href` attribute:

```
>>> for link in links:
...     address = link["href"]
...     text = link.text
...     print(f"{text}: {address}")
...
Aphrodite: /profiles/aphrodite
Poseidon: /profiles/poseidon
Dionysus: /profiles/dionysus
```

The URLs contained in each `href` attribute are relative URLs, which aren't very helpful if you want to navigate to them later using Mechanical Soup. If you happen to know the full URL, you can assign the portion needed to construct a full URL. In this case, the base URL is just http://olympus.realpython.org. Then you can concatenate the base URL with the relative URLs found in the `src` attribute:

```
>>> base_url = "http://olympus.realpython.org"
>>> for link in links:
...     address = base_url + link["href"]
...     text = link.text
...     print(f"{text}: {address}")
...
Aphrodite: http://olympus.realpython.org/profiles/aphrodite
Poseidon: http://olympus.realpython.org/profiles/poseidon
Dionysus: http://olympus.realpython.org/profiles/dionysus
```

You can do a lot with just the `.get()`, `.select()`, and `.submit()` methods. That said, Mechanical Soup's is capable of much more. To learn more about Mechanical Soup, check out the official docs.

Review Exercises

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

1. Use Mechanical Soup to provide the correct username "zeus" and password "ThunderDude" to the login page submission form located at http://olympus.realpython.org/login.
2. Display the title of the current page to determine that you have been redirected to the `/profiles` page.
3. Use Mechanical Soup to return to login page by going "back" to the previous page.
4. Provide an incorrect username and password to the login form, then search the HTML of the returned webpage for the text "Wrong username or password!" to determine that the login process failed.

Leave feedback on this section »

15.4 Interact With Websites in Real-Time

Sometimes we want to be able to fetch real-time data from a website that offers continually updated information. In the dark days, before you learned Python programming, you would have been forced to sit in front of a browser, clicking the "Refresh" button to reload the page each time you want to check if updated content is available. Instead, you can easily automate this process using the `.get()` method of the Mechanical Soup `Browser` object.

Open up your browser of choice and navigate to http://olympus.realpython.org/dice. This page simulates a roll of a 6-sided die, updating the result each time you refresh the browser. As an example of working with real-time data, you will write a script that periodically scrapes this page for a new result. While this example is admittedly contrived, you will learn the basics of interacting with a website to retrieve periodically updated results.

The first thing you need to do is determine which element on the page contains the result of the die roll. Do this now by right-clicking anywhere on the page and clicking on "View page source." A little more than halfway down the HTML code, there is an `<h2>` tag that looks like this:

```
<h2 id="result">4</h2>
```

The text of the `<h2>` tag might be different for you, but this is the page element you need to scrape the result.

Note

For this example, you can easily check that there is only one element on the page with `id="result"`. Although the `id` attribute is supposed to be unique, in practice you should always check that the element you are interested in is uniquely identified. If not, you need to be creative with how you select that element in your code.

Let's start by writing a simple script that opens the /dice page, scrapes the result, and prints it to the console:

```python
import mechanicalsoup

browser = mechanicalsoup.Browser()
page = browser.get("http://olympus.realpython.org/dice")
tag = page.soup.select("#result")[0]
result = tag.text

print(f"The result of your dice roll is: {result}")
```

This example uses the BeautifulSoup `.select()` to find the element with `id=result`. The string `"#result"` passed to `.select()` uses the CSS ID selector `#` to indicate `result` is an `id` value.

To periodically get a new result, you'll need to create a loop that loads the page at each step of the loop. So everything below the line `browser = mechanicalsoup.Browser()` in the above script needs to go in the body of the loop.

For this example, let's get 4 rolls of the dice at 30-second intervals. To do that, the last line of your code needs to tell Python to pause running for 30 seconds. You can do this with the `sleep()` function from Python's `time` module. The `sleep()` function takes a single argument that represents the time to sleep in seconds. Here's a simple example to illustrate how the `sleep()` function works:

```python
import time

print("I'm about to wait for five seconds...")
time.sleep(5)
print("Done waiting!")
```

If you run the above example, you see that the `"Done waiting!"` message isn't displayed until 5 seconds have passed since the first `print()` function is executed.

For the die roll example, you'll need to pass the number 30 to `sleep()`. Here's the updated script:

```python
import time
import mechanicalsoup

browser = mechanicalsoup.Browser()

for i in range(4):
    page = browser.get("http://olympus.realpython.org/dice")
    tag = page.soup.select("#result")[0]
    result = tag.text
    print(f"The result of your dice roll is: {result}")
    time.sleep(30)
```

When you run the script, you will immediately see the first result printed to the console. After 30 seconds, the second result is displayed, then the third and finally the fourth. What happens after the fourth result is printed?

The script continues running for another 30 seconds before it finally stops!

Well, of *course* it does! That's what you told it to do! But it's kind of a waste of time. You can stop it from doing this by using an `if` statement to run the `time.sleep()` function only for the first three requests:

```python
import time
import mechanicalsoup

browser = mechanicalsoup.Browser()

for i in range(4):
    page = browser.get("http://olympus.realpython.org/dice")
    tag = page.soup.select("#result")[0]
    result = tag.text
    print(f"The result of your dice roll is: {result}")

    # Wait 30 seconds if this isn't the last request
    if i < 3:
        time.sleep(30)
```

Note

With techniques like this, you can scrape data from websites that periodically update their data. However, you should be aware that requesting a page multiple times in rapid succession can be seen as suspicious, or even malicious, use of a website. It's possible to crash a server with an excessive volume of request, so you can imagine that many websites are concerned about the volume of requests to their server!

Most websites publish a Terms of Use document. A link to this document can often be found in the website's footer. You should always read this document before attempting to scrape data from a website. If you can not find the Terms of Use, try to contact the website owner and ask them if they have any policies regarding request volume.

Failure to comply with the Terms of Use could result in your IP being blocked, so be careful and be respectful!

Review Exercises

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

1. Repeat the example in this section to scrape the die roll result, but additionally include the current time of the quote as obtained from the webpage. This time can be taken from part of a string inside a `<p>` tag that appears shortly after the result of the roll in the webpage's HTML.

Leave feedback on this section »

15.5 Summary and Additional Resources

Working with data from the Internet can be complicated. The structure of websites varies significantly from one site to the next, and even a single website can change often. Although it is possible to parse data from the web using tools in Python's standard library, there are many tools on PyPI that can help simplify the process.

In this chapter, you learned about Beautiful Soup and Mechanical Soup, two tools that help you write Python programs to automate website interactions. Beautiful Soup is used to parse HTML data collected from a website. Mechanical Soup is used to interact with website components, such as clicking on links and submitting forms. With tools like Beautiful Soup and Mechanical Soup, you can open up your programs to the world.

Web scraping techniques are used in many real-world disciplines. For example, investigative journalists rely on information collected from vast numbers of resources. Programmers have developed several tools for scraping, parsing, and processing data from websites to help journalists gather data and understand connections between people, places, and events.

Writing automated web scraping programs is fun. The Internet has no shortage of crazy content that can lead to all sorts of exciting projects. Just remember, not everyone wants you pulling data from their web servers. Always check a website's Terms of Use before you start scraping, and be respectful about how you time your web requests so that you don't flood a server with traffic.

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-15

Additional Resources

For more information on interacting with the web with Python, check out the following resources:

- Practical Introduction to Web Scraping in Python
- API Integration in Python
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 16

Scientific Computing and Graphing

Python is one of the leading programming languages in scientific computing and data science.

Python's popularity in this area is due, in part, to the wealth of third-party packages available on PyPI for manipulating and visualizing data.

From cleaning and manipulating large data sets, to visualizing data in plots and charts, Python's ecosystem has the tools you need to analyze and work with data.

In this chapter, you will learn how to:

- Work with arrays of data using `numpy`
- Create charts and plots with `matplotlib`

Let's dive in!

Leave feedback on this section »

16.1 Use NumPy for Matrix Manipulation

In this section, you will learn how to store and manipulate matrices of data using the NumPy package. Before getting to that, though, let's take a look at the problem NumPy solves.

If you have ever taken a course in linear algebra, you may recall that a matrix is a rectangular array of numbers. You can easily create a matrix in pure Python with a list of lists:

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

This seemingly works well. You can access individual elements of the matrix using their indices. For example, to access the second element of the first row of the matrix, you would type:

```
>>> matrix[0][1]
2
```

Now suppose you want to multiply every element of the matrix by 2. To do this, you need to write a nested `for` loop that loops of every element of each row of the matrix. You might use a nested `for` loop, like this:

```
>>> for row in matrix:
...     for i in range(len(row)):
...         row[i] = row[i] * 2
...
>>> matrix
[[2, 4, 6], [8, 10, 12], [14, 16, 18]]
```

While this may not seem so hard, the point is that in pure Python, you need to do a lot of work from scratch to implement even simple linear algebra tasks. Think about what you need to do if you want to multiply two matrices together!

NumPy provides nearly all of the functionality you might ever need out-of-the-box and is more efficient than pure Python. NumPy is written in the C language, and uses sophisticated algorithms for efficient computation, bringing you speed and flexibility.

Note

Even if you have no interest in using matrices for scientific computing, you still might find it helpful at some point to store data in a NumPy matrix because of the many useful methods and properties it provides.

For instance, perhaps you are designing a game and need an easy way to store, view and manipulate a grid of values with rows and columns. Rather than creating a list of lists or some other complicated structure, using a NumPy array is a simple way to store your two-dimensional data.

Install NumPy

Before you can work with NumPy, you'll need to install it using `pip`:

```
$ pip3 install numpy
```

Once NumPy has finished installing, you can see some details about the package by running `pip3 show`:

```
$ pip3 show numpy
Name: numpy
Version: 1.15.0
Summary: NumPy: array processing for numbers, strings,
         records, and objects.
Home-page: http://www.numpy.org
Author: Travis E. Oliphant et al.
Author-email: None
License: BSD
Location: c:\realpython\venv\lib\site-packages
Requires:
```

```
Required-by:
```

In particular, notice that the latest version at the time of writing is version 1.15.0.

Create a NumPy `array`

Now that you have NumPy installed let's create the same matrix from the first example in this section. Matrices in NumPy are instances of the `ndarray` object, which stands for "*n*-dimensional array."

> **Note**
>
> An *n*-dimensional array is an array with *n* dimensions. For example, a 1-dimensional array is a list. A 2-dimensional array is a matrix. Arrays can also have 3, 4, or more dimensions.
>
> In this section, we will focus on arrays with one or two dimensions.

To create an `ndarray` object, you can use the `array` alias. You initialize `array` objects with a list of lists, so to re-create the matrix from the first example as a NumPy array, you can do the following:

```
>>> import numpy as np
>>> matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> matrix
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

Notice how NumPy displays the matrix in a conveniently readable format? This is even true when printing the matrix with the `print()` function:

```
>>> print(matrix)
[[1 2 3]
 [4 5 6]
 [7 8 9]]
```

Accessing individual elements of the array works just like accessing elements in a list of lists:

```
>>> matrix[0][1]
2
```

You can optionally access elements with just a single set of square brackets by separating the indices with a comma:

```
>>> matrix[0, 1]
2
```

At this point, you might be wondering what the major difference is between a NumPy `array` and a Python `list`. For starters, NumPy arrays can only hold objects of the same type (for instance, all numbers) whereas Pythons lists can hold mixed types of objects. Check out what happens if you try to create an `array` with mixed types:

```
>>> np.array([[1, 2, 3], ["a", "b", "c"]])
array([['1', '2', '3'],
       ['a', 'b', 'c']], dtype='<U11')
```

NumPy doesn't raise an error. Instead, the types are converted to match one another. In this case, NumPy converts every element to a string. The `dtype='<U11'` that you see in the above output means that this array can only store Unicode strings whose length is at most 11 bytes.

On the one hand, the automatic conversions of data types can be helpful, but it can also be a potential source of frustration if the data types are not converted in the manner you expect. For this reason, it is generally a good idea to handle your type conversion *before* initializing an `array` object. That way you can be sure that the data type stored in your `array` matches your expectations.

Note

For more examples of how NumPy arrays differ from Python lists, checkout out this FAQ answer.

In NumPy, each dimension in an array is called an `axis`. Both of the previous matrices you have seen have two axes. Arrays with two axes are also called **two-dimensional arrays**. Here is an example of a three-dimensional array:

```
>>> matrix = np.array([
...     [[1, 2, 3], [4, 5, 6]],
...     [[7, 8, 9], [10, 11, 12]],
...     [[13, 14, 15], [16, 17, 18]]
... ])
```

To access an element of the above `array`, you need to supply three indices:

```
>>> matrix[0][1][2]
6

>>> matrix[0, 1, 2]
6
```

If you think creating the above three-dimensional array looks confusing, you'll see a better way to create higher dimensional arrays later in this section.

Array Operations

Once you have an `array` object created, you can start to unleash the power of NumPy and perform some operations.

Recall from the first example in this section how you had to write a nested `for` loop to multiply each element in a matrix by the number 2. In NumPy, this operation is as simple as multiplying your `array` object by 2:

```
>>> matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> 2 * matrix
array([[ 2,  4,  6],
       [ 8, 10, 12],
       [14, 16, 18]])
```

You can just as easily perform element-wise arithmetic on multi-dimensional arrays as well:

```
>>> second_matrix = np.array([[5, 4, 3], [7, 6, 5], [9, 8, 7]])
>>> second_matrix - matrix
array([[ 4,  2,  0],
       [ 3,  1, -1],
       [ 2,  0, -2]])
```

All of the basic arithmetic operators (`+`, `-`, `*`, `/`) operate on arrays element for element. For example, multiplying two arrays with the `*` operator does *not* compute the product of two matrices. Consider the following example:

```
>>> matrix = np.array([[1, 1, 1], [1, 1, 1], [1, 1, 1]])
>>> matrix * matrix
array([[1, 1, 1],
       [1, 1, 1],
       [1, 1, 1]])
```

To calculate an actual matrix product, you can use the `@` operator:

```
>>> matrix @ matrix
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
```

Note

The @ operator was introduced in Python 3.5, so if you are using an older version of Python you must multiply matrices differently. NumPy provides a function called matmul() for multiplying two matrices:

```
>>> np.matmul(matrix, matrix)
array([[3, 3, 3],
       [3, 3, 3],
       [3, 3, 3]])
```

The @ operator actually relies on the np.matmul() function internally, so there is no real difference between the two methods.

Other common array operations are listed here:

```
>>> matrix = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

>>> # Get a tuple of axis length
>>> matrix.shape
(3, 3)

>>> # Get an array of the diagonal entries
>>> matrix.diagonal()
array([1, 5, 9])

>>> # Get a 1-dimensional array of all entries
>>> matrix.flatten()
array([1, 2, 3, 4, 5, 6, 7, 8, 9])

>>> # Get the transpose of an array
>>> matrix.transpose()
array([[1, 4, 7],
       [2, 5, 8],
       [3, 6, 9]])
```

```
>>> # Calculate the minimum entry
>>> matrix.min()
1

>>> # Calculate the maximum entry
>>> matrix.max()
9

>>> # Calculate the average value of all entries
>>> matrix.mean()
5.0

>>> # Calculate the sum of all entries
>>> matrix.sum()
45
```

Stacking and Shaping Arrays

Two arrays can be stacked vertically using `np.vstack()` or horizontally using `np.hstack()` if their axis sizes match:

```
>>> A = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
>>> B = np.array([[10, 11, 12], [13, 14, 15], [16, 17, 18]])

>>> np.vstack([A, B])
array([[ 1,  2,  3],
       [ 4,  5,  6],
       [ 7,  8,  9],
       [10, 11, 12],
       [13, 14, 15],
       [16, 17, 18]])

>>> np.hstack([A, B])
array([[ 1,  2,  3, 10, 11, 12],
       [ 4,  5,  6, 13, 14, 15],
       [ 7,  8,  9, 16, 17, 18]])
```

You can also reshape arrays with the `np.reshape()` function:

```
>>> A.reshape(9, 1)
array([[1],
       [2],
       [3],
       [4],
       [5],
       [6],
       [7],
       [8],
       [9]])
```

Of course, the total size of the reshaped array must match the original array's size. For instance, you can't execute `matrix.reshape(2, 5)`:

```
>>> A.reshape(2, 5)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ValueError: cannot reshape array of size 9 into shape (2, 5)
```

In this case, you are trying to shape an array with 9 entries into an array with 2 columns and 5 rows. This requires a total of 10 entries.

The `np.reshape()` function can be particularly helpful in combination with `np.arange()`, which is NumPy's equivalent to Python's `range()` function. The main difference is that `np.arange()` returns an `array` object:

```
>>> matrix = np.arange(1, 10)
>>> matrix
array([1, 2, 3, 4, 5, 6, 7, 8, 9])
```

Just like with `range()`, `np.arange()` starts with the first argument and ends just before the second argument. So, `np.arange(1, 10)` returns an `array` containing the numbers 1 through 9.

Together, `np.arange()` and `np.reshape()` provide a useful way to create a matrix:

```
>>> matrix = matrix.reshape(3, 3)
>>> matrix
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

You can even do this in a single line by chaining the calls to `np.arange()` and `np.reshape()` together:

```
>>> np.arange(1, 10).reshape(3, 3)
array([[1, 2, 3],
       [4, 5, 6],
       [7, 8, 9]])
```

This technique for creating matrices is particularly useful for creating higher-dimensional arrays. Here's how to create a three-dimensional array using `np.array()` and `np.reshape()`:

```
>>> np.arange(1, 13).reshape(3, 2, 2)
array([[[ 1,  2],
        [ 3,  4]],

       [[ 5,  6],
        [ 7,  8]],

       [[ 9, 10],
        [11, 12]]])
```

Of course, not every multi-dimensional array can be built from a sequential list of numbers. In that case, it is often easier to create and flat, one-dimensional list of entries and then `np.reshape()` the array into the desired shape:

```
>>> arr = np.array([1, 3, 5, 7, 9, 11, 13, 15, 17, 19, 21, 23])
>>> arr.reshape(3, 2, 2)
```

```
array([[[ 1,  3],
        [ 5,  7]],


       [[ 9, 11],
        [13, 15]],


       [[17, 19],
        [21, 23]]])
```

In the list passed to `np.array()` in the above example, the difference between any pair of consecutive numbers is 2. You can simplify the creation of these kinds of arrays by passing an optional third argument the `np.arange()` called the **stride**:

```
>>> np.arange(1, 24, 2)
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23])
```

With that in mind, you can re-write the previous example even more simply:

```
>>> np.arange(1, 24, 2).reshape(3, 2, 2)
array([[[ 1,  3],
        [ 5,  7]],


       [[ 9, 11],
        [13, 15]],


       [[17, 19],
        [21, 23]]])
```

Sometimes you need to work with matrices of random data. With NumPy, creating random matrices is easy. The following creates a random 3 x 3 matrix:

```
>>> np.random.random([3, 3])
array([[0.27721176, 0.66206403, 0.20722988],
       [0.15722803, 0.06286636, 0.47220672],
       [0.55657541, 0.27040345, 0.24558674]])
```

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 3 × 3 NumPy array named `first_matrix` that includes the numbers 3 through 11 by using `np.arange()` and `np.reshape()`.
2. Display the minimum, maximum and mean of all entries in `first_matrix`.
3. Square every entry in `first_matrix` using the `**` operator, and save the results in an array named `second_matrix`.
4. Use `np.vstack()` to stack `first_matrix` on top of `second_matrix` and save the results in an array named `third_matrix`.
5. Use the `@` operator to calculate the matrix product of `third_matrix` by `first_matrix`.
6. Reshape `third_matrix` into an array of dimensions 3 × 3 × 2.

Leave feedback on this section »

16.2 Use `matplotlib` for Plotting Graphs

In the previous section, you learned how to work with arrays of data using the `NumPy` package. While `NumPy` makes working with and manipulating data simple, it does not provide a means for human consumption of data. For that, you need to visualize your data.

Data visualization is a broad topic, complete with its own theory and a host of tools for displaying and interacting with visualizations. In this section, you will get an introduction to the matplotlib package, which is one of the more popular packages for quickly creating two-dimensional figures. Initially released in 2003, `matplotlib` is one of the oldest Python plotting libraries available. It remains popular and is still being actively developed to this day.

If you have ever created graphs in `MATLAB`, you will find that `matplotlib` in many ways directly emulates this experience. The similarities between `MATLAB` and `matplotlib` are intentional. The `MATLAB` plotting interface was a direct inspiration for `matplotlib`. Even if you haven't used `MATLAB`, you will likely find creating plots with `matplotlib` to be simple and straightforward.

Let's dive in!

Install `matplotlib`

You can install `matplotlib` from your terminal with `pip3`:

```
pip3 install matplotlib
```

You can then view some details about the package with `pip3 show`:

```
$ pip3 show matplotlib
Name: matplotlib
Version: 2.2.3
Summary: Python plotting package
Home-page: http://matplotlib.org
Author: John D. Hunter, Michael Droettboom
Author-email: matplotlib-users@python.org
License: BSD
Location: c:\realpython\venv\lib\site-packages
Requires: python-dateutil, pytz, kiwisolver, numpy,
          cycler, six, pyparsing
Required-by:
```

In particular, note that the latest version at the time of writing is version 2.2.3.

Basic Plotting With `pyplot`

The `matplotlib` package provides two distinct means of creating plots. The first, and simplest, method is through the `pyplot` interface. This is the interface that MATLAB users will find the most familiar.

The second method for plotting in `matplotlib` is through what is known as the object oriented API. The object-oriented approach offers more control over your plots than is available through the `pyplot` interface. However, the concepts are generally more abstract.

In this section, you'll learn how to get up and running with the `pyplot` interface. You'll be pumping out some great looking plots in no time!

Note

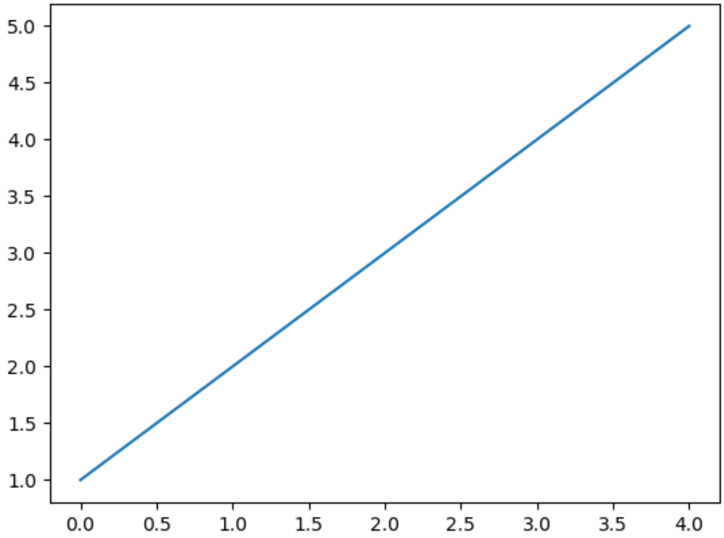
The developers of `matplotlib` suggest you try to use the object-oriented API instead of the `pyplot` interface. In practice, if the `pyplot` interface offers you everything you need, then don't be ashamed to stick with it!

That said, if you are interested in learning more about the object-oriented approach, check out Real Python's Python Plotting With Matplotlib (Guide).

Let's start by creating a simple plot. Open IDLE and run the following script:

```
from matplotlib import pyplot as plt


plt.plot([1, 2, 3, 4, 5])
plt.show()
```

A new window appears displaying the following plot:



In this simple script, you created a plot with just a single line of code. The line `plt.plot([1, 2, 3, 4, 5])` creates a plot with a line through the points (0, 1), (1, 2), (2, 3), (3, 4), and (4, 5). The list `[1, 2, 3, 4, 5]` that you passed to the `plt.plot()` function represents the y-values of the points in the plot. Since you didn't specify any x-values, `matplotlib` automatically uses the indices of the list elements which, since Python starts counting at 0, are 0, 1, 2, 3 and 4.

The `plt.plot()` function creates a plot, but it does not display anything. The `plot.show()` function must be called to display the plot.

Note

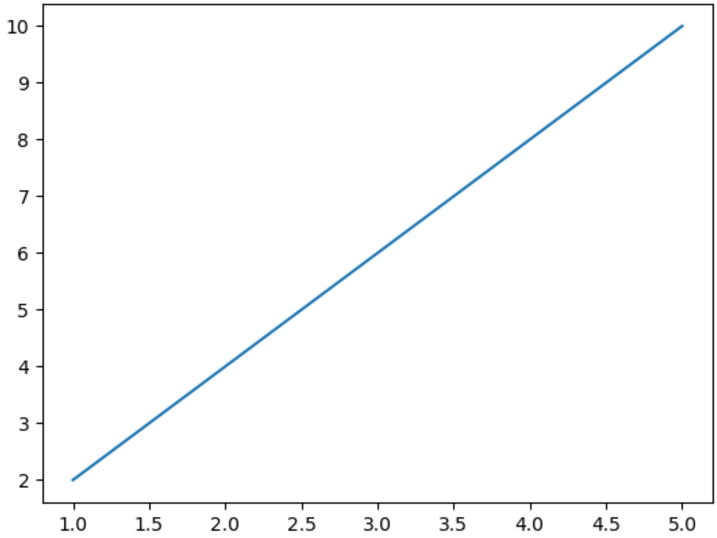
If you are working in Windows, you should have no problem recreating the above plot from IDLE's interactive window. However, some operating systems have trouble displaying plots with `plot.show()` when called from the interactive window. We recommend working through each example in a new script.

If `plt.show()` works from the interactive window on your machine and you decide to follow along that way, be aware that once the figure is displayed in the new window, control isn't returned to the interactive window until you close the figure's window. That is, you won't see a new `>>>` prompt until the figure's window has been closed.

You can specify the x-values for the points in your plot by passing two lists to the `plt.plot()` function. When two arguments are provided to `plt.plot()`, the first list specifies the x-values and the second list specifies the y-values:

```python
from matplotlib import pyplot as plt


xs = [1, 2, 3, 4, 5]
ys = [2, 4, 6, 8, 10]


plt.plot(xs, ys)
plt.show()
```

Running the above script produces the following plot:

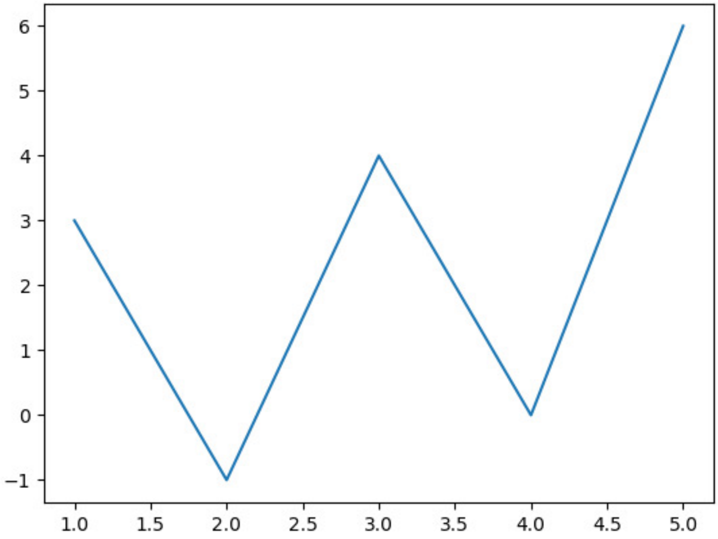


At first glance, this figure may look exactly like the first. However, the labels on the axes now reflect the new x- and y-coordinates of the points.

You can use `plot()` to plot more than lines. In the graphs above, the points being plotted just happen to all fall on the same line. By default, when plotting points with `.plot()`, each pair of consecutive points being plotted is connected with a line segment.

The following plot displays some data that doesn't fall on a line:

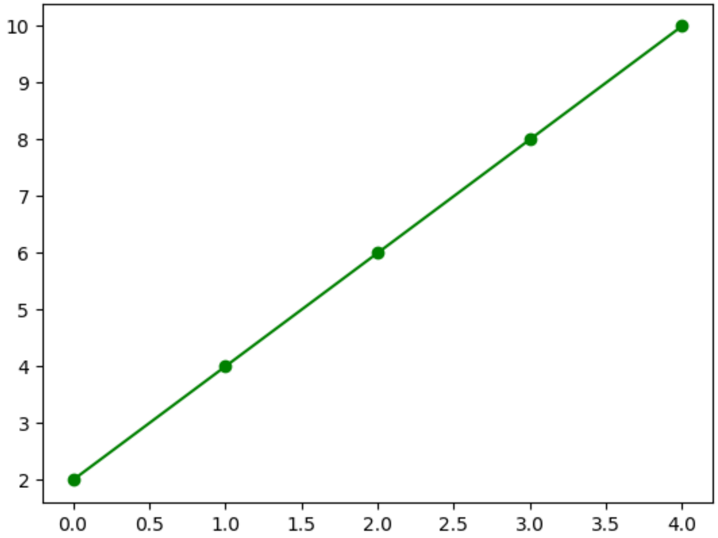
```python
from matplotlib import pyplot as plt

xs = [1, 2, 3, 4, 5]
ys = [3, -1, 4, 0, 6]

plt.plot(xs, ys)
plt.show()
```



There is an optional "formatting" argument that can be inserted into `plot()` after specifying the points to be plotted. This argument specifies the color and style of lines or points to draw.

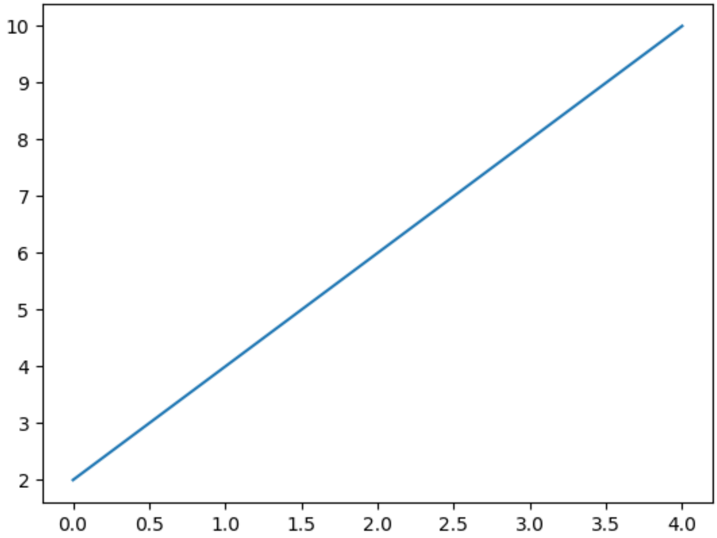
Unfortunately, the standard is borrowed from MATLAB and (compared to most Python) the formatting is not very intuitive to read or remember. The default value is "solid blue line," which would be represented by the format string `b-`. If we wanted to plot green circular dots connected by solid lines instead, we would use the format string `g-o` like so:

```
from matplotlib import pyplot as plt


plt.plot([2, 4, 6, 8, 10], "g-o")
plt.show()
```



Note

For reference, the full list of possible formatting combinations can be found here.

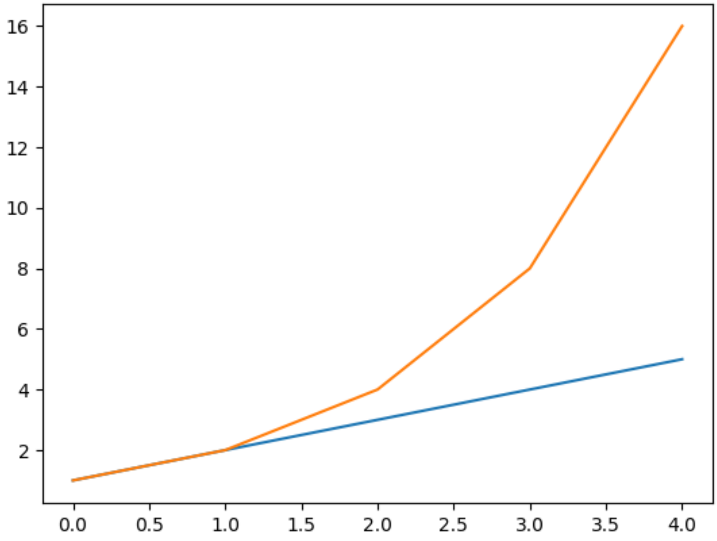


Plot Multiple Graphs in the Same Window

If you need to plot multiple graphs in the same window, you can do so a few different ways.

You can pass multiple pairs of x- and y-value lists:

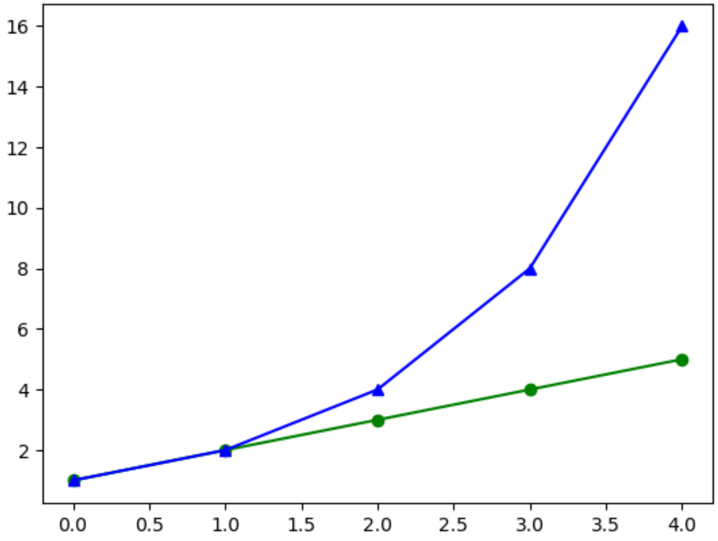
```python
from matplotlib import pyplot as plt

xs = [0, 1, 2, 3, 4]
y1 = [1, 2, 3, 4, 5]
y2 = [1, 2, 4, 8, 16]

plt.plot(xs, y1, xs, y2)
plt.show()
```



Notice that each graph is displayed in a different color. This built-in functionality of the `plot()` function is convenient for making easy-to-read plots very quickly.

If you want to control the style of each graph, you can pass the formatting strings to the `plot()` in addition to the x- and y-values:

```
from matplotlib import pyplot as plt


xs = [0, 1, 2, 3, 4]
y1 = [1, 2, 3, 4, 5]
y2 = [1, 2, 4, 8, 16]


plt.plot(xs, y1, "g-o", xs, y2, "b-^")
plt.show()
```



Passing multiple sets of points to `plot()` may work well when you only have a couple of graphs to display, but if you need to show many, it might make more sense to display each one with its own `plot()` function.

For example, the following script displays the same plot as the previous example:

```python
from matplotlib import pyplot as plt


plt.plot([1, 2, 3, 4, 5], "g-o")
plt.plot([1, 2, 4, 8, 16], "b-^")
plt.show()
```

Plot Data From NumPy Arrays

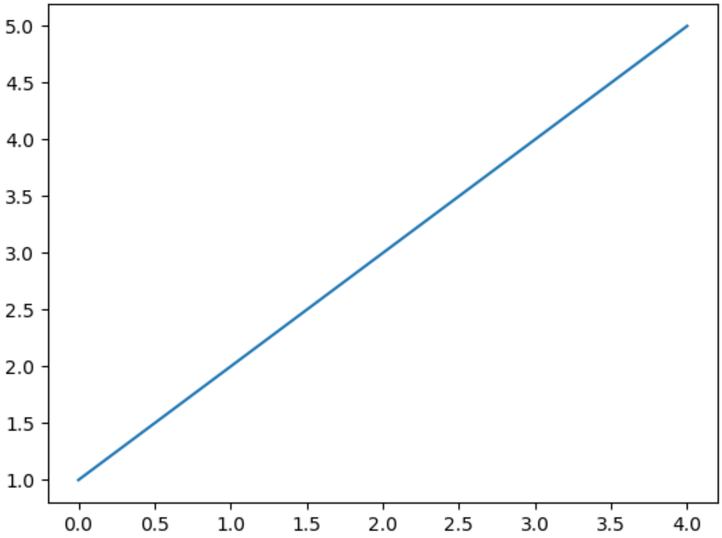
Up to this point, you have been storing your data points in pure Python lists. In the real world, you will most likely be using something like a NumPy `array` to store your data. Fortunately, `matplotlib` plays nicely with `array` objects.

Note

If you do not currently have NumPy installed, you need to install it with `pip`. For more information, please refer to the previous section in this chapter.

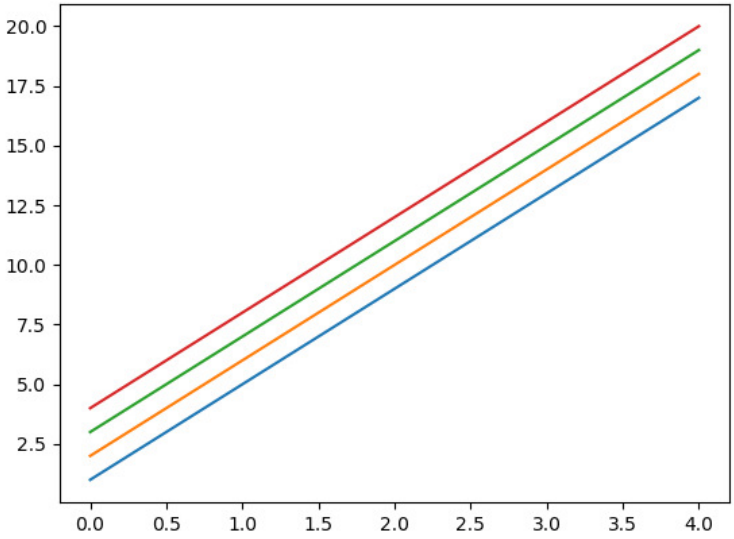
For example, instead of a `list`, you can use NumPy's `arange()` function to define your data points and then pass the resulting `array` object to the `plot()` function:

```
from matplotlib import pyplot as plt
import numpy as np

array = np.arange(1, 6)

plt.plot(array)
plt.show()
```



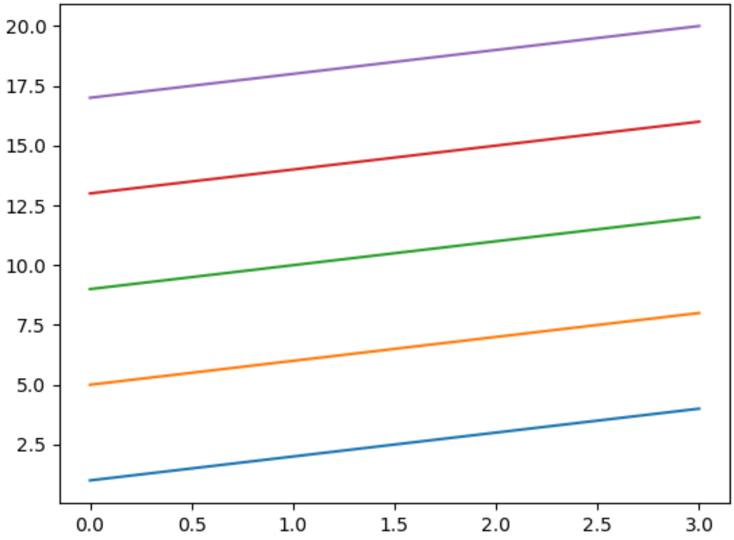
Passing a two-dimensional `array` plots each *column* of the array as the y-values for a graph. For example, the following script plots four lines:

```
from matplotlib import pyplot as plt
import numpy as np

data = np.arange(1, 21).reshape(5, 4)

# data now contains the following array:
# array([[ 1,  2,  3,  4],
#        [ 5,  6,  7,  8],
#        [ 9, 10, 11, 12],
#        [13, 14, 15, 16],
#        [17, 18, 19, 20]])

plt.plot(data)
plt.show()
```



If instead you want to plot the rows of the matrix, you need to plot the transpose of the array. The following script plots the five rows of the same array from the previous example:

```
from matplotlib import pyplot as plt
import numpy as np

data = np.arange(1, 21).reshape(5, 4)

plt.plot(data.transpose())
plt.show()
```

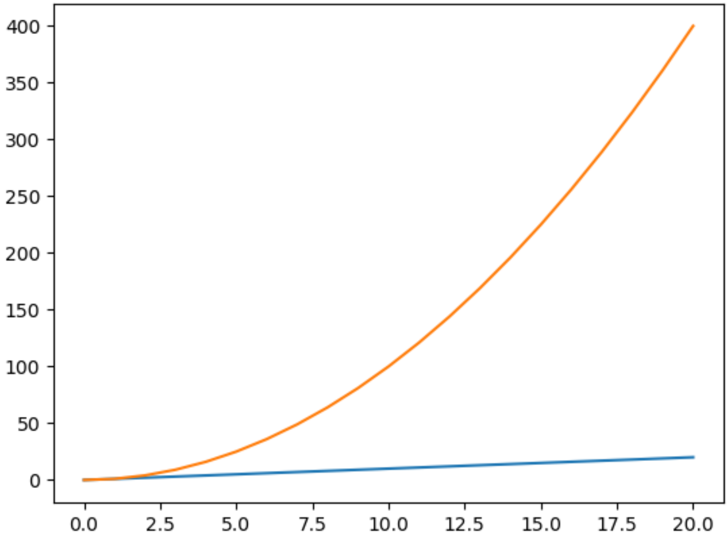


Format Your Plots to Perfection

So far, the plots you have seen don't provide any information about what the plot represents. In this section, you will learn how to change the format and layout of your plots to make them easier to understand.

Let's start by plotting the amount of Python learned in the first 20 days of reading Real Python versus another website:

```python
from matplotlib import pyplot as plt
import numpy as np

days = np.arange(0, 21)
other_site = np.arange(0, 21)
real_python = other_site ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.show()
```



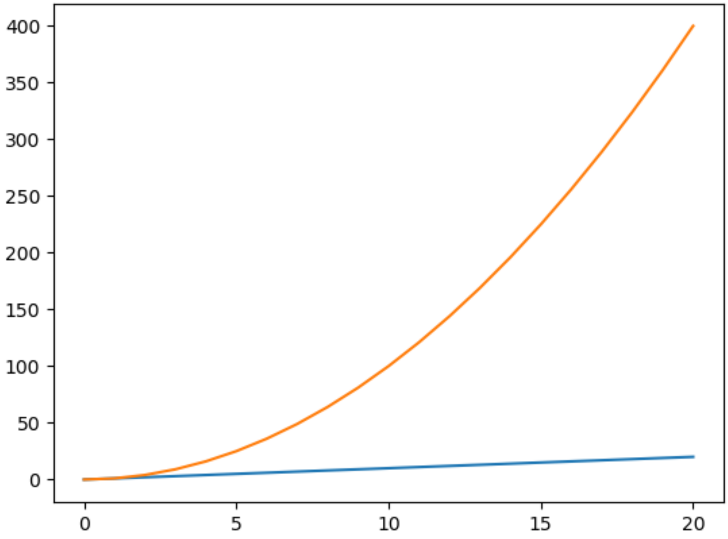
As you can see, the gains from reading Real Python are exponential! However, if you showed this graph to someone else, they may not understand what's going on.

First of all, the x-axis is a little weird. It is supposed to represent days but is displaying half days instead. It would also be helpful to know what each line and axis represents. A title describing the plot wouldn't hurt, either.

Let's start with adjusting the x-axis. You can use the `plt.xticks()` function to specify where the ticks should be located by passing a `list` of locations. If we pass the list `[0, 5, 10, 15, 20]`, the ticks should mark days 0, 5, 10, 15 and 20:

```python
from matplotlib import pyplot as plt
import numpy as np
```

```
days = np.arange(0, 21)
other_site = np.arange(0, 21)
real_python = other_site ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.xticks([0, 5, 10, 15, 20])
plt.show()
```

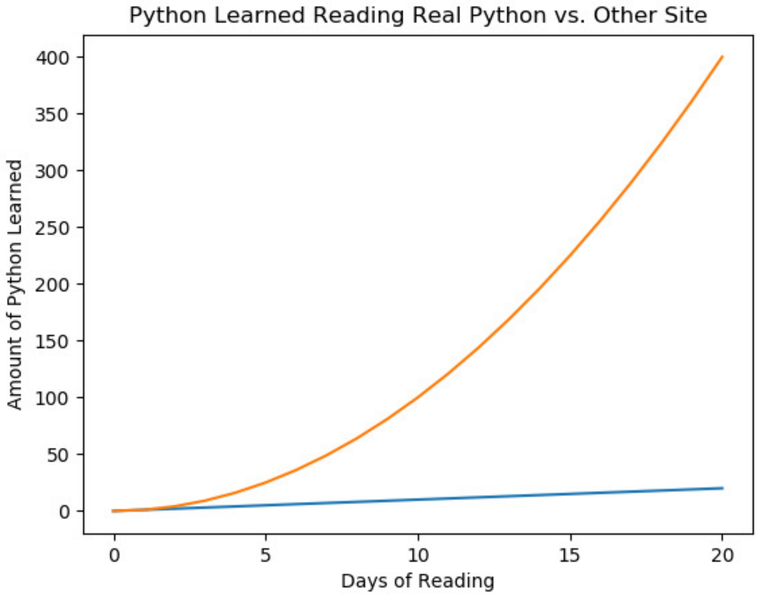


Nice! That's a little easier to read, but it still isn't clear what each axis represents.

You can use the `plt.xlabel()` and `plt.ylabel()` to label the x- and y-axes, respectively. Just provide a string as an argument, and `matplotlib` displays the label on the corresponding axis.

While we're labeling things, let's go ahead and give the plot a title with the `plt.title()` function:

```python
from matplotlib import pyplot as plt
import numpy as np

days = np.arange(0, 21)
other_site = np.arange(0, 21)
real_python = other_site ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.xticks([0, 5, 10, 15, 20])
plt.xlabel("Days of Reading")
plt.ylabel("Amount of Python Learned")
plt.title("Python Learned Reading Real Python vs Other Site")
plt.show()
```

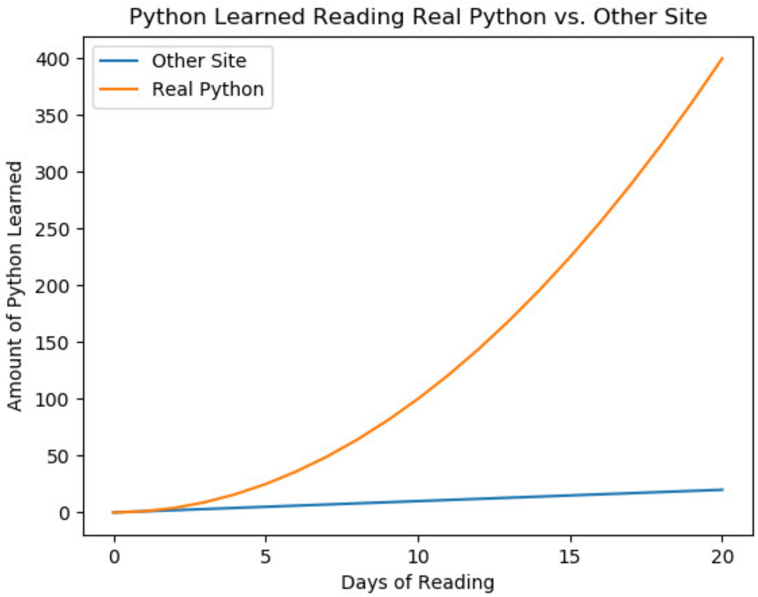


Now we're starting to get somewhere!

There's only one problem. It's not clear which graph represents Real Python and which one represents the other website.

To clarify which graph is which, you can add a legend with the `plt.legend()` function. The primary argument of the `legend()` function is a list of strings identifying each graph in the plot. These strings must be ordered in the same order the graphs were added to the plot:

```
from matplotlib import pyplot as plt
import numpy as np

days = np.arange(0, 21)
other_site = np.arange(0, 21)
real_python = other_site ** 2

plt.plot(days, other_site)
plt.plot(days, real_python)
plt.xticks([0, 5, 10, 15, 20])
plt.xlabel("Days of Reading")
plt.ylabel("Amount of Python Learned")
plt.title("Python Learned Reading Real Python vs Other Site")
plt.legend(["Other Site", "Real Python"])
plt.show()
```



Note

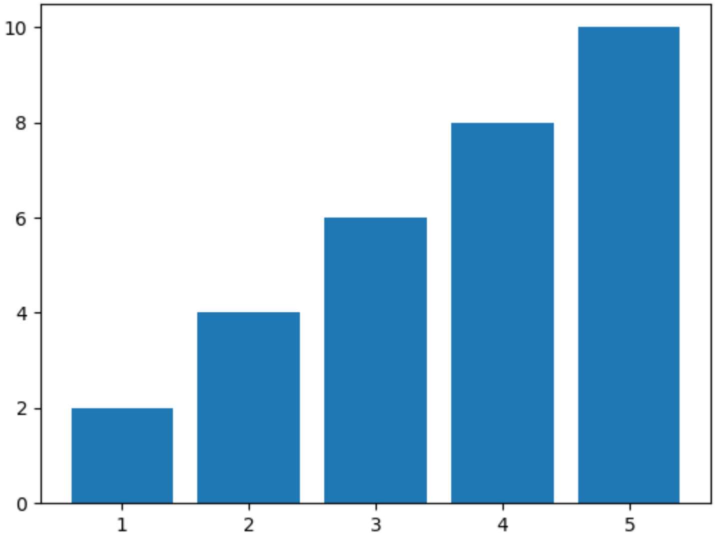
There are many ways to customize legends. For more information, check out the Legend Guide in the `matplotlib` documentation.

Other Types of Plots

Aside from line charts, which up until now you have seen exclusively, `matplotlib` provides simple methods for creating other kinds of charts.

One frequently used type of plot in basic data visualization is the bar chart. You can easily create bar charts using the `plt.bar()` function. You must provide at least two arguments to `bar()`. The first is a list of x-values for the center point for each bar, and the second is the value for the top of each bar:

```python
from matplotlib import pyplot as plt

xs = [1, 2, 3, 4, 5]
tops = [2, 4, 6, 8, 10]

plt.bar(xs, tops)
plt.show()
```



Just like the `plot()` function, you can use a NumPy `array` instead of a list. The following script produces a plot identical to the previous one:

```python
from matplotlib import pyplot as plt
import numpy as np

xs = np.arange(1, 6)
tops = np.arange(2, 12, 2)
```

```
plt.bar(xs, tops)
plt.show()
```

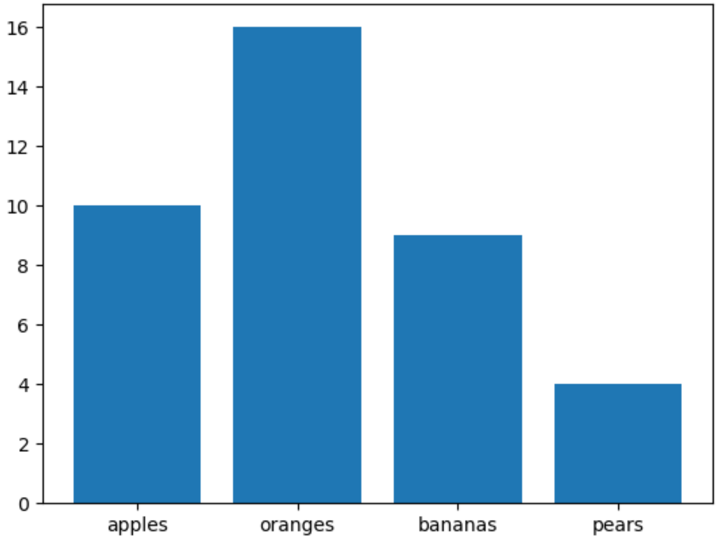
The `bar()` function is more flexible than it lets on. For example, the first argument doesn't need to be a list of numbers. It could be a list of strings representing categories of data.

Suppose you wanted to plot a bar chart representing the data contained in the following dictionary:

```
fruits = {
    "apples": 10,
    "oranges": 16,
    "bananas": 9,
    "pears": 4,
}
```

You can get a list of the names of the fruits using `fruits.keys()`, and the corresponding values using `fruits.values()`. Check out what happens when you pass these to the `bar()` function

```
from matplotlib import pyplot as plt

fruits = {
    "apples": 10,
    "oranges": 16,
    "bananas": 9,
    "pears": 4,
}

plt.bar(fruits.keys(), fruits.values())
plt.show()
```



The names of the fruits are conveniently used as the tick labels along the x-axis.

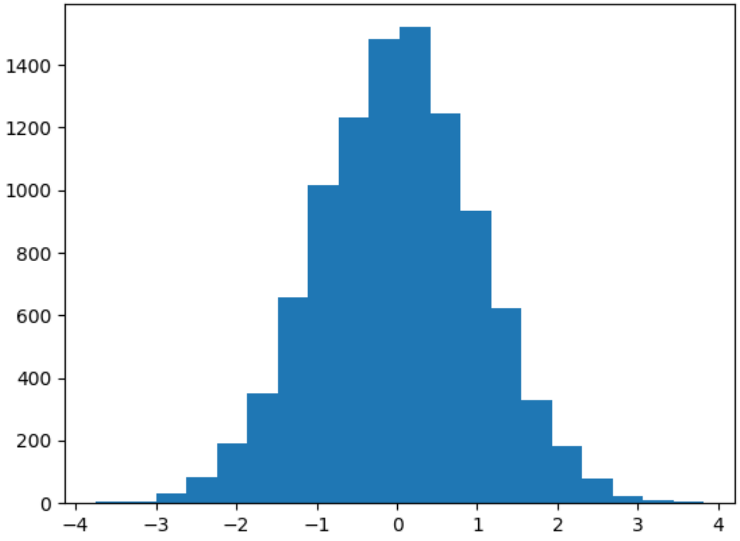
Note

Using a list of strings as x-values works for the `plot()` function as well, although it often makes less sense to do so.

Another commonly used type of graph is the histogram, which shows how data is distributed. You can make simple histograms easily with the `plt.hist()` function. You must supply `hist()` with a list (or array) of values and a number of bins to use.

For instance, we can create a histogram of 10,000 normally distributed random numbers binned across 20 possible bars with the following, which uses NumPy's `random.randn()` function to generate an array of normally distributed random numbers:

```python
from matplotlib import pyplot as plt
from numpy import random

plt.hist(random.randn(10000), 20)
plt.show()
```



Note

For a detailed discussion of creating histograms with Python, check out Python Histogram Plotting: NumPy, Matplotlib, Pandas & Seaborn on Real Python.

Save Figures as Images

You may have noticed that the window displaying your plots has a toolbar at the bottom. You can use this toolbar to save your plot as an image file.

More often than not, you probably don't want to have to sit at your computer and click on the save button for each plot you want to export. Fortunately, `matplotlib` makes it easy to save your plots programmatically.

To save your plot, use the `plt.savefig()` function. Pass the path to where you would like to save your plot as a string. The example below saves a simple bar chart as `bar.png` to the current working directory. If you would like to save to somewhere else, you must provide an absolute path.

```python
from matplotlib import pyplot as plt
import numpy as np

xs = np.arange(1, 6)
tops = np.arange(2, 12, 2)

plt.bar(xs, tops)
plt.savefig("bar.png")
```

> **Note**
>
> If you want to both save a figure and display it on the screen, make sure that you save it first before displaying it!
>
> The `show()` function pauses execution of your code and closing the display window destroys the graph, so trying to save the figure after calling `show()` results in an empty file.

Work With Plots Interactively

When you are initially tweaking the layout and formatting of a particular graph, it can be helpful to change parts of the graph without having to re-run an entire script just to see the results.

One of the easiest ways to do this is with a Jupyter Notebook, which creates an interactive Python interpreter session that runs in your browser.

Jupyter notebooks have become a staple for interacting with and exploring data, and work great with both NumPy and `matplotlib`.

For an interactive tutorial on how to use Jupyter Notebooks, check out Jupyter's IPython In Depth tutorial.

Review Exercises

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

1. Recreate as many of the graphs shown in this section as you can by writing your own scripts without referring to the provided code.
2. It is a well-documented fact that the number of pirates in the world is correlated with a rise in global temperatures. Write a script `pirates.py` that visually examines this relationship:
   - Read in the file `pirates.csv` from the Chapter 16 practice files folder.
   - Create a line graph of the average world temperature in degrees Celsius as a function of the number of pirates in the world—that is, graph Pirates along the x-axis and Temperature along the y-axis.
   - Add a graph title and label your graph's axes.
   - Save the resulting graph out as a PNG image file.
   - Bonus: Label each point on the graph with the appropriate Year. You should do this programmatically by looping through the actual data points rather than specifying the individual position of each annotation.

Leave feedback on this section »

16.3 Summary and Additional Resources

In this chapter, you learned about two packages commonly used in the Python scientific computing stack.

In the first section, "Use NumPy for Matrix Manipulation," you learned about the NumPy package. NumPy is used for working with multi-dimensional arrays of data. It introduces the `ndarray` object, which is commonly created using the `array` alias.

A NumPy `array` is a homogenous data type, meaning it can only store a single type of data. For example, a NumPy `array` can contain all integers, or all floats, but cannot contain both integers *and* floats. You also saw some useful functions and methods for manipulating NumPy `array` objects. Finally, you were introduced to the NumPy `arange()` function, which works a lot like Python's very own `range()` function, except that returns a one-dimensions NumPy `array` object.

In the second section, "Use `matplotlib` for Plotting Graphs," you learned how to use the `matplotlib` package to create simple plots using the `pyplot` interface. You built line charts, bar charts and histograms from pure Python lists and NumPy arrays using the `plot()`, `bar()` and `hist()` functions. You learned how to style and layout your plots by adding plot and axis titles, tick markers and legends.

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-16

Additional Resources

With the knowledge you gained in this chapter you should be able to work with basic data arrays and produce some simple plots. If your goal is to use Python for data science or scientific computing, you now have some foundational knowledge. To further your study, you may want to check out the following resources:

- Real Python Data Science Tutorials
- Recommended resources on realpython.com

Leave feedback on this section »

Chapter 17

Graphical User Interfaces

Throughout this book, you have been creating **command-line applications**, which are programs that are started from and produce output in a terminal window.

Command-line apps are fine for making tools that you or other developers might use, but the vast majority of software users never want to open a terminal!

Graphical **U**ser **I**nterfaces, called **GUIs** for short, provide windows with components like buttons and text fields that simplify how a user interacts with a program.

In this chapter, you will learn how to:

- Add a simple GUI to a command line application with `EasyGUI`
- Create full-featured GUI applications with `tkinter`

Let's get started!

Leave feedback on this section »

17.1 Add GUI Elements With EasyGUI

The EasyGUI library provides a simple interface for adding GUI elements to your scripts. To get started, you'll need to install the package with `pip3`:

```
$ pip3 install easygui
```

Once installed, you can check out some details of the package with `pip show`:

```
$ pip3 show easygui
Name: easygui
Version: 0.98.1
Summary: EasyGUI is a module for very simple, very easy GUI
         programming in Python.  EasyGUI is different from other
         GUI generators in that EasyGUI is NOT event-driven.
         Instead, all GUI interactions are invoked by simple
         function calls.
Home-page: https://github.com/robertlugg/easygui
Author: easygui developers and Stephen Ferg
Author-email: robert.lugg@gmail.com
License: BSD
Location: c:\realpython\venv\lib\site-packages
Requires:
Required-by:
```

In particular, note that version 0.98.1 is the latest version at the time of writing.

Get Started With EasyGUI

EasyGUI is different from other GUI modules because it doesn't rely on events. Most GUI applications are **event-driven**, meaning that the flow of the program depends on actions taken by the user. This is part of what makes GUI programming so complicated. Any object that might change in response to the user has to "listen" for different "events" to occur.

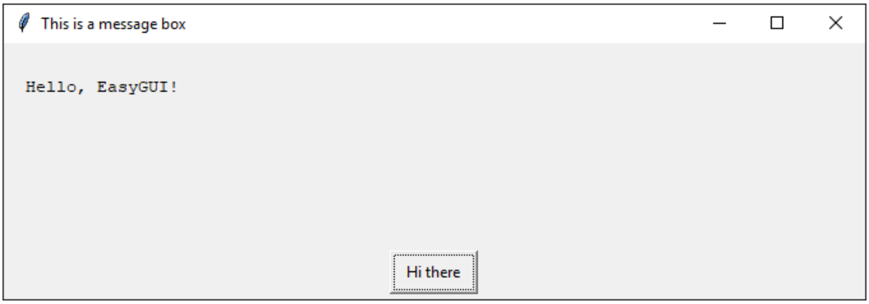
By contrast, EasyGUI is executed sequentially, just like all of the Python scripts you've written up to this point. Program flow with EasyGUI typically works like this:

1. At some point in the code, a visual element is displayed on the user's screen.
2. Execution of the code pauses until the user provides input with the visual element.
3. The user's input is returned as an object and execution of the code is resumed.

To get a feel for how EasyGUI works, open a new interactive window in IDLE and execute the following lines of code:

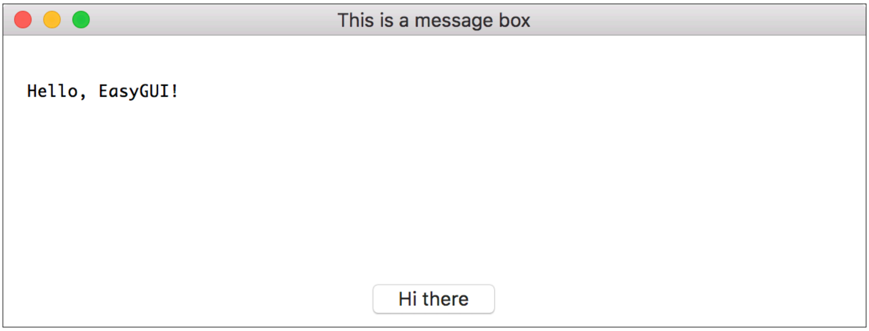
```
>>> import easygui as gui
>>> gui.msgbox("Hello, EasyGUI!", "This is a message box", "Hi there")
```

You should see something like the following window appear on your screen:



Hello, EasyGUI!

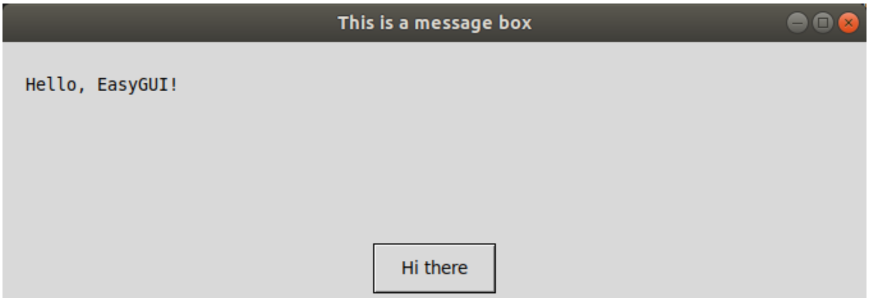
On macOS, it looks more like this:



This is a message box

Hello, EasyGUI!

And in Ubuntu:



This is a message box

Hello, EasyGUI!

If nothing appears, see the note below. For the rest of this section, Windows screenshots will be shown.

There are a few things to notice about the window that is displayed. First, the string `"Hello, EasyGUI!"` that you passed as the first argument to the `msgbox()` function is displayed as the message in the message box. The second argument, `"This is a message box"` is displayed as the title of the message box. Finally, the third argument `"Hi there"` is displayed as text on the button at the bottom of the window.

Press the "Hi there" button. The window will close, and you should see `'Hi there'` displayed as output in the interactive window. When the window closes, the text value of the button is returned as a Python string. If the window is closed without pressing the button, then the value `None` is returned.

If no third argument is provided to `msgbox()`, the button will have a value of "OK" by default.

Note

If you work with EasyGUI or Tkinter in an IDLE session, you might run into problems with freezing or "stuck" windows, and other issues. This happens because of disagreements between the new windows you create and the IDLE window itself.

If you think this might be happening, you can always try running your code or script by running Python from the command prompt (Windows) or Terminal (macOS/Linux). You can start an interactive Python session from the terminal by executing the `python3` command.

The `msgbox()` function is useful for displaying a message to the user in a familiar format, but it doesn't provide the user with many options for interacting with your program. EasyGUI provides many different kinds of "boxes" for displaying information and offering choices to your users.

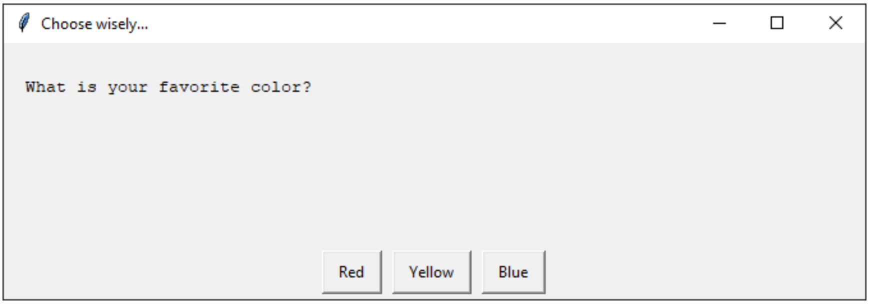
Work With Other GUI Elements

One simple way to provide some choices to your user is with `gui.buttonbox()`. This function displays a window that looks a lot like the one displayed by `msgbox()`, but it can provide a number of different buttons for the user to push.

The `buttonbox()` function takes three arguments: a string to display as a message in the window, a string to display as the window title, and a tuple (or list) of strings to display as buttons:

```
>>> choices = ("Red", "Yellow", "Blue")
>>> gui.buttonbox(
...     "What is your favorite color?", "Choose wisely...", choices)
```

Executing the above code produces the following window:



When the user presses a button, one of the three options is returned as a string (or `None` is returned if the user closes the window without making a selection). If needed, you can assign this value to a variable and use it later on in your code.

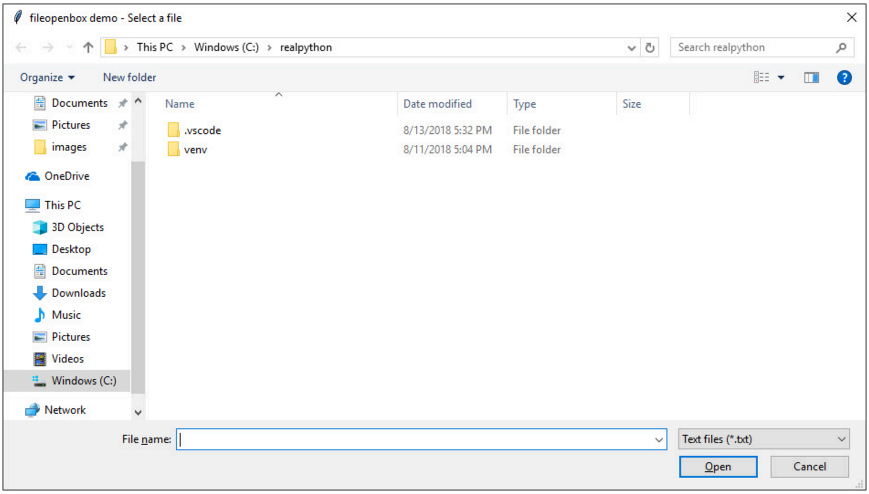
Aside from `msgbox()` and `buttonbox()`, EasyGUI provides several other functions for creating different kinds of windows. Try running each line in the code below to see the various types of windows created and how different values are returned.

```
>>> choices = ("Red", "Yellow", "Blue")
>>> title = "Choose wisely..."

>>> gui.indexbox("What is your favorite color?", title, choices)

>>> gui.choicebox("What is your favorite color?", title, choices)

>>> gui.multchoicebox(
...     "What are your favorite colors?", title, choices)

>>> gui.enterbox("What is your favorite color?", title)

>>> gui.passwordbox(
...     "What is your favorite color? (I won't tell.)", title)

>>> gui.textbox("Please describe your favorite color:")
```

File and Directory Selection Dialogs

Another useful window you can create with EasyGUI is a file selection dialog box. This is done with the `gui.fileopenbox()` function, which takes three string arguments: a message, the window title, and a file format. For example, the following creates a dialog box for opening a `.txt` file:

```
>>> gui.fileopenbox("Select a file", "fileopenbox demo", "*.txt")
```



fileopenbox demo - Select a file

Name	Date modified	Type	Size
.vscode	8/13/2018 5:32 PM	File folder	
venv	8/11/2018 5:04 PM	File folder	

In the previous example, the third argument passed to `fileopenbox()` is `"*.txt"`. This uses the **wildcard** `*` symbol to automatically filter the viewable files in the dialog to only those with a `.txt` extension. The user still has the option to select any file he or she wants by choosing “All files (.)” in the file type drop-down.

Notice also that the message and title strings are both displayed in the title bar of the dialog box. Typically, you would pass an empty string as the first argument so that only the title of the window is displayed. You can also pass specific arguments by assigning strings to **keywords** in the function call.

For example, the following displays a file selection dialog box with just the title and file type specified:

```
>>> gui.fileopenbox(title="Open a file...", default="*.txt")
```

This is a great way to pass arguments to the function because the code is much more readable. You can quickly see what each argument is by looking at the keyword it is assigned to. You will need to look at the documentation for any library you are working with to know what keyword arguments are accepted.

With the file selection dialog box open, select a file and click the "Open" button. You will notice that a string containing the full path to the selected file is returned. It is important to note that the file has not actually been opened. To do that you would need to use the `open()` function like you learned to do in Chapter 11.

What do you think happens if the user types in the name of a file that doesn't exist and tries to open it? Fortunately, the dialog box won't let them! This is one less thing you have to worry about programming into your code.

Just like a `msgbox()` and `buttonbox()`, the value `None` is returned if the user presses "Cancel" or closes the dialog box without selecting a file.

EasyGUI has two other functions that generate windows nearly identical to the one generated by `fileopenbox()`:

- The `diropenbox()` function that opens a window that can be used to select a folder instead of a file. When the user presses the "Open" button, the full path the directory is returned as a string.
- The `filesavebox()` function opens a window to select a file to be saved rather than opened. This dialog box also confirms that the user wants to overwrite the file if the chosen name already exists. When the "Save" button is pushed, the full path to the file is returned as a string. The file is not actually saved. You will have to code that functionality into your program.

Exit Your Program Gracefully

Suppose you have written a program for extracting pages from a PDF file (you will get a chance to do so in the assignment that follows this section). The first thing the program might do is use `fileopenbox()` to open a window to allow the user to select a file.

What do you do if the user decides they don't want to run the program and presses the "Cancel" button or closes the window?

You must make sure that your programs handle these situations gracefully, meaning the program shouldn't crash or produce any unexpected output. In the situation described above, the program should stop running altogether.

There are a few different ways to end the execution of a program programmatically. The simplest way is with Python's built-in `exit()` function.

> **Note**
>
> If you're running the script in IDLE, `exit()` will also close the current interactive window. It's very thorough.

For example, the following code snippet shows how to use `exit()` when the user presses the "Cancel" button in a file selection dialog box:

```
import easygui as gui

input_path = gui.fileopenbox(
    title="Choose a file...", default="*.txt")

if input_path is None:
    exit()
```

A Real-World Example

You have now seen enough to put together a practical GUI application! Let's build a small program for rotating the pages of a PDF file.

The flow of the program will work like this:

1. Display a file selection dialog for opening a PDF file.
2. If the user canceled the dialog (`None` was returned), then exit the program.
3. Otherwise, let the user select a rotation amount (90, 180 or 270 degrees).
4. Display a file selection dialog for saving the rotated PDF.
5. If the user tries to save a file with the same name as the input file:
    - Alert the user of the problem with a message box.
    - Return to step 4.
6. If the user canceled the "save file" dialog (`None` was returned), then exit the program.
7. Finally, open the selected PDF, rotate all of the pages, and save the rotated PDF to the selected file name using the techniques you learned in Chapter 13.

When designing an application, it is often beneficial to plan out each step as we have done above. For large applications, you may want to draw diagrams describing the program flow. This should be done before you start coding.

Now that we have a plan for the application let's tackle each step one at a time. Open a new script window in IDLE to follow along.

First, you need to import EasyGUI and PyPDF2 (if you need a refresher on working with PyPDF2, you may want to skim over Chapter 13):

```
import easygui as gui
from PyPDF2 import PdfFileReader, PdfFileWriter
```

Next, you need to display a file selection dialog box using EasyGUI's `fileopenbox()` function. You can assign the selected file path to a variable called `input_path`. If `input_path` is `None`, you should stop the program with the `exit()` function:

```
open_title = "Select a PDF to rotate..."
file_type = "*.pdf"
input_path = gui.fileopenbox(title=open_title, default=file_type)

if input_path is None:
    exit()
```

Now you need to allow the user to select an amount (in degrees) to rotate the PDF pages. You could use any number of the selection dialogs provided by EasyGUI, but for simplicity, let's use a `buttonbox()`. The available choices are 90, 180 and 270 degrees.

```
choices = ("90", "180", "270")
message = "Rotate the PDF clockwise by how many degrees?"
degrees = gui.buttonbox(message, "Choose rotation...", choices)
degrees = int(degrees)
```

Next, you need to get the output file path from the user using EasyGUI's `filesavebox()` function. According to the plan, the program should not let the user save to the same path as the input file path.

You can use a `while` loop to keep showing the user a warning and asking them to choose a file to save to until they pick a path that is different from the input file path.

To display the warning, you can use the `msgbox()` function. Remember that this function takes three arguments: a message, a title, and a button value. The default button value is "OK," which serves the purpose well here, so you only need to pass the first two arguments.

Finally, if the user presses "Cancel" on the file save dialog box, or closes it without selecting a file to save to, the program should exit.

Here's what all of this looks like:

```python
save_title = "Save the rotated PDF as..."
output_path = gui.filesavebox(title=save_title, default=file_type)

warn_title = "Warning!"
warn_message = "Cannot overwrite original file!"
while input_path == output_path:
    gui.msgbox(warn_message, warn_title)
    output_path = gui.filesavebox(title=save_title, default=file_type)

if output_path is None:
    exit()
```

Now you have everything you need to actually open the input file, rotate the pages and then save to a new file:

```python
input_file = PdfFileReader(input_path)
output_pdf = PdfFileWriter()

for page in input_file.pages:
    page = page.rotateClockwise(degrees)
    output_pdf.addPage(page)

with open(output_path, "wb") as output_file:
    output_pdf.write(output_file)
```

Review Exercises

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

1. Recreate the three different GUI elements pictured in this section by writing your own scripts without referring to the provided code.

2. Save each of the values returned from these GUI elements into new variables, then print each of them.

3. Test out `indexbox()`, `choicebox()`, `multchoicebox()`, `enterbox()`, `passwordbox()` and `textbox()` to see what GUI elements they pro-

duce. You can use the `help()` function to read more about each function in the interactive window. For instance, you can type:

```
>>> import easygui as gui
>>> help(gui.indexbox)
```

4. The GUI program for rotating PDF pages in this section has a problem. If the user closes the `buttonbox()` used to select degrees, the program crashes (try it out!). Fix this by using a `while` loop to keep displaying the selection dialog if `degrees` is `None`.

Leave feedback on this section »

17.2 Challenge: Write a GUI to Help a User Modify Files

Write a script `partial_PDF.py` that extracts a specific range of pages from a PDF file based on file names and a page range supplied by the user. The program should run as follows:

1. Let the user choose a file using EasyGUI's `fileopenbox()` function.
2. Let the user choose a beginning page to select using an `enterbox()`.
3. If the user enters invalid input, use a `msgbox()` to let the user know that there was a problem, then ask for a beginning page again using an `enterbox()`.
4. Let the user choose an ending page using another `enterbox()`.
5. If the user enters an invalid ending page, use a `msgbox()` to let the user know that there was a problem, then ask for a beginning page again using an `enterbox()`.
6. Let the user choose an output file name using a `filesavebox()`.
7. If the user chooses the same file as the input file, let the user know the problem using a `msgbox()`, then ask again for a file name using a `filesavebox()`.
8. Output the new file as a section of the input file based on the user-supplied page range. The user should be able to supply "1" to mean

the first page of the document. There are some potential issues that your script should be able to handle. These include:

- If the user cancels out of a box, the program should exit without any errors.
- Check that page numbers supplied are valid numbers.
- Check that the page range itself is valid.

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

Leave feedback on this section »

17.3 Introduction to Tkinter

For many basic tasks where GUI elements are needed one at a time, EasyGUI can save a lot of effort compared to creating an entire GUI program. If you want to build a complete GUI application, you need a more powerful solution.

Tkinter is a GUI framework (also sometimes called a GUI toolkit) that operates at a lower level than EasyGUI, making it much more powerful.

There are a lot of different tools available for building GUIs. The Tkinter framework is often considered the *de facto* Python GUI framework. It even comes bundled with Python by default!

A common criticism of Tkinter is that its GUI elements look outdated. If you want a shiny, modern interface, Tkinter may not give you what you need. However, Tkinter is lightweight, has no external dependencies, and is relatively simple to use compared to other toolkits. For many applications, it is an excellent choice.

Note

As mentioned in the last section, because IDLE is also built with Tkinter, you might encounter difficulties when running your own GUI programs within IDLE.

If you find that the GUI window you are trying to create is unexpectedly freezing or appears to be making IDLE misbehave in some unexpected way, try running your script via your command prompt (Windows) or Terminal (Mac/Linux) to check if IDLE is the real culprit.

Your First Tkinter Application

GUI applications exist as **windows**, which are just the application boxes you're used to using everywhere. Each has a title, a minimize button, a close button, and usually the ability to be resized.

Windows are composed of one or more **frames** that contain the application's contents, such as menus, text labels, and buttons. Frames are used to separate the window's content into different sections.

In traditional lingo, frames and all the different objects inside of them are called **widgets**. In this first example, you'll create a window that contains a single widget.

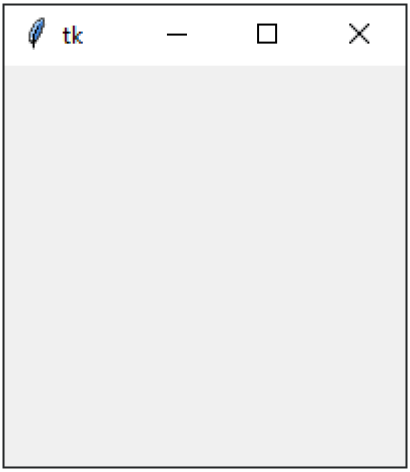
Start by opening a new interactive window in IDLE. The first thing you need to do is import the Tkinter module:

```
>>> import tkinter as tk
```

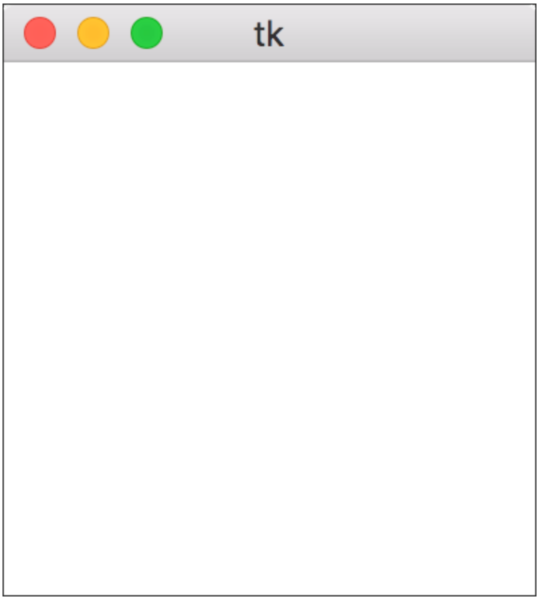
Windows in Tkinter are instances of Tkinter's `Tk` class. Go ahead and create a new window and assign it to the variable `window`:

```
>>> window = tk.Tk()
```

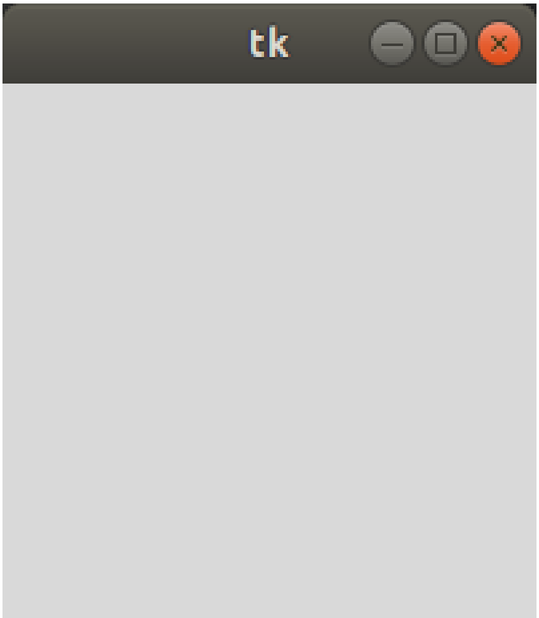
On Windows, a new window pops up:



The window looks like this on macOS:



On Ubuntu:



Windows-style app windows will be shown for the rest of this section. One take-away from these screenshots is that apps made with Tkinter look native to the platform that they are run on. This is not the case with every Python GUI framework.

Now that you have a window, the next step is to add a widget. You can create a simple text label using the `tk.Label` class, which can be initialized with a string for the label's text. Create a `Label` widget with the text "Hello, Tkinter" and assign it to a variable called `greeting`:

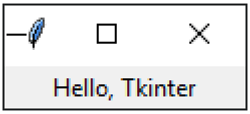
```
>>> greeting = tk.Label(text="Hello, Tkinter")
```

You may have noticed that the window you created didn't change. You just created a `Label` widget, but it hasn't been added to the window yet.

There are a number of ways to add widgets to windows. One of the simplest is with the widget's `.pack()` method. When you `.pack()` a widget into a window, Tkinter sizes the window as small as it can while still fully encompassing the widget. Check out what happens when you run the following:

```
>>> greeting.pack()
```

The window you created should now look like this:


Hello, Tkinter

Even this simple example is much more complicated than EasyGUI. As you will see, though, Tkinter is far more flexible.

Now let's look at how to run a Tkinter application from a script. If you re-type the example into a script and run it, no window appears.

When working with Tkinter in a script, you must call the `.mainloop()` method on your window (the instance of the `Tk` class) to show the window. Here's the same example from above in script form:

```
import tkinter as tk

window = tk.Tk()

greeting = tk.Label(text="Hello, Tkinter")
greeting.pack()

# Show the window
window.mainloop()
```

Work With Widgets

Widgets are the bread and butter of Tkinter. Each widget in Tkinter is defined by a class, just like the `Label()` widget you used in the simple application above. There are many widget classes available to you in Tkinter. The following table lists a few that you will use to create a couple of small applications in the next section:

Widget Class	Description
`Label`	A widget used to display text or an image on the screen.
`Button`	A button that can contain text or an image and can be tied to a Python function to perform an action.
`Text`	A text entry widget that allows multiline text entry. This is similar to an HTML `<textarea>` element.
`Entry`	A text entry widget that allows only a single line of text. If you are familiar with HTML, this is similar to an `<input>` element.
`Frame`	A rectangular region used to group related widgets or provide padding between widgets.

Note

Tkinter has many more widgets. These are just a few commonly used ones. You can see more in the Basic Widgets and More Widgets references at TkDocs.

You have already seen the `Label` widget, so let's look at `Button`.

`Button` Widgets

You create new `Button` widgets with the `tk.Button` class:

```
button = tk.Button()
```

All widgets are configurable via a number of attributes that control various visual properties. The particular attributes a widget has depend on the widget's class, but there is a lot of overlap. For example, most widgets have a `width` and `height` attribute that determine its size.

You can configure a widget's attributes in a couple of ways. One way is to access attributes by name using dictionary-style subscript notation. For instance, the following creates a `Button` widget whose width is 25 pixels and height is 5 pixels:

```python
button = tk.Button()
button["width"] = 25
button["height"] = 5
```

You can control the color of a widget with the `background` attribute:

```python
button["background"] = "blue"
```

There are several ways to assign a color. The easiest is to assign a valid color name as a string. Valid color names include `"red"`, `"orange"`, `"yellow"`, `"green"`, `"blue"`, `"purple"`, and many more.

Note

> You can find a handy chart of all valid color names here. For a full reference of color names, including macOS and Windows-specific system colors that are controlled by the current system theme, check out this list.

You can also specify a color using RGB values:

```python
button["background"] = "#34A2FE"
```

If you don't know what this means, don't worry about it right now. Just know that this method gives you far more control over the color than using a color name.

To set the Button's text, use the `text` attribute:

```python
button["text"] = "Click me!"
```

Most widgets that display text—like `Button` and `Label` widgets—have a `text` attribute. You can set the text color with `foreground`:

```python
button["foreground"] = "yellow"
```

Note

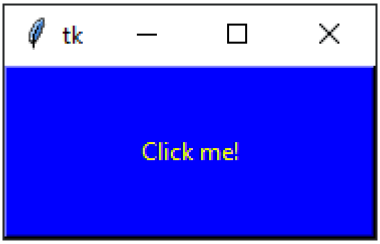
The background and foreground attributes have short aliases: bg and fg, respectively:

```
button["bg"] = "blue"
button["fg"] = "yellow"
```

Let's take at our configured Button. Open IDLE and run the following script:

```
import tkinter as tk

window = tk.Tk()

button = tk.Button()
button["text"] = "Click me!"
button["width"] = 25
button["height"] = 5
button["background"] = "blue"
button["foreground"] = "yellow"
button.pack()

window.mainloop()
```

This produces the following window:



You can also configure a widget using keyword arguments passed to its constructor. The following script uses this method to produce the same window as above:

```python
import tkinter as tk

window = tk.Tk()

button = tk.Button(
    text="Click me!",
    width=25,
    height=5,
    background="blue",
    foreground="yellow",
)
button.pack()

window.mainloop()
```

`Entry` Widgets

The `Entry` widget looks a lot like an HTML `<input>` element and is used for accepting input from the user's keyboard. Although `Entry` widgets display text, they do not have `text` attribute like `Label` and `Button` widgets.

You can insert text in an `Entry` widget with`.insert()` method, which inserts some text into the widget at a specified position:

```python
entry = tk.Entry()
entry.insert(0, "Hello!")
```

The above code creates an `Entry` widget and inserts the text `"Hello!"`. Calling `.insert()` on an `Entry` widget that already contains text inserts the new value at the specified position and shifts all of the other characters to the right. For example, assuming the above code has already been executed, suppose you execute the following:

```python
entry.insert(2, "a")
```

The text of the widget now reads `"Heallo!"`.

More often than not, you will not need to display text in an `Entry`, but rather retrieve the text a user has entered. To do this, use the `.get()` method:

```
text_entered = entry.get()
```

You'll see a real example using the `Entry` widget in the next section.

`Text` Widgets

The `Text` widget works in much the way was an `Entry` widget. You can insert text using the `.insert()` method and providing a numerical index:

```
text_box = tk.Text()
text_box.insert(0, "Hello")
```

You can also append text to any existing text in the in the `Text` widget by using the special constant `tk.END`. Assuming you had an existing `Text` widget like the one defined above, running the following would result in the text in the `Text` widget reading `"Hello World!"`:

```
text_box.insert(tk.END, " World!")
```

Just like `Entry` widgets, you can retrieve the text from a `Text` widget as a string using the `.get()` method. Unlike the `.get()` method for `Entry` widgets, you *must* supply at least one argument to the `Text` widget's `.get()` method. This argument specifies the numerical index of the character you want to retrieve.

To retrieve the first character of the first line, you would pass the float `1.0` to the `Text` widget's `.get()` method. If you wanted to get the fifth character from the third line, you would use `.get(3.5)`. This seems odd because the line indices start at 1, but the character indices start at 0! But, that's how it works!

To get more than just a single character, you pass two arguments to

`.get()`. The first is the index of the first character you want, again using a float to indicate line and character index, and the second in the index of the last character. The string returned included both the starting and ending characters.

For example, to get the word `"Hello"` from the above `Text` widget, you would do the following:

```
text_box.get(1.0, 1.4)
```

However, any index that would need to end in 0 (except for the zeroth character) needs to be increased by one. For example, to get the word `"World"` from the above `Text` widget, you need to retrieve the sixth and tenth characters from the first line. However, the following returns an empty string:

```
text_box.get(1.6, 1.10)
```

Instead, you need to do the following:

```
text_box.get(1.6, 1.11)
```

Often, you need to retrieve all of the text in a `Text` widget. You can do that by passing `1.0` and `tk.END` to `.get()`:

```
text_box.get(1.0, tk.END)
```

By setting the start index to `1.0` and the final index to `tk.END`, all of the text in `text_box` gets returned as a string.

Now that you have seen some basic widgets let's discuss how frames are used to organize widgets in your application.

Assign Widgets to Frames

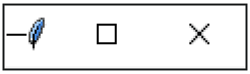
Up until now, you have been assigning widgets to the main application window. This works well for only the simplest GUI applications. For more complicated applications, you can use the `Frame` widget to group and organize related widgets in your application.

You create a frame just like you would any other widget. The following script creates a blank `Frame` and assigns it to the main application window:

```
import tkinter as tk

window = tk.Tk()
frame = tk.Frame()
frame.pack()

window.mainloop()
```

First, a new window is created with the `tk.Tk` class and a frame is created with the `tk.Frame` class. The `frame.pack()` method packs the frame into the window so that the window sizes itself as small as possible to encompass the frame.

When you run the above script, you get some seriously uninteresting output:



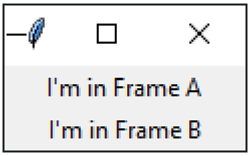
An empty `Frame` widget is practically invisible. They are best thought of as containers for other widgets. You can assign a widget to a frame by setting the widget's `master` attribute:

```
frame = tk.Frame()
label = tk.Label(master=frame)
```

To get a feel for how this works, let's write a script that creates two `Frame` widgets called `frame_a` and `frame_b`. `frame_a` will contain a label with the text `"I'm in Frame A"`, and `frame_b` will have the label `"I'm in Frame B"`. Here's one way to do that:

```python
import tkinter as tk

window = tk.Tk()

frame_a = tk.Frame()
label_a = tk.Label(master=frame_a, text="I'm in Frame A")
label_a.pack(side="top")

frame_b = tk.Frame()
label_b = tk.Label(master=frame_b, text="I'm in Frame B")
label_b.pack()

frame_a.pack()
frame_b.pack()

window.mainloop()
```

Notice that `frame_a` is packed into the window before `frame_b` in the code above. The window that opens shows the "Frame A" label on top of the "Frame B" label:

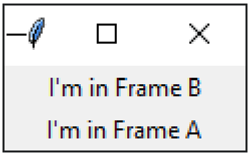

I'm in Frame A
I'm in Frame B

Now see what happens when you swap the order of `frame_a.pack()` and `frame_b.pack()`:

```python
import tkinter as tk

window = tk.Tk()

frame_a = tk.Frame()
label_a = tk.Label(master=frame_a, text="I'm in Frame A")
```

```
label_a.pack(side="top")

frame_b = tk.Frame()
label_b = tk.Label(master=frame_b, text="I'm in Frame B")
label_b.pack()

# Order of `frame_a` and `frame_b` is swapped
frame_b.pack()
frame_a.pack()

window.mainloop()
```

The output now looks like:



I'm in Frame B
I'm in Frame A

Now `label_b` is on top. It follows `frame_b` because it was assigned to it.

Adjust Frame Appearance With Reliefs

`Frame` widgets can be configured with a `relief` attribute that creates a border around the frame. You can set `relief` to be any of the following values:

- `tk.FLAT`, no border effect (this is the default value).
- `tk.SUNKEN`, which creates a sunken effect.
- `tk.RAISED`, which creates a raised effect.
- `tk.GROOVE`, which creates a grooved border effect.
- `tk.RIDGE`, which creates a ridged effect.

To apply the border effect, you must set the `borderwidth` attribute to a value greater than 1. This attribute adjusts the width of the border, in pixels.

The best way to get a feel for what each effect looks like is to see them for yourself. Here is a simple script that packs five `Frame` widgets into a window, each with a different value for the `relief` argument.

```python
import tkinter as tk

# 1
border_effects = {
    "flat": tk.FLAT,
    "sunken": tk.SUNKEN,
    "raised": tk.RAISED,
    "groove": tk.GROOVE,
    "ridge": tk.RIDGE,
}

window = tk.Tk()

for relief_name in border_effects:
    # 2
    frame = tk.Frame(master=window)
    frame["relief"] = border_effects[relief_name]
    frame["borderwidth"] = 5
    frame.pack(side=tk.LEFT)

    # 3
    label = tk.Label(master=frame)
    label["text"] = relief_name
    label["font"] = ("Helvetica", "12")
    label.pack()

window.mainloop()
```

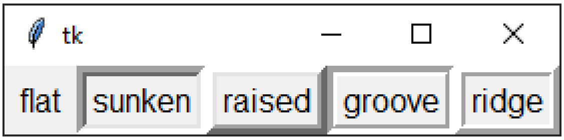
Let’s break that script down some.

1. First, a dictionary is created whose keys are the names of the dif-

ferent relief effects available in Tkinter, and whose values are the corresponding Tkinter objects. This dictionary is assigned to the `border_effects` variable.

2. After creating the window, object, a `for` loop is used to loop over each relief name in the `border_effects` dictionary. At each step in the loop, a new `Frame` widget is created and assigned to the `window` object. The `relief` attribute is set to the corresponding relief in the `border_effects` dictionary, and the `border` attribute is set so that the effect is visible. The `frame` is then packed into the window using the `.pack()` method. The `side` keyword argument you see is telling Tkinter which direction to pack the `frame` objects. You'll see more on how this works in the next section.
3. Finally, a `Label` widget is created to display the name of the relief and is packed into the `frame` object just created.

The window produced by the above script looks like this:



In this image, you can see that the `tk.FLAT` relief effect creates a flat looking frame. The `tk.SUNKEN` effect adds a border that gives the frame the appearance of being sunk into the window, while the `tk.RAISED` effect gives the frame a border that makes it appear to protrude from the screen. The `tk.GROOVE` adds a border that appears as a sunken groove around an otherwise flat frame, and the `tk.RIDGE` effect gives the appearance of a raised lip around the edge of the frame.

In this section, you learned about the basics of Tkinter including how to create a window, use widgets, and work with frames. At this point, you can make some simple windows displaying some messages, but a full-blown application is still out of reach.

In the next section, you'll learn how to control the layout of your applications using Tkinter's powerful geometry managers. Before you move on, take a crack at some of the review exercises to make sure you've mastered the foundations!

Review Exercises

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

1. Try to re-create all of the screenshots in this section without looking at the source code. If you get stuck, check the code and finish your re-creation. Then wait for 10–15 minutes and try again. Repeat this until you can produce all of the screenshots on your own. (Focus on the output. It's okay if your own code is slightly different from the code in the book.)

Leave feedback on this section »

17.4 Control Layout With Geometry Managers

Up until now, you have been adding widgets to windows and frames with the `.pack()` method, but you haven't been told what exactly this method does. Let's clear things up!

The `.pack()` method is an example of a **geometry manager**, which is a fancy way of saying that `.pack()` manages the layout of your windows and frames. This isn't the only way to manage the layout, however. There are two other geometry managers available in Tkinter: `.grid()` and `.place()`.

Each frame and window in your application can use only one geometry manager. However, different frames can use different geometry managers.

Note

Calling `.pack()` on one widget and `.grid()` on another can cause mayhem if both widgets are assigned to the same frame or window. Your program may freeze, and you will need to use `Ctrl+C`, or some other means, to force your program to quit.

If you notice your application freezes, it is a good idea to check and see if you are using two different geometry managers in the same frame or window.

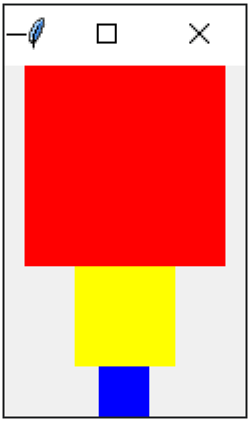
The `.pack()` Geometry Manager

The `.pack()` geometry manager is simple, but it is actually quite difficult to describe precisely what it does. You can think of it as trying to organize widgets the most efficiently in the given amount of space.

The best way to get a feel for `.pack()` is to look at an example. Let's see what happens when you `.pack()` three `Label` widgets into a frame:

```python
import tkinter as tk

window = tk.Tk()

frame1 = tk.Frame(master=window, width=100, height=100)
frame1["background"] = "red"
frame1.pack()

frame2 = tk.Frame(master=window, width=50, height=50)
frame2["background"] = "yellow"
frame2.pack()

frame3 = tk.Frame(master=window, width=25, height=25)
frame3["background"] = "blue"
frame3.pack()

window.mainloop()
```

The `.pack()` geometry manager by default places each frame below the previous one, in the order that they are assigned to the window:



Using `.pack()` does not give you tons of control, but it is useful in certain situations.

For example, suppose you want to stack three frames vertically, as you did in the previous example, but want to make sure each frame fills the entire width of the window. You can set the `fill` keyword argument to specify which direction the frames should fill. The options are `tk.X` to fill in the horizontal direction, `tk.Y` to fill vertically, and `tk.BOTH` to fill in both directions.

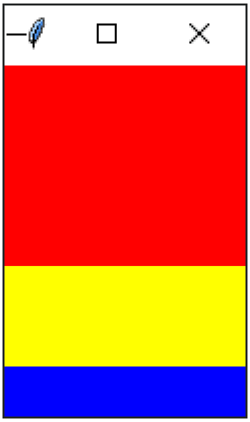
Here's how you would stack the three frames so that each one fills the whole window horizontally:

```
import tkinter as tk

window = tk.Tk()

frame1 = tk.Frame(master=window, height=100)
frame1["background"] = "red"
frame1.pack(fill=tk.X)
```

```
frame2 = tk.Frame(master=window, height=50)
frame2["background"] = "yellow"
frame2.pack(fill=tk.X)

frame3 = tk.Frame(master=window, height=25)
frame3["background"] = "blue"
frame3.pack(fill=tk.X)

window.mainloop()
```

Notice that the `width` keyword argument is missing from each frame. It is no longer necessary because the `.pack()` method on each frame is set to fill horizontally.

The above script produces a window that looks like this:



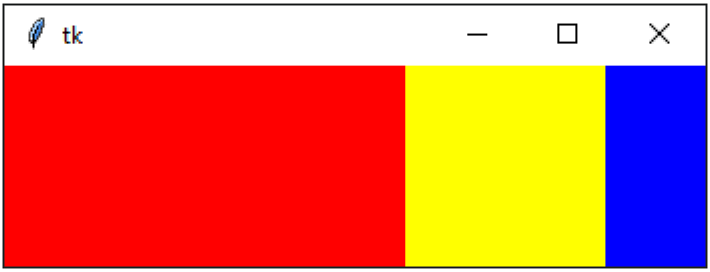
One of the nice things about filling the window with `.pack()` is that the fill is responsive to window resizing. Try widening the window generated by the previous script to see how this works.

If you need to put a bunch of frames side by side, you can use the `side` keyword argument of `.pack()` to specify the direction. The available options are `tk.TOP`, `tk.BOTTOM`, `tk.LEFT` and `tk.RIGHT`. For example, the following script places three frames side by side from left to right and expands each frame to fill the window vertically:

```
import tkinter as tk

window = tk.Tk()

frame1 = tk.Frame(master=window, width=200, height=100)
frame1["background"] = "red"
frame1.pack(side=tk.LEFT)

frame2 = tk.Frame(master=window, width=100)
frame2["background"] = "yellow"
frame2.pack(fill=tk.Y, side=tk.LEFT)

frame3 = tk.Frame(master=window, width=50)
frame3["background"] = "blue"
frame3.pack(fill=tk.Y, side=tk.LEFT)

window.mainloop()
```

This time, you have the specify the `height` keyword argument on at least one of the frames to force the window to have some height. In the above example, `frame1` is given a width and height of 200 pixels. Setting `fill=tk.Y` for the other frames forces them to fill the window in the vertical direction, and setting `side=tk.LEFT` arranges each frame horizontally.

The resulting window looks like this:



Just like setting `fill=tk.X` made the frames resize responsively when the window is resized horizontally, setting fill=tk.Y makes the frames resize responsively when the window is resized vertically. Try it out!

To make the layout truly response, you can set an initial size for your frames using the width and height attributes. Then set the fill keyword argument of the .pack() method to tk.BOTH and set the expand keyword argument to True:

```
import tkinter as tk

window = tk.Tk()

frame1 = tk.Frame(master=window, width=200, height=100)
frame1["background"] = "red"
frame1.pack(fill=tk.BOTH, side=tk.LEFT, expand=True)

frame2 = tk.Frame(master=window, width=100)
frame2["background"] = "yellow"
frame2.pack(fill=tk.BOTH, side=tk.LEFT, expand=True)

frame3 = tk.Frame(master=window, width=50)
frame3["background"] = "blue"
frame3.pack(fill=tk.BOTH, side=tk.LEFT, expand=True)

window.mainloop()
```

When you run the above script, you see a window that initially looks the same as the one generated in the previous example. The difference is that now you can resize the window however you want and the frames expand and fill the window responsively. Pretty cool!

The .place() Geometry Manager

You can use the .place() method of a widget to control the precise location that it should occupy in a frame or window. You must provide two keyword arguments, x and y that specify the x- and y-coordinates for the top-left corner of the widget.

Keep in mind that the "origin" is the top left corner of the frame or window, so you can think of the `y` argument of `.place()` as the number of pixels from the top of the window, and the `x` argument as the number of pixels from the left of the window.

Here's an example of how to `.place()` geometry manager works:

```
import tkinter as tk

window = tk.Tk()

# 1
frame = tk.Frame(master=window)
frame["width"] = 150
frame["height"] = 150
frame.pack()

# 2
label1 = tk.Label(master=frame)
label1["text"] = "I'm at (0, 0)"
label1["background"] = "red"
label1.place(x=0, y=0)

# 3
label2 = tk.Label(master=frame)
label2["text"] = "I'm at (75, 75)"
label2["background"] = "yellow"
label2.place(x=75, y=75)

window.mainloop()
```

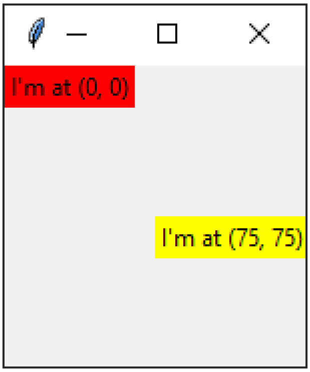
After creating a new window, the script works as follows:

1. Create a new `Frame` widget, called `frame1`, that is 150 pixels wide by 150 pixels tall, and pack it into the window using the `.pack()` method.
2. Create a new `Label`, called `label1`, with a yellow background and

place it in `frame1` at position (0, 0).

3. Create a new `Label`, called `label2`, with a red background and place it in `frame1` at position (75, 75).

Save and run the above script. The following window is displayed:



The `.place()` geometry manager is not often used. It suffers from two main problems. First, it can be tough to manage your layout if you have a lot of widgets. Moreover, the widgets are placed at an absolute position that does not change as the window is resized.

The `.grid()` Geometry Manager

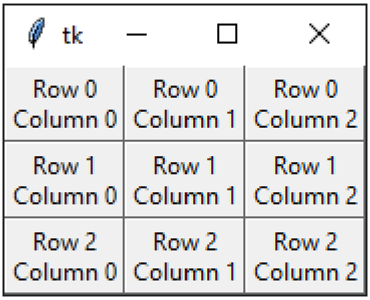
The geometry manager you will likely use the most often is the `.grid()` geometry manager. It provides more control than `.pack()` without the rigidity of `.place()`.

The `.grid()` geometry manager works by creating a grid of cells in a window or frame. For each widget, you must specify the row and column index using the `row` and `column` keyword arguments.

For example, the following script creates a 3 × 3 grid of frames with `Label` widgets packed into them:

```python
import tkinter as tk

window = tk.Tk()

for i in range(0, 3):
    for j in range(0, 3):
        frame = tk.Frame(master=window)
        frame["relief"] = tk.RAISED
        frame["borderwidth"] = 1
        frame.grid(row=i, column=j)

        label = tk.Label(master=frame)
        label["text"] = f"Row {i}\nColumn {j}"
        label.pack()

window.mainloop()
```

Here's what the resulting window looks like:



Row 0 Column 0	Row 0 Column 1	Row 0 Column 2
Row 1 Column 0	Row 1 Column 1	Row 1 Column 2
Row 2 Column 0	Row 2 Column 1	Row 2 Column 2

This is the first example that shows two geometry managers being used at the same time. Each frame is attached to the `window` with the `.grid()` geometry manager. Similarly, each `label` is attached to its master frame with the `.pack()` method.

The important thing to realize here is that even though the `.grid()` method is called on each `frame` object, the geometry manager applies to the `window` object. Similarly, the layout of each `frame` is controlled with the `.pack()` geometry manager.

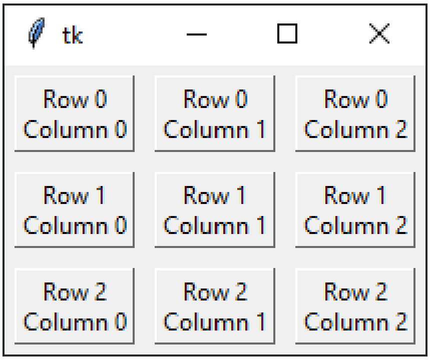
You can control the layout even more by adding some padding to the widgets. **Padding** is just some blank space that surrounds a widget and separates it visually from its contents.

There are two types of padding: **external padding** and **internal padding**. External padding adds some space around the outside of a widget. It is controlled with two keyword arguments that you pass to the `.grid()` method: `padx`, which adds padding in the horizontal direction, and `pady`, which adds padding in the vertical direction.

Let's add some padding around the outside of the frames in the previous example:

```python
import tkinter as tk

window = tk.Tk()

for i in range(0, 3):
    for j in range(0, 3):
        frame = tk.Frame(master=window)
        frame["relief"] = tk.RAISED
        frame["borderwidth"] = 1
        frame.grid(row=i, column=j, padx=5, pady=5)

        label = tk.Label(master=frame)
        label["text"] = f"Row {i}\nColumn {j}"
        label.pack()

window.mainloop()
```

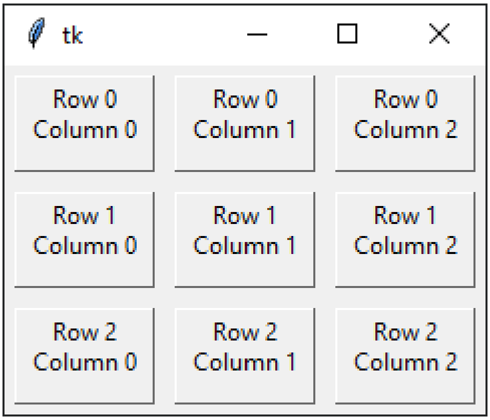
Here's the resulting window:



Now let's see what happens if we add some internal padding to each frame:

```
import tkinter as tk

window = tk.Tk()

for i in range(0, 3):
    for j in range(0, 3):
        frame = tk.Frame(master=window)
        frame["relief"] = tk.RAISED
        frame["borderwidth"] = 1
        frame.grid(row=i, column=j, padx=5, pady=5, ipadx=5, ipady=5)

        label = tk.Label(master=frame)
        label["text"] = f"Row {i}\nColumn {j}"
        label.pack()

window.mainloop()
```

The window now looks like this:



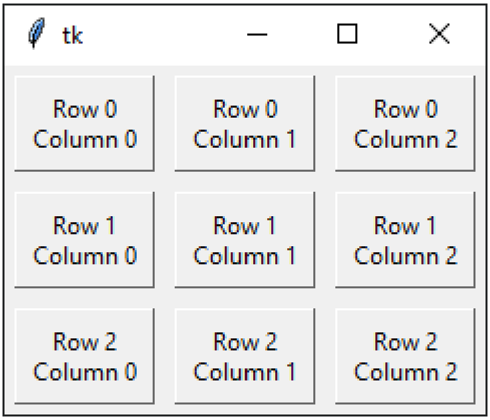
If you look closely, you'll notice that the internal padding for the frame is working just fine in the horizontal direction. Each `Label` widget has about 5 pixels of padding to its left and right. However, the padding in the vertical direction is *there* but the `Label` seems to have all of the padding below, and none above it.

This happens because we are packing each `Label` into its corresponding `Frame`. By default, the `side` argument of the `.pack()` method is set to `tk.TOP`, and this overrides and padding that might be there.

You can instead add some external padding to the `Label` widget by passing values to the `padx` and `pady` values of the `.pack()` method:

```
import tkinter as tk

window = tk.Tk()

for i in range(0, 3):
    for j in range(0, 3):
        frame = tk.Frame(master=window)
        frame["relief"] = tk.RAISED
        frame["borderwidth"] = 1
        frame.grid(row=i, column=j, padx=5, pady=5)
        label = tk.Label(master=frame)
        label["text"] = f"Row {i}\nColumn {j}"
        label.pack(padx=5, pady=5)

window.mainloop()
```

The window produced now has some padding around each `Label` widget:



That looks pretty nice, but if you try and expand the window in any direction, you'll notice that the layout isn't very responsive. The whole grid stays at the top left corner of the window.

You can adjust how the rows and columns of the grid grow as the window is resized using the `.columnconfigure()` and `.rowconfigure()` methods on the `window` object. Remember, the grid is attached to `window`, even though you are calling the `.grid()` method on each `Frame` widget.

Both `.columnconfigure()` and `.rowconfigure()` takes three arguments:

1. The index of the column or row in the grid that you want to configure
2. A keyword argument called `weight`, which defaults to 0, determines how the column or row should respond to window resizing relative to the other columns and rows

3. A keyword argument called `minsize` that sets the minimum size in pixels of the row height or column width

If `weight` is set to `0`, the column or row does not expand as the window resizes. If every column and row is given a weight of 1, they all grow at the same rate. If one column has a weight of 1 and another a weight of 3, then the second column expands at three times the rate of the first as the window is resized.

Let's adjust the previous script to handle window resizing better:

```python
import tkinter as tk

window = tk.Tk()

for i in range(0, 3):
    window.columnconfigure(i, weight=1, minsize=75)
    window.rowconfigure(i, weight=1, minsize=50)

    for j in range(0, 3):
        frame = tk.Frame(master=window)
        frame["relief"] = tk.RAISED
        frame["borderwidth"] = 1
        frame.grid(row=i, column=j, padx=5, pady=5)

        label = tk.Label(master=frame)
        label["text"] = f"Row {i}\nColumn {j}"
        label.pack(padx=5, pady=5)

window.mainloop()
```

Notice that the `.columnconfigure()` and `.rowconfigure()` methods are placed in the body of the outer `for` loop. You could configure each column and row outside of the `for` loop, but this would require writing an additional six lines of code. Utilizing the loop cuts down on code length while maintaining readability.

On each iteration of the loop, the `i`-th column and row are configured to have a weight of 1, so each row and column expands at the same rate. The minsize argument is set to 75 for each column and 50 for each row. This makes sure the Label widget always displays its text without chopping off any characters.

Try running the script to get a feel for how it works!

You can also align objects inside of a grid cell to the top, right, bottom, or left part of the cell. By default, objects are aligned to the top of the grid cell.

To specify how objects should be aligned in the cell, use the sticky keyword argument of the .grid() method. It is called sticky because it forces an object to stick to one side of the cell. You can set sticky to any one of the following:

- tk.N to align to the top of the cell
- tk.E to align to the right side of the cell
- tk.S to align to the bottom of the cell
- tk.W to align to the left side of the cell

The names tk.N, tk.E, tk.S and tk.W come from the cardinal directions: north, south, east, and west.

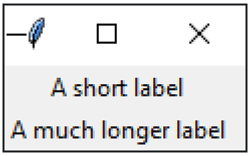
The sticky argument is useful for aligning labels. By default, labels are centered in the grid cell. For example, consider the following script:

```
import tkinter as tk

window = tk.Tk()

label1 = tk.Label()
label1["text"] = "A short label"
label1.grid(row=0, column=0)

label2 = tk.Label()
```

```
label2["text"] = "A much longer label"
label2.grid(row=2, column=0)

window.mainloop()
```

This script creates two labels: `label1`, which contains a short string, and `label2`, which contains a longer string of text. The `.grid()` method is used to put `label1` and `label2` into `window`, with `label1` on the first row and `label2` on the second row.

The window generated by this script looks like this:



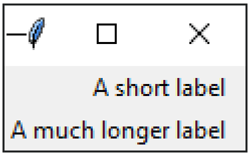
A short label

A much longer label

Notice that `label1` is centered above `label2`. To align both labels to the ride side of their corresponding grid sells, set `sticky=tk.E` in each call to `.grid()`:

```
import tkinter as tk

window = tk.Tk()

label1 = tk.Label()
label1["text"] = "A short label"
label1.grid(row=0, column=0, sticky=tk.E)

label2 = tk.Label()
label2["text"] = "A much longer label"
label2.grid(row=2, column=0, sticky=tk.E)

window.mainloop()
```

No the window generated by running script shows both labels aligned to the right:


A short label
A much longer label

You may notice that the text in `label2` doesn't look like it has changed position at all. That's because the width of the window always wraps around the widest element as tightly as possible. So, even though you told `label2` to stick the right edge of the cell, it doesn't look like anything has changed because the width of the whole window is exactly equal to the width of `label2`.

The `.grid()` geometry manager offers quite a bit more flexibility than you have seen here, including cells that span multiple rows and columns. For more information, check out the Grid Geometry Manager section of the TkDocs tutorial.

Now that you have the basics of Tkinter's geometry managers down, the next step is to assign actions to buttons to bring your applications to life. In the next section, you'll learn how to do this by building two apps: an interactive temperature converter and a simple text editor.

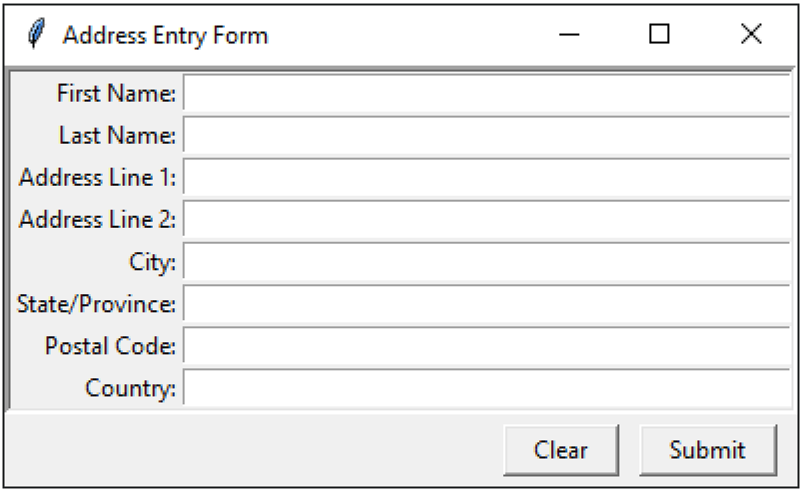
But first, make sure you've mastered this section's content by working on the following review exercises.

Review Exercises

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

1. Try to re-create all of the screenshots in this section without looking at the source code. If you get stuck, check the code and finish your re-creation. Then wait for 10–15 minutes and try again. Repeat this until you can produce all of the screenshots on your own. (Focus on the output. It's okay if your own code is slightly different from the code in the book.)

2. Below is an image of a window made with Tkinter. Try and recreate the window using the techniques you have learned thus far.



Leave feedback on this section »

17.5 Make Your Applications Interactive

By now, you have a pretty good idea how to create a window in Tkinter, add some widgets, and control the application layout. But if you want your application to actually do anything, you're kind of stuck.

In this section, you'll see how to handle user interaction by building two simple applications. First, you will build a simple temperature conversion app that converts Celsius temperatures to Fahrenheit. Then you will build a simple text editor that allows you to open, edit and save text files.

Before you can build these apps, though, you need to learn how to connect buttons to actions.