

Python Programming

from Beginner to

Paid Professional

Part 1

Learn Python for Automation & IT

with Video Tutorials

By

A. J. Wright

Python Programming from Beginner to Paid Professional

Copyright © AB Prominent Publisher

9791220820387

Published in the United States

All rights No part of this book and the accompanying video tutorials may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews. Every effort has been made in the preparation of this book and the accompanying videos to ensure the accuracy of the information presented. However, the information contained in this book and the videos is sold without warranty, either express or implied. The author/publisher, its dealers and distributors will not be held liable for any damages caused or alleged to be caused directly or indirectly by this book and its videos. The author/publisher has endeavored to provide trademark information about all the companies and products mentioned in this book. However, he cannot guarantee the accuracy of this information.

Table of Contents

[About the Author](#)

[How this Book can Help You](#)

[How to Use the Video Tutorials, Programs & Practice Exercises](#)

[Why Learn Python Programming?](#)

[1. The Foundation: Getting Started with Python Programming](#)

[1.1. Specialization Introduction](#)

[1.2. Course Introduction](#)

[1.3. A Minute to Set Yourself up for Success](#)

[1.4. Welcome to the Course](#)

[1.4.1. How the 6-Week Deadline Works](#)

[1.4.2. Getting and Giving Help](#)

[1.4.3. Finding Out More Information](#)

[1.5. Official Python Discussion Forums: Join, Meet & Greet](#)

[2. Introduction to Programming](#)

[2.1. The Beginning of Your Programming Journey.](#)

[2.2. What is Programming?](#)

[2.2.1. Difference Between *Script* and *Program*](#)

[2.3. What is Automation?](#)

[2.4. Getting Computers to Work for You](#)

[2.5. Discussion Forums: Your Hopes for Automation](#)

[2.6. Practice Quiz 1: Introduction to Coding in General - 5 Questions](#)

[2.6.1. Answers to Practice Quiz 1](#)

[3. Setting up Your Python & Programming Environment](#)

[3.1. What is Python?](#)

[3.1.1. How to Execute Python Codes](#)

[3.2. A Note on Syntax and Code Blocks](#)

[3.3. Why is Python Relevant to IT?](#)

[3.4. How to Become a Pythoneer or Pythonista](#)

[3.5. Other Languages](#)

[3.6. Practice Quiz 2: Introduction to Python - 5 Questions](#)

[3.6.1. Answers to Practice Quiz 2](#)

[4. Hello, World!](#)

[4.1. How to Write Hello World in Python](#)

[4.1.1. Program Comments \(#\)](#)

[4.1.2. How to Write Comments](#)

[4.2. How to Get Information from the User](#)

[4.3. Python Can Be Your Calculator](#)

[4.4. Cheat Sheet 1: First Programming Concepts](#)

[4.5. Practice Quiz 3: Hello World - 5 Questions](#)

[4.5.1. Answers to Practice Quiz 3](#)

[5. Module Review](#)

[5.1. First Steps Wrap Up](#)

[5.2. Module 1 Graded Assessment - 10 Questions](#)

[5.2.1. Solutions to Module 1 Graded Assessment](#)

[6. Expressions and Variables](#)

[6.1. Basic Python Syntax introduction](#)

[6.2. Data Types](#)

[6.3. Data Types Recap](#)

[6.4. Variables](#)

[6.4.1. Variable Name Restrictions](#)

[6.5. Expressions, Numbers and Type Conversions](#)

[6.6. Implicit versus Explicit Conversion](#)

[6.7. Practice Quiz 4: 5 Questions](#)

[6.7.1. Answers to Practice Quiz 4](#)

[7. Functions](#)

[7.1. Defining Functions](#)

[7.2. Defining Functions Recap](#)

[7.3. Returning Values](#)

[7.4. Returning Values Using Functions](#)

[7.5. The Principles of Code Reuse](#)

[7.6. Code Style](#)

[7.6.1. Principles for Creating Well-styled Code](#)

[7.7. Practice Quiz 4: 5 Questions](#)

[7.7.1. Answers to Practice Quiz 5](#)

[8. Conditionals](#)

[8.1. Comparing Things](#)

[8.2. Comparison Operators Recap](#)

[8.3. Branching with IF statements](#)

[8.4. If Statements Recap](#)

[8.5. Else Statements](#)

[8.6. Else Statements and Modulo Operator Recap](#)

[8.7. Elif Statements](#)

[8.8 Cheat Sheet 2: Conditionals](#)

[8.9. More Complex Branching with elif Statements](#)

[8.10. Practice Quiz 6: 5 Questions](#)

[8.10.1. Answers to Practice Quiz 6](#)

[9. Module Review](#)

[9.1. Basic Syntax Wrap Up](#)

[9.2. Why I Like Python](#)

[9.3. What I Don't Like About Python](#)

[9.4. Module 2 Graded Assessment - 10 Questions](#)

[9.4.1. Solutions to Module 2 Graded Assessment:](#)

[10. While Loops](#)

[10.1. Introduction to Loops](#)

[10.2. What is a While loop?](#)

[10.3. Anatomy of a While Loop](#)

[10.4. More While Loop Examples](#)

[10.5. Why Initializing Variables Matters](#)

[10.6. Common Pitfalls with Variable Initialization](#)

[10.7. Infinite Loops and How to Break Them](#)

[10.8. Infinite loops and Code Blocks](#)

[10.9. Practice Quiz 7: 5 Questions](#)

[10.9.1. Answers to Practice Quiz 7](#)

[11. For Loops](#)

[11.1. What is a For Loop?](#)

[11.2. For Loops Recap](#)

[11.3. More for Loop Examples](#)

[11.4. A Closer Look at the Range\(\) Function](#)

[11.5. Nested For Loops](#)

[11.6. Common Errors in For Loops](#)

[11.7. Cheat Sheet 3: Loops](#)

[11.8. Practice Quiz 8: 4 Questions](#)

[11.8.1. Answers to Practice Quiz 8](#)

[12. Recursion \(Optional\).](#)

[12.1. What is recursion?](#)

[12.2. Recursion in Action in the IT Context](#)

[12.3. Additional Recursion Sources](#)

[12.4. Practice Quiz 9: 5 Questions](#)

[12.4.1. Answers to Practice Quiz 9](#)

[13. Module Review](#)

[13.1. Loops Wrap Up](#)

[13.2. Module 3 Graded Assessment – 10 Questions](#)

[13.2.1. Solutions to Module 3 Graded Assessment:](#)

[14. Strings](#)

[14.1. Basic Structures Introduction](#)

[14.2. What is a string?](#)

[14.3. The Parts of a String](#)

[14.4. String Indexing and Slicing Recap](#)

[14.5. Creating New Strings](#)

[14.6. Basic String Methods](#)

[14.7. More String Methods](#)

[14.8. Advanced String Methods](#)

[14.9. Formatting Strings](#)

[14.10. String Formatting Recap](#)

[14.11. Cheat Sheet 4: String Reference](#)

[14.12. Cheat Sheet 5: Formatting Strings](#)

[14.13. Practice Quiz 10: 5 Questions](#)

[14.13.1. Answers to Practice Quiz 10](#)

[15. Lists](#)

[15.1. What is a List?](#)

[15.2. Lists Defined](#)

[15.3. Modifying the Contents of a List](#)

[15.4. Modifying Lists](#)

[15.5. Lists and Tuples](#)

[15.6. Tuples Recap](#)

[15.7. Iterating over Lists and Tuples](#)

[15.8. Iterating Over Lists Using Enumerate](#)

[15.9. List Comprehensions 1](#)

[15.10. List Comprehensions Recap](#)

[15.11. Cheat Sheet 6: Lists and Tuples Operations](#)

[15.12. Practice Quiz 11: 6 Questions](#)

[15.12.1. Answers to Practice Quiz 11](#)

[16. Dictionaries](#)

[16.1. What is a Dictionary?](#)

[16.2. Dictionaries Defined](#)

[16.3. Iterating over the Contents of a Dictionary.](#)

[16.4. Iterating Over Dictionaries Recap](#)

[16.5. Dictionaries versus Lists](#)

[16.6. Cheat Sheet 7: Dictionary Methods](#)

[16.7. Practice Quiz 12: 5 Questions](#)

[16.7.1. Answers to Practice Quiz 12](#)

[17. Module Review](#)

[17.1. Basic Structures Wrap Up](#)

[17.2. Module 4 Graded Assessment – 10 Questions](#)

[17.2.1. Solutions to Module 4 Graded Assessment](#)

[18. Object-oriented Programming \(OOP\)](#)

[18.1. OOP Introduction](#)

[18.2. What is OOP?](#)

[18.3. Definition of OOP](#)

[18.4. Classes and Objects in Python](#)

[18.5. Classes and Objects in Detail](#)

[18.6. Defining New Classes](#)

[18.7. Defining Classes Recap](#)

[18.8. Practice Quiz 13: 5 Questions](#)

[18.8.1 Answers to Practice Quiz 13](#)

[19. Classes and Methods](#)

[19.1 Instance Methods](#)

[19.2. What Is a Method?](#)

[19.3. Constructors and Other Special Methods](#)

[19.4. Special Methods Recap](#)

[19.5. Documenting Functions, Classes and Methods](#)

[19.6. Documenting with Docstrings](#)

[19.7. Cheat Sheet 8: Classes and Methods](#)

[19.8. About Jupyter Notebooks \(Optional\).](#)

[19.9. Help with Jupyter Notebooks](#)

[19.10. Challenge Lab 1: Methods and Classes Lab](#)

[20. Code Reuse](#)

[20.1. Inheritance](#)

[20.2. Object Inheritance](#)

[20.3. Composition](#)

[20.4. Object Composition](#)

[20.5. Python Modules](#)

[20.6. Augmenting Python with Modules](#)

[20.7. Supplemental Reading for Code Reuse](#)

[20.8. Challenge Lab 2: Code Reuse Lab](#)

[21. Module Review](#)

[21.1. OOP Wrap Up](#)

[21.2. Challenge Lab 3: Practice Notebook \(Object Oriented Programming\)](#)

[22. Writing a Script from the Ground Up](#)

[22.1. Final Project Introduction](#)

[22.2. Problem Statement](#)

[22.3. Research](#)

[22.4. Planning](#)

[22.5. Writing the Script](#)

[22.6. Putting It All Together](#)

[22.7. Challenge Lab 4: Putting It All Together](#)

[23. Final Project](#)

[23.1. Final Project Overview](#)

[23.2. Final Project Help](#)

[23.2.1 Project goal](#)

[23.3. Final Project \(Challenge Lab 5\).](#)

[23.3.1. Instructions](#)

[23.4. Final Project Grading](#)

[24. Course Wrap up](#)

[24.1. Congratulations!](#)

[24.2. Discussion Forums: Share Your Learner Journey.](#)

[24.3. Sneak Peek of the Next Course \(Part 2\).](#)

[25. How to Download the Course Resources](#)

[25.1. How to Get Further Help](#)

[25.2. More Helpful Resources](#)

About the Author

I have well over 15 years of software development experience. In the last few years I used my experience to develop solutions using various programming languages on Windows, Linux, MacOS and PLCs (programmable logic I also built solutions from scratch and went as far as modifying open source software to meet my client's needs.

I study hard and carry out researches constantly. I also love helping people get jobs or making their businesses successful. Nowadays, I work with a dedicated team of Python programmers who look into specific automation problems and proffer lasting solutions to them.

How this Book can Help You

This is not just *another* Python programming book. It is an intensive and *practical* Python programming course. It is part 1 of a 3-part series which serves as my exhaustive collection of step-by-step tutorials on the latest version 3 of Python programming language. It is a self-paced course that is excellent for beginners and accomplished experts alike. If you want to have fun learning or revising your Python programming with ease, this is the right course for you.

You will find this book indispensable if you are a computer programmer, an automation engineer or professional, a system administrator working in an IT firm, a data analyst/journalist, an educator, a computer science student or just anyone looking to acquire Python programming skills they need to succeed in their job or career. Yes, this course is exactly what you need to become a Pythoneer or Pythonista.

This course has 6 modules spread out over 25 chapters of both rich text and visual tutorials. You're not in this alone. I'm going to help you through it. Watching people coding is very different from learning how to code. So, you will not only be learning Python in this course, you will also be *doing*.

As you complete the tutorials, you're also going to get tested *a lot* on the materials we are covering by following Python best practices. Although this is a self-paced course, I strongly recommend that you complete it in not more than 6 weeks. For example, if you can complete one module every week, you can finish the course in 6 weeks.

To fully understand the basics of Python 3 programming, I strongly recommend you watch all the **53 in-depth HD videos** which are available in the **course resources folder** that you can download. The link for download is in Chapter 25 of this book. These video tutorials simplify everything you need to understand, and help you **speed up your**

Important terms and definitions discussed in this book are printed in bold texts, **like** Practice quizzes and answers are included at the end of each chapter to help you test how much you have improved. Go to Chapter 25 right now. You will find the link to the **course resources** Once you open this link, you will be able to **download all the course videos, graded assessments and their solutions, and handy cheat sheets that give you all the information you need at a**

How to Use the Video Tutorials, Programs & Practice Exercises

Except for revision purposes, the videos are not intended to be watched in isolation without first studying the content of this book. You should watch each video when you reach the point where it is referenced. This will ensure you fully understand the concept being discussed. The serial number and title (file name) of the relevant video to watch is stated at the point where it is referenced.

To have a solid foundation of Python, you must have an in-depth knowledge and develop Python coding skills quickly. Therefore, I strongly advise you attempt all the practice exercises like quizzes, graded assessments and projects in this book *on your* and *before* you check the answers and solutions provided. Write your own program as soon as you see me write mine in each video. Then cross check your programs with mine. Feel free to pause or rewatch the videos any time. You can also use or modify any of my codes as you wish.

Since I assume you have no knowledge of Python programming, I prepared this course in such a way that when you study it along with the accompanying videos you will not only have an in-depth knowledge of Python 3 programming, you will also gain a lot of job experience you need to build automation and innovation, and earn higher salary.

This book begins with the fundamental knowledge you need to start writing your very first Python script. It goes on to teach the advanced topics you need to become a paid professional in the field of Python programming. So, after completing this course you will have a clear understanding of Object-oriented Programming and be able to apply it to real world industrial automation.

The methods presented in this book and the accompanying videos are those that are usually employed in the real world of IT and are the important ones you really need to learn. So the information in this book is very valuable, not only to those who are just starting out, but also to other intermediate Python programmers.

Merely reading a copious Python programming book, or referring to Python help contents, is far from enough for learning how to “speak” Python language fluently. This book, unlike many others, takes the practical approach. It is designed in a course format with rich practice quizzes, assessments, challenge labs and interesting projects (with solutions) to engage you and give you hands-on practical experience.

First, this book will give you a big head start if you have never written a line of code before. Then it will teach you the techniques you need to learn, design and build anything from simple to complex and very useful Python programs.

Why Learn Python Programming?

If you learn Python, you will go mainstream. In case you haven't noticed, there are hundreds of today's most successful technology/IT companies using Python programs, such as Google, Netflix, Instagram, Reddit, Lyft, Spotify and so many others. Python is also being used at Bloomberg, the New York Times, and even at many local banks.

Python has many clear paths to finding meaningful job and work. For example, here is the link to [apply to top remote USA Python developer](#)

Although some of these potential jobs are quite obvious - such as becoming a Python developer - there are other careers where the knowledge of Python is an asset that are more unexpected.

Module 1

“If milk gets bad it becomes yoghurt. Yoghurt is more valuable than milk. If it gets even worse, it turns to cheese. Cheese is more valuable than yoghurt. You are not bad because you made mistakes.

Mistakes are the experiences that make you more valuable as a person. Christopher Columbus made a navigational error that made him discover America. Alexander Fleming’s mistake led him to invent Penicillin. Don’t let your mistakes get you down. It is not practice that makes perfect. Our mistakes enable us to learn to make ourselves perfect!” - Josh Aisosa

1. The Foundation: Getting Started with Python Programming

In this module I'll introduce you to the book's course format. Then, we'll dive into the basics of programming languages and syntax, as well as automation using scripting. I'll also introduce you to the Python programming language and some of the benefits it offers. Last up, we'll cover some basic functions and keywords of the language, along with some arithmetic operations.

1.1. Specialization Introduction

Working in IT is more than just a job. It's a career path. Research shows that the field of IT support is a launchpad for future career growth and better wages. In fact, a study on the subject was recently conducted by the Harvard Business School Accenture and Burning Glass entitled "Bridge the Gap".

It found that among today's middle-skilled jobs which require training but not a formal college degree, IT support offers clear pathways to prosperity. I saw this phenomenon play out at Google, in their IT support program. Those who push themselves to learn how to code in Python typically saw strong career growth.

They built skills that are critical to accessing higher level positions in the IT field, and after honing those skills through hard work and determination, they advanced into more technical IT support specialists. Some of them are systems administrators, technical solutions engineers, and even site reliability engineers. The common thread across all of these roles is knowing how to write code to solve problems and automate solutions.

By expanding your toolbox to include coding skills, you open a window into the world of systems management that can lead you towards more advanced technical roles down the line. Python in particular is having a huge surge. According to the 2019 Stack Overflow Developer Survey, Python is the coding language most people want to learn. The second most loved by those who already know it and the fourth most popular overall.

So why take this course to learn how to code in Python? Well, first it's geared towards people who are already in or aspiring to be in the field of IT. Maybe you're thinking bigger about your current IT role and want to work towards managing operations at scale, or maybe you're just starting out and looking to break into the IT industry.

Perhaps you've taken an IT Support Professional Certificate program somewhere already, or you have equivalent IT support knowledge with basic computing skills, like working with files and directories, familiarity with networking concepts, and understanding how to install software on your computer. In any case, this course is tailor-made for you. Second, this course offers three hands-on methods of teaching coding, Python and automation: Code blocks, Jupyter notebooks, and labs.

Third, I worked with an awesome group of Googlers who helped me prepare this course. They all started their careers in IT support, then learned programming and moved onto more technical roles like me. I can't wait to share our stories with you on how we use Python in our day-to-day activities. So, I will introduce you to the Python programming language with a special focus on how this language applies to automating tasks in the world of IT systems support and administration. I'm super excited to teach you this course.

When I was younger, I had no idea that careers in IT even existed. Later, I remember going to

a System Administration Summit where there were hundreds of men and about three women that were sysadmins. A lot has changed since then but there's still so much we can do to bring new ideas and representation into the IT field. That's why I want to share my knowledge with as many people as possible (and the reason I prepared this book in a rich course format).

I love my Python programming and I love the people I work with because they make it easy to ask for help and offer their guidance. This type of support network allows our team and ultimately our industry to be more successful. I understand from experience that it can feel pretty intimidating and maybe even a bit scary to learn a coding language.

Just remember, everyone started where you are right now with the first command, the first script, and of course, the first of many errors. When I started out in my career, I strive to get everything perfectly right the first time I tried it. But that actually slowed down my progress. So don't be afraid to make mistakes, it will give you a leg up. So let's get down to it. What's ahead?

This book (Part 1) begins with a crash course in Python where you will learn to write simple programs and understand their role in automation. Next (Part 2), we'll get more hands-on focus on how Python interacts with the operating system. After that, we'll cover how to use Git and GitHub to manage versions of your code. Then we will focus on troubleshooting and debugging techniques to find and solve the root cause of problems in IT infrastructure.

The next course (Part 3) covers automating at scale where you will learn to deploy configuration management on a fleet of either physical or virtual machines running in the Cloud.

Last up, we will bring all this knowledge together and complete two final projects designed to solve tasks that you might encounter in real-world IT settings. You can post your projects to GitHub to show off your fancy new skills to employers or friends or both.

That was a lot to rattle off! Are you excited? You're in very good hands!

So, Let's get ready to learn some new skills and maybe even have some laughs along the way. Let's dive in straight to the next section.

1.2. Course Introduction

If you work in IT, computer programming skills open up an incredible amount of opportunity. Being able to write scripts and programs that tell your computer to perform a task equips you with an invaluable tool. Not only does it make your work easier and more efficient, it can help you grow faster and advance further in your IT career.

But how do you even start to learn a programming language like Python? How do you recognize when to tell a computer to perform a task? And how do you then write a program to actually get your computer complete the task you want it to do? The thought of learning to write a program in Python can make you feel a whole bunch of emotions excitement, anticipation that feeling of wanting to dive right in and get going and also fear.

You might ask yourself, can I really learn how to code or do I have it in me? I'm here to tell you, yes, you can absolutely do this. Learning how to program can be scary and intimidating, but at the same time it's really fun and really exciting. In coding, as in as in life if we're going to get philosophical, the most rewarding work is usually a bit challenging, but ultimately well worth the effort. Of course, I'm able to say all this from experience, especially the cheesy parts.

The role of a sysadmin can vary a lot from company to company and even within different teams in the same company. I happen to work in the corporate identity and access management

operations team, which is a really long way of saying that we make sure that everyone is represented correctly and if they need to access certain resources, they can.

What I love the most about being a sysadmin is that the role has so many diverse functions. We handle loads of unique problems and edge cases from tinkering with different systems to collaborating with other teams. I am always learning something new, so it's really hard to get bored.

It all starts with knowing how to automate, if you're an IT support specialist, a systems administrator, or in a role somewhere in between knowing how to get computers to do the hard work for you will set you apart from others in similar it roles and make your life much easier. Think about it, would you rather manually deploy 100 computers on your own or tell your computer to do it all for you all at once?

No-brainer, right? Having a coding skill set can help you grow into more specialized roles like a systems administrator, Cloud Solutions engineer, DevOps specialist, site reliability engineer, or who knows, maybe even web developer or data analyst. The point is, being able to write a program is an essential tool in your IT toolkit and more and more employers are looking for these skills in the people they hire.

If you've ever learned a new skill, like playing a musical instrument, speaking a foreign language, knitting, or skateboarding. You know that getting good at something new requires a lot of practice. For me, I love to learn new languages and I'm proud to

say I now speak Spanish, Arabic, French, and I even know ten words in Russian.

Our world is shaped by the words and the languages we speak and while some words may be unique to one language you can always find similarities that help you learn and understand. Being able to connect the dots between cultures allows me to see things others might not, kind of sounds like this applies to IT programming, huh? My point is, whether you're learning French or Python, it 's never easy.

You have to start small, learn the basics and practice those until you master them. Only then can you move on to more complex and impressive stuff. We'll start slow, master the foundation's together and you'll soon be ready for more challenging stuff. By the end of this course, you'll understand the benefits of programming in IT roles.

You'll be able to write simple programs using Python, figure out how the building blocks of programming fit together, and combine all this knowledge to solve a complex programming problem. That's right, by the end of this course, you're going to write a program in Python that's designed to solve a real-world IT problem. Super exciting right?

We'll start off by diving into the basics of writing a computer program. You'll get hands-on experience with programming concepts through interactive exercises and real-world examples. You'll quickly start to see how computers can perform a multitude of tasks. You just have to write code that tells them what to do.

Along the way, we'll be talking about automation, which is process of getting computers to automatically do a task that us humans normally have to do by hand.

Now, some of the stuff can get a little complex and confusing. I promise to do my best to make these lessons clear and easy to understand, but if you get stuck at any point, please feel free to re-watch the videos. Practice as much as you like and take the time you really need to understand these topics. The goal of this course isn't to teach you everything there is to know about software engineering because yikes, that would be a long course.

Instead, I'm going to introduce you to some of the key concepts of programming and scripting that will empower you to spot opportunities for automation in real life. You're about to learn a skill that can help you take your career to whole new levels. Are you excited? I'm excited too, so let's jump in!

1.3. A Minute to Set Yourself up for Success

Did you know that people tends to get more out of a course like this, and are more likely to succeed when they set their intention to complete the course before they start? Use a pen and paper to complete your commitment statement below, and help yourself reach your goals. Don't worry – this is just for you and is not graded.

I commit to my goal to complete this course. I choose to complete it because...

the sentence. What motivated you to take this

When I run into obstacles or lack motivation at some point during the course, I will ...

the sentence. What would you do or say to motivate your future

(Finally, commit by writing your name here)

Make sure you save your commitment where you can see it everyday.

1.4. Welcome to the Course

This course is designed to teach you the foundations of programming in Python. We're excited to join you on this journey as you learn one of the most-in-demand job skills in IT today. In the U.S. alone, according to [Burning Glass](#) data from May 2019, there were approximately 530K job openings in 2018 asking for Python skills. This course requires no previous knowledge of programming.

1.4.1. How the 6-Week Deadline Works

I deliberately set a deadline of one week for each module for you need to complete this course. Heads up: These deadlines are there to help you organize your time, but you can take the course at your own pace. If you "miss" a deadline, you can just reset it to a new date. There's no time limit in which you have to finish the course, and you move on to the second part of the course whenever you finish.

1.4.2. Getting and Giving Help

Here are a few ways you can give and get help:

[Official Python Discussion](#) You can share information and ideas with your fellow learners in the Python discussion forums. These are also great places to find answers to questions you may have. If you're stuck on a concept, are struggling to solve a practice exercise, or you just want more information on a subject, the discussion forums are there to help you move forward.

[Contact](#) Use the Contact us link (email link) at the end of Chapter 25 to request information on specific issues. These may include error messages, challenge labs, projects, and problems with video download or playback.

1.4.3. Finding Out More Information

Throughout this course, I'll be teaching you the basics of programming and automation. I'll provide a lot of information through videos and text readings. But sometimes, you may also need to look things up on your own, now and throughout your career. Things change fast in IT, so it's critical to do your own research so you stay up-to-date on what's new. We recommend you use your favorite search engine to find additional information — it's great practice for the real world!

On top of search results, here are some good programming resources available online:

[The official Python](#) This tutorial is designed to help people teach themselves Python. While it goes in a different order than the one I am taking here, it covers a lot of the same subjects that we explore in this course. You can refer to this resource for extra information on these subjects.

The [official language](#) This is a technical reference of all Python language components. At first, this resource might be a little too complex, but as you learn how Python works and how it's built, this can be a useful reference to understand the details of these interactions.

1.5. Official Python Discussion Forums: Join, Meet & Greet

Join, meet and greet your [learner](#). You're now a part of a growing group of learners all investing in their future by learning how to program in Python.

Say hi! Write a brief introduction to help other learners get to know you better. Not sure what to write? Here's a few tips:

Hopes and Why are you taking this course? What are your expectations? What excites you most about learning to program with Python? What do you hope to achieve once you've completed this course?

Hobbies and What other things are you interested in, besides programming or IT? You might find fellow learners with the same interests!

Where are you from. There are learners from all over the globe! Why not share with others which part of the world you live in? But remember, don't share specific details like your personal mailing address.

You don't have to take part. But it's more fun if you do!

2. Introduction to Programming

2.1. The Beginning of Your Programming Journey

As the Chinese proverb says, a journey of a thousand miles begins with a single step. Today's a big day, you're taking your first step in your journey to learn how to write scripts in Python. It's going to be a little challenging at times, but really it's not that scary. We'll go slow and give you everything you need to fully grasp each concept before we move along.

In the next few sections, you'll discover the fundamental concepts of computer programming. You'll learn what a programming language is, what scripting is, what languages are out there other than Python, and how this all relates to IT. We'll also have you coding before you know it with small coding exercises we've cooked up to give you hands-on practice with Python. This will include writing your very first Python script. But always keep in mind, if at any point along the way you feel lost or confused, don't panic.

You can read any section again or watch the videos as many times as you need to let the concept sink in, plus you can ask questions in the discussion forums, which is one of the best ways to find extra information and connect with other learners. When I was asked to participate in this program, it made me think about when I first started to code. If I could give that younger version of myself a piece of advice, this is what I would tell her: it never works the first time.

Seriously, as a newbie, I expected it all to work like magic. I thought that following the rules and getting it right the first time would prove my value as a coder, but that's just not true, not even the best of the best. If you expect to write perfect code on the first shot you're going to be disappointed. You hear that younger self? Try not to feel overwhelmed by the details. Connecting the dots only comes with experience, so the best way to learn is to just jump in.

The truth is everyone learns at their own pace. If you already know some of these concepts, feel free to skip ahead to the parts that interests you the most. If you're starting from scratch, take as long as you need for each concept. The assessments will be right there waiting for you when you're done, and if at any point you start doubting yourself, remember, even the most advanced programmers started thinking, Python... what's Python?

Well, we're about to learn all about it, so let's dive in to do a rundown of what programming is.

2.2. What is Programming?

At a basic level, a computer program is a recipe of instructions that tells your computer what to do. When you write a program, you create a step by step recipe of what needs to be done to complete a task and when your computer executes the program it reads what you wrote and follows your instructions to the letter. How nice is that? The recipe is written in a code called *programming*

Programming languages are actually similar to humans spoken languages since they have a *syntax* and Now if it's been awhile since your last grammar class, here's a quick refresher on syntax and semantics.

In a human language, **syntax is the rules for how a sentence is constructed** while **semantics refers to the actual meaning of the** In English, sentences generally have both a subject, that's a person, place, or thing, and a predicate usually a verb and a statement that explains what the subject is doing.

Let's take the sentence, *Paula loves to program in Python* as an example. In this sentence, Paula is the subject and loves to program in Python is the predicate. To form a sentence that others can understand, you need to know both the syntax that constructs the sentence and the semantics that gives it meaning. The same applies to programming languages.

In a programming language like Python, the syntax is the rules for how each instruction is written and the semantics is the effects the instructions have. Much like spoken languages, there are lots of programming languages to choose from. Each has its own history, features, and applications but they all share the same fundamental ideas. So once you understand the basic concepts in one programming language, it becomes much easier to learn another.

Lastly, computers always do exactly what they're told. So when you write a program, it's important to be super clear about what you want the computer to do. Learning the syntax and semantics of the programming language you choose will allow you to do just that. Make sense? Before we continue, let's spend a moment on terminology.

2.2.1. Difference Between *Script* and *Program*

In the next few sections you'll hear the term script being used a bunch. So what's the difference between a *script* and a *Program*? The line between the two can be a bit blurry. In this course, we'll use the terms interchangeably. In general, you can think of scripts as programs with a short development cycle that can be created and deployed rapidly.

In other words, a script is a program that is short, simple, and can be written very quickly. In this course we'll focus on a specific scripting language called python which we'll use to learn the basics of programming. We'll learn about the python syntax, the rules of how to write a python program, and the semantics or meaning of the different pieces involved.

Before we start learning how to code and having you write your first python script, let's talk more about what automation is and why it's useful.

2.3. What is Automation?

Although we might not realize it, we reap the benefits of automation all the time in our daily lives. Do you ever pay your bills with scheduled payments or use a self check out at the grocery store? I always set my coffee machine to start brewing before I've even gotten out of bed. The promise of fresh coffee makes early mornings way easier.

Automation is the process of replacing a manual step with one that happens automatically. Take a traffic light for example, which continuously regulates the flow of vehicles at an intersection. A traffic light requires a human intervention only when it needs repairs or maintenance.

The automatic regulation of traffic means that humans don't have to stand at the intersection manually signaling when cars should stop or go. Instead, people can concentrate on more complex, creative, or difficult tasks like focusing on where you're driving. What's more, traffic lights don't get tired, bored, or accidentally display a green light when they meant red. This highlights another benefit of automation consistency.

Let's face it, us humans are flawed and sometimes we make mistakes, a human performing the same tasks hundreds of times will never be as consistent as a machine doing the same thing. But for all of its advantages automation isn't a solution for every situation, some tasks just aren't suited for automation.

For example, they may require a degree of creativity or flexibility that automatic systems can't provide or for more complicated or less frequently executed tasks creating the automation may actually be more effort or cost than it's worth. Think about when you get a haircut. What would it take to automate the actions of cutting hair with a machine?

The clients height, the shape of their head, their current hair length, and desired hairstyle would all need to be taken into account when designing the automatic system. We need to replicate the creativity and skills of a trained specialist along with extensive testing to ensure the clients safety and quality haircut. And if you've ever had a bad experience at a hair salon, you know quality can be subjective.

In this case, the cost and effort of automation just isn't worth the benefits of an automatic haircut would provide which is why we don't have robot hairstylists. Not too complex, right? Automation is a powerful tool when used in the right place at the right moment. It can save time, reduce errors, increase consistency, and provide a way to centralized solutions and mistakes making them easier to fix.

Throughout this course, and an upcoming ones we'll be talking about when it makes sense to apply automation and exactly how you do it. Eventually knowing when and where to use automation will become automatic for you.

2.4. Getting Computers to Work for You

Working in IT, a lot of what we do boils down to using a computer to perform a certain task. In your job you might create user accounts, configure the network, install software, backup existing data, or execute a whole range of other computer based tasks from day to day.

Back in my first IT job, I realized that every day I sat at my computer to work I typed the same three commands to authenticate into systems. Those credentials timed out everyday by design, for security reasons, so I created a script that would automatically run these commands for me every morning to avoid having to type them myself.

Funny enough, the team that monitors anomalous activity discovered my little invention and contacted me to remove it, oops! Tasks performed by a computer that need to be done multiple times with little variation are really well suited for automation, because when you automate a task you avoid the possibility of human errors, and reduce the time it takes to do it.

Imagine this scenario, your company had a booth at a recent conference and has gathered a huge list of emails from people interested in learning more about your products. You want to send these people your monthly email newsletter, but some of the people on the list are already subscribed to receive it.

So how do you make sure everyone receives your newsletter, without accidentally sending it to the same person twice? Well, you could manually check each email address one by one to make sure you only add new ones to the list, sounds boring and inefficient, right?

It could be, and it's also more error prone, you might accidentally miss new emails, or add emails that were already there, or it might get so boring you fall asleep at your desk. Even your automated coffee machine won't help you out there. So what could you do instead?

You could get the computer to do the work for you. You could write a program that checks for duplicates, and then adds each new email to the list. Your computer will do exactly as its told no matter how many emails there are in the list, so it won't get tired or make any mistakes.

Even better, once you've written the program you can use the same code in the future situations, saving you even more time, pretty cool, right? It gets better, think about when you're going to send these emails out, if you send them out manually you'll have to send the same email to everybody, personalizing the emails would be way too much manual work.

If instead you use automation to send them, you could have the name and company of each person added to the email automatically. The result? More effective emails, without you spending hours inserting names into the text. Automating tasks

allows you to focus on projects that are a better use of your time, letting computers do the boring stuff for you.

Learning how to program is the first step to being able to do this, if you want to get computers to do the work for you, you're in the right place. Earlier in this section, I told you about the first task I ever automated, now I want to tell you about the coolest thing I ever automated.

It was a script that changed a bunch of access permissions for a whole lot of my company's Internal Services. The script reversed a large directory tree with tons of different files, checked the file contents, and then updated the permissions to the services based on the conditions that I laid out in the script.

Okay, I admit I'm a total nerd, but I still think it's really cool. Next up, it's time to share your ideas. What things would you like to automate using programming? While these discussion prompts are optional, they're really fun. Seriously, they let you get to know your fellow learners a bit, and collaborate on ideas and insights.

Make sure you read what others are saying, they may give you ideas that you haven't even thought of. After that, you're ready to take your very first quiz of the course. Don't worry, it's just for practice.

2.5. Discussion Forums: Your Hopes for Automation

What everyday tasks would you like to automate through programming? Share your ideas, hopes, and goals with your [fellow](#)

2.6. Practice Quiz 1: Introduction to Coding in General - 5 Questions

From now on, you will need to open a fresh notepad or Word file in your computer to answer a quiz like this. Time Allowed: **15**

1. What's a computer program? Select only the correct response. – *1 point*

A set of languages available in the computer.

A process for getting duplicate values removed from a list.

A list of instructions that the computer has to follow to reach a goal.

A file that gets copied to all machines in the network.

2. What's the syntax of a language? Select only the correct response. – *1 point*

The rules of how to express things in that language.

The subject of a sentence.

The difference between one language and another.

The meaning of the words.

3. What's the difference between a program and a script? Select only the correct response. – *1 point*

There's not much difference, but scripts are usually simpler and shorter.

Scripts are only written in Python.

Scripts can only be used for simple tasks.

Programs are written by software engineers; scripts are written by system administrators.

4. Which of these scenarios are good candidates for automation?

Select all that apply. – 1 point

Generating a sales report, split by region and product type.

Creating your own startup company.

Helping a user who's having network troubles.

Copying a file to all computers in a company.

Interviewing a candidate for a job.

Sending personalized emails to subscribers of your website.

Investigating the root cause of a machine failing to boot.

5. What are semantics when applied to programming code and pseudocode? – 1 point

The rules for how a programming instruction is written.

The difference in number values in one instance of a script compared to another.

The effect the programming instructions have.

The end result of a programming instruction.

The correct answers are provided below. Use them to grade your performance (X out of 5) in this quiz. Always use the correct answers provided to grade your performance in each of the subsequent practice exercises in this book. **X** is the total number

of correct answers you get out of a total of 5. Don't forget to save your result!

2.6.1. Answers to Practice Quiz 1

C. At a basic level, a computer program is a recipe of instructions that tells your computer what to do.

A. In a human language, syntax is the rules for how a sentence is constructed, and in a programming language, syntax is the rules for how each instruction is written.

A. The line between a program and a script is blurry; scripts usually have a shorter development cycle. This means that scripts are shorter, simpler, and can be written very quickly.

A, D, Sending out periodic emails is a time-consuming task that can be easily automated, and you won't have to worry about forgetting to do it on a regular basis.

C. Like human language, the intended meaning or effect of words, or in this case instructions, are referred to as semantics.

3. Setting up Your Python & Programming Environment

3.1. What is Python?

Welcome back. How did you do on your first quiz? If you got most of the questions right, great job. If not, no worries it's all part of learning. I'll be here to help you check that you've really got your head around these concepts with regular quizzes like this.

If you ever find a question tricky, go back and review the videos and then try the quiz again. You want to feel super comfortable with what you've learned before jumping into the next lesson. Remember, take your time. Whenever you're ready just move on.

Okay. Feeling good? Great. Let us dive in.

In this course, we will use the Python programming language to demonstrate basic programming concepts and how to apply them to writing scripts. We have mentioned that there are a bunch of programming languages out there. So why pick Python?

Well, we chose Python for a few reasons. First off, programming in Python usually feels similar to using a human language. This is because Python makes it easy to express what we want to do with syntax that's easy to read and write.

Check out this example.

```
1 friends = ["Taylor", "Alex", "Pat", "Eli"]
```

```
2 for friend in friends:  
3     print("Hi " + friend)
```

There is a lot to unpack here so don't worry if you don't understand it right away, we'll get into the nitty-gritty details later in the course. But even if you've never seen a line of code before, you might be able to guess what this code does. Each line of your code may be numbered 1, 2, 3, etc by the programming editor (or IDE) you use, as shown in the example above, but it can be removed in many editors.

This code defines a list with names of friends and then creates a greeting for each name in the list. Now it is your turn to make friends with Python. You will try something like this out pretty soon and see what happens.

3.1.1. How to Execute Python Codes

Python code can be written on the Python interpreter console. But to write a Python code on the interpreter console, we first need to install Python on our computer. One thing I do not like about the console is there is no direct way or a single command to clear it.

Type the following few lines of code to clear your console screen (for Make sure you press the Enter or Return key at the end of each line.

```
import os
os.system('cls')
```

Alternatively, you can use this script:

```
import os
def clear():
    os.system('cls')
clear()
```

Use the following few lines of code to clear your console screen (for Make sure you press the Enter or Return key at the end of each line.

```
import os
os.system('clear')
```

Alternatively, you can use this script:

```
import os
def clear():
    os.system('clear')
clear()
```

You can also use a text editor, assuming you have installed Python, to write and execute your Python codes. Editors are more colorful and user-friendly. Throughout this course, you will execute Python code using either the Python interpreter for your operating system type (Windows, Linux or Mac) or an IDE (integrated development Environment) such as **Pycharm** (use [this link to download](#)) We will use the Python interpreter more frequently in this course.

Here's a list of some popular editors and open source applications we can use to write our codes.

Ordinary text editor such as a notepad, or [Sublime Text](#) (runs locally on your computer).

Python interpreter (use this [download](#) which you get by installing Python locally on your computer.

This also runs locally on your computer. It can number every line of your code.

Command window (also called

Online interpreters such as

Jupyter open-source web application for creating and sharing documents containing live code, visualizations, equations, narrative text, etc. You will use this later for your projects.

Watch the following video to learn how to download and install Pycharm, a popular IDE among Python developers. If you have not downloaded all the tutorials videos in this course, go to Chapter 25 right now. Make sure you save all your videos where you can easily find them.

Video 1 (8:38 *How to download and Install Python and Pycharm (community version)*)

Now, we'll start with some small coding exercises using code blocks. Later on as you develop your skills, you'll work on larger more complex coding exercises using other tools. Getting good at something takes a whole lot of practice and programming and Python is no different. I recommend that you practice every example we share in this course on your own. You can practice using an [online Python](#)

Now I am sure you are wondering what the heck is a Python interpreter. In programming, an interpreter is the program that *reads* and *executes* code. Remember how we said a computer program is like a recipe with step-by-step instructions?

Well, if your recipe is written in Python, the Python interpreter is the program that reads what is in the recipe and translates it into instructions for your computer to follow. Eventually, you'll want to

install Python on your computer so you can run it locally and experiment with it as much as you like.

You can practice with the quizzes I provide and with the online interpreters and code pads that we'll give you links to in the next section. We'll provide a whole bunch of exercises for you but feel free to come up with your own and share them in the discussion forums. Feel free to get creative. Afterall, this is your chance to show off your new skills!

3.2. A Note on Syntax and Code Blocks

When writing code, using correct syntax is super important. Even a small typo, like a missing parentheses or an extra comma, can cause a syntax error and the code won't execute at all. Yikes. If your code results in an error or an exception, pay close attention to syntax and watch out for minor mistakes.

If your syntax is correct, but the script has unexpected behavior or output, this may be due to a semantic problem. Remember that syntax is the rules of how code is constructed, while semantics are the overall effect the code has. It is possible to have syntactically correct code that runs successfully, but doesn't do what we want it to do.

When working with the code blocks in exercises for this course, be mindful of syntax errors, along with the overall result of your code. Just because you fixed a syntax error doesn't mean that the code will have the desired effect when it runs! Once you've fixed an error in your code, don't forget to retry it to check your work again.

3.3. Why is Python Relevant to IT?

Remember how we mentioned that Python is simple and easy to use? Python makes it easy to express the fundamental concepts of programming like data structures and algorithms with easy to read syntax. This makes Python a great language to use to learn programming. And there are other reasons to pick Python, too.

Python is super popular in the IT industry, making it one of the most common programming languages used today. Python isn't new. Its **first version** was released by **Guido van Rossum** back in 1990. Since then, the community that develops it has grown and the language has advanced a lot. Whenever there's a significant change to the semantics or syntax of the language, a new major version is released.

In 2000, Python 2 was released. In 2008, we got Python 3. In this course, we'll use the latest Python 3.9.6, which came out in June 2021. For many years, Python was considered a beginner's language and was mostly used for teaching concepts or writing very small simple scripts, like in this course. But in recent years, the adoption of Python has grown dramatically.

One reason for this is that the language has become more powerful. It's also because there's more tools available in Python for a growing range of applications. You can use Python to calculate statistics, run your e-commerce site, process images, interact with web services, and do a whole host of other tasks.

Python is perfect for automation. It lets you automate everyday tasks by writing simple scripts that are easy to understand and easy to maintain. That's why Python is the language of choice for lots of people working in IT support, system administration, and web development. Not only that, but it's also used in fast-growing areas of IT, like **machine learning** or **data**

Last but not least, Python is available for download on a wide variety of operating systems, like Windows, Linux, and Mac OS. And what's more, Python is so popular in the workplace that if you are currently working in IT, you've most likely encountered it already. And if you're planning a career in IT, chances are you'll interact with Python quite a bit. So there's a whole lot of reasons for why Python is relevant to today's IT industry.

A large part of programming is learning through trial and error and asking questions. So if at any point you get stuck, don't get discouraged. Making mistakes helps you improve. The more you see failure or broken code as an opportunity to learn, the quicker you'll master programming.

I remember the first Python script I ever wrote. It took a lot of refactoring, debugging, and testing to get it to work. I relied on a lot of my teammates for help and mentorship and wound up spending more time on Stack Overflow than actually writing the code. Thankfully, you don't have to reinvent the wheel.

There's almost always someone on the Internet who's trying to do what you're doing and can help point you in the right direction when you're stuck. Sometimes it takes a village. It's really

important to keep in mind that even experienced programmers may need to ask a colleague a question from time to time or look something up on the Internet.

Whether you're a programming novice or have some experience in software development, remember, the best programmers overcome challenges by seeking help or using other resources. Once you've completed this program, you'll be well on your way to confidently programming in basic Python.

There's lots of information online that will help you continue to develop your programming skills. For example, there are lots of online courses for specific programming languages. You'll find answers to your Python coding questions in the [official Python](#)

You can use sites like Stack Overflow to discuss and share with other developers. And you can ask questions in the discussion forums mentioned in the previous chapter. You can even subscribe to some of the Python mailing lists to keep in the know on the latest updates to the language.

You're opening the door to the whole world of programming, and it's super exciting to be joining the development community. The most important thing to remember is that you're never alone. Any questions you may have, any time in your career, there are resources out there to help you find the answers you need.

Wow, that was a lot of information. Feel free to take a quick break, grab something to drink, and then head on over to section

3.4 to learn more about Python and the resources out there to help you learn.

3.4. How to Become a Pythoneer or Pythonista

You should practice using Python on your own. Practicing on your own as much as you can is the best way to become a Pythoneer or Pythonista. Open the “Additional Resources” folder you downloaded earlier. You will find useful files, such as *How to Learn More 40+ Project Ideas for Beginners, Intermediate and Advanced* and so much more!

In this file, you will find links to additional resources that can help you grow your Python language.

3.5. Other Languages

Although we picked Python for this course, it's important to note that it's just one of the many coding languages out there. Think of a given programming language as just one of the many powerful tools in your IT toolbox. Each language has its unique set of pros and cons. Some run faster than others. Some are better suited for enterprise applications. Others are particularly good at crunching numbers.

There are platform-specific scripting languages like *PowerShell* which is used on Windows, and *Bash* which is used on Linux. Both are widely used by system administrators on those platforms. There are also general-purpose scripting languages similar to Python, like *Pearl* or which are also widely used for scripting and automation.

which was originally developed as a client-side scripting language for the web is increasingly used as a server-side language for a broader set of tasks. And the list doesn't stop there.

There's a vast array of traditional languages to explore like C, C++, Java, or Go. As you progress in your career in IT, you'll probably encounter a number of different languages and learn when to use each of them. But let's not get ahead of ourselves. First, we have Python to get our heads around.

A nice feature of learning the basics of programming in one language is that you can generally apply the same concepts you

learn to other languages. This means that once you're familiar with Python, you'll find it easier to pick up new coding languages as you'll spot and understand similarities and differences between them.

After all, every language needs to do some common things like create variables, control the flow of a program, read input, and display output, even if they do these tasks using different approaches. As we called out earlier, learning a programming language is somewhat similar to learning a foreign language. You'll need to grasp the syntax and semantics for that language.

Luckily for us, once you know the fundamentals of programming, learning another language is much easier than learning a second foreign language. There are a lot more similarities between programming languages than differences.

To explore some of the similarities and differences between various scripting languages, let's take a look at a simple program that prints the words **World!** ten times in three different languages, Python, Bash and PowerShell. See Fig. 3.5.1.

Python:

```
for i in range(10):  
    print("Hello, World!")
```

Bash:

```
for i in {1..10}; do  
    echo Hello, World!  
done
```

PowerShell:

```
for ($i=1; $i -le 10; $i++) {  
    Write-Host "Hello, World!"  
}
```

Fig. 3.5.1: A program that prints the "hello, world" ten times in three different languages: Python, Bash and PowerShell

As you can see, each language uses a different approach to printing hello world. But look closer and you'll see similarities too. Each language must somehow put text onto the screen. The command for Python is **print** (p must be in lower case). For Bash it's **echo** and for PowerShell it's **Write-Host**

Also notice that each language has to count to ten in some way. While Python does this by specifying `range(10)` Bash uses a sequence notation to count from 1 to 10. PowerShell has the most complex

syntax in this example, but it also boils down to starting at 1 and counting up to 10.

So as we've just seen there's a whole lot of programming languages out there, but don't let that scare you. In this course, you will only need to focus on learning Python. Once you can speak Python you can go on to learn any other language you want. Up next, we've got another quiz to help you practice what you've just learned.

3.6. Practice Quiz 2: Introduction to Python - 5 Questions

1. Fill in the correct Python command to put "My first Python program" onto the screen. - 1 point

1 print(_)

2. Python is an example of what type of programming language? - 1 point

Client-side scripting language

Machine language

Platform-specific scripting language

General purpose scripting language

3. Convert this Bash command into Python: - 1 point

echo Have a nice day.

4. Fill in the correct Python commands to put "This is fun!" without the quotes onto the screen 5 times. - 1 point

1 for i in range(_):

2 print("_")

5. Write the Python code snippet that corresponds to the following Javascript snippet: - 1 point

```
1 for (let i = 0; i < 10; i++) {
```

```
2   console.log(i);
```

```
3 }
```

3.6.1. Answers to Practice Quiz 2

1. `print("My first Python program")`

2.

3. `print("Have a nice day.")`

4.

1 `for i in range(5):`

2 `print("This is fun!")`

5.

1 `for i in range(10):`

2 `print(i)`

4. Hello, World!

4.1. How to Write Hello World in Python

Now that you've got an idea of what Python code looks like, let's check out one of the most basic examples and dive deeper into what's going on. Get ready. We're going to use the Python interpreter to make our computer say hello to the world.

Video 2 (1:04 *Hello World program in Python*)

Like the statement `hello world` for example, the **print** function is part of the basic Python language. Whenever we use keywords or functions that are part of the language, we're using the programming language's syntax to tell the computer what to do. So, what are *functions* and

Functions are pieces of code that perform a unit of work. We'll talk a lot more about functions later on, and you'll even learn how to write your own. **Keywords** are reserved words that are used to construct instructions. These words are the core part of the language and can only be used in specific ways.

Some examples include `and` and `We'll` explain all of those and a bunch more later in the course. As we called out, the keywords and functions used in Python are what makes up the syntax of the language. Once we understand how they work, we can use them to construct more complex expressions that get the computer to do what we want it to do.

Last off, notice how “Hello, world” is written between double quotation marks. Wrapping text in quotation marks indicates that the text is considered a string, which means it's text that will be manipulated by our script. In programming, any text that isn't inside quotation marks is considered part of the code.

Now, for a bit of trivia, do you know why we printed the “Hello, world” in our example? Well, printing hello world has been the traditional way to start learning a programming language since way back in the '70s when it was used as the first example in a famous programming book called the C programming language.

In Python, the hello world example is just one line, in C, it's three lines, in other languages, it can be even more. While learning to write hello world won't teach you the whole language, it gives you a first impression of how functions are used, and how a program written in that language looks.

4.1.1. Program Comments (#)

Comments are an extremely useful feature in most programming languages. Everything you've written in your programs so far is Python code. As your programs become longer and more complicated, you should add notes within your programs that describe your overall approach to the problem you're solving. A comment allows you to write notes in English within your programs.

4.1.2. How to Write Comments

In Python, the hash mark indicates a comment. Anything following a hash mark in your code is ignored by the Python interpreter.

For example:

```
1 # This is the first line of my program.
```

```
2 print("Hello world!") # This is the second line of my program.
```

```
Hello world!
```

As you can see, the result of our program is only “Hello world!”. Python ignores the first line completely and executes only the part of the second line that does *not* follow the # mark.

All right, now that we've written our first piece of Python code, I think you're ready for something a bit more challenging than hello world. Ready? Let's do it!

4.2. How to Get Information from the User

On the whole, for a program to be useful it needs to get at least some information from the user. With this data, the program can take actions that are relevant to the user, instead of generic actions, like printing hello world.

Data can be provided to a computer in a bunch of different ways. For example, on a website you might input data by entering text into text fields or clicking links. If you're using a mobile application, maybe you'll click on buttons or select preferences from a drop-down menu.

In a command line program, you might provide additional data by passing strings this parameters to the program, or you could have the program ask you for data interactively. All of these various platforms, programs, and apps process data differently.

Some might take the contents of a file as data to be processed, others gather data from other sources and process it in the background. Remember our earlier example, when we automated the process of identifying and removing duplicate emails?

There, the data provided to the program was the list of emails, which would usually be given in a file that lists the emails one per line. Whichever way your application gets the data, it will need to come from somewhere.

For our first examples in this course, we'll just have the data as its own line in our block of code. This is limited, but straightforward. Later in this course, and in upcoming courses, we'll introduce you to better ways of feeding data into your code. For now though, let's see this idea in action in a very simple example in the next video.

Video 3 (0:28 *How to Get Information from the User*)

Next up, we'll learn a few other easy things that you can get Python to do for you.

4.3. Python Can Be Your Calculator

There's a ton of things that you can do with Python and you'll learn many of them in this course. But before we dive into complex subjects, let's have some fun with another simple task that you can do with Python. We are going to make Python our calculator. Watch this video:

Video 4 (0:57 *Python can be your calculator*)

Here's more examples:

```
print(10+5)
```

15

```
print(-1/4)
```

-0.25

Easy. Repeating or periodic numbers are printed in a longer format. Let's try 1 divided by 3.

```
print(1/3)
```

0.3333333333333333

In math theory, when 1 is divided by 3, the digit 3 repeats forever after the decimal point. Of course, it's hard to display something that repeats for ever. So instead, we have a representation showing lots of decimal places. Not too hard, right?

If you're starting to worry that this is becoming an algebra course, relax. We're not going to do anything more complex than what we've just seen. If you're thinking, "Why would I use Python instead of just a normal calculator?" That's a valid question.

Experimenting in this way, you get familiar with the language's math capabilities. In IT jobs, there are many tasks that require you to use math calculations. You might need to count how many times a certain word appears in a text, or work out the average time it takes for an operation to complete, or how much you have to compress an image to fit in certain size constraints.

Whatever you need to calculate, writing a script can help you do it faster and with more accuracy. So you need to know what mathematical operations are available to you. Python actually has a lot more advanced numeric capabilities that are used for data analysis, statistics, machine learning, and other scientific applications. We won't get into these in this course. But if you want to learn more about them on your own, there's a wealth of online resources available.

4.4. Cheat Sheet 1: First Programming Concepts

Now open and study the “First Programming Concepts Cheat Sheet” you downloaded to help you with programming concepts that we've just covered. After that, it's time for another quiz. This time with a few small coding exercises. Remember, if something is unclear, you can re-watch the videos as many times as you need. Ready? You've got this.

4.5. Practice Quiz 3: Hello World - 5 Questions

1. What are functions in Python? - 1 point

Functions let us to use Python as a calculator.

Functions are pieces of code that perform a unit of work.

Functions are only used to print messages to the screen.

Functions are how we tell if our program is functioning or not.

2. What are keywords in Python? - 1 point

Keywords are reserved words that are used to construct instructions.

Keywords are used to calculate mathematical operations.

Keywords are used to print messages like "Hello World!" to the screen.

Keywords are the words that we need to memorize to program in Python.

3. What does the print function do in Python? - 1 point

The print function generates PDFs and sends it to the nearest printer.

The print function stores values provided by the user.

The print function outputs messages to the screen

The print function calculates mathematical operations.

4. Output a message that says "Programming in Python is fun!" to the screen. - 1 *point*

5. Replace the ___ placeholder and calculate the ratio: - 1 *point*

Tip: to calculate the square root of a number x , you can use $x^{**}(1/2)$.

```
ratio = _
```

```
print(ratio)
```


4.5.1. Answers to Practice Quiz 3

1. Python functions encapsulate a certain action, like outputting a message to the screen in the case of `print()`.

2. A. Using the reserved words provided by the language we can construct complex instructions that will make our scripts.

3. C. Using the `print()` we can generate output for the user of our programs.

4. `print("Programming in Python is fun!")`

5.

```
1 ratio = (1+(5**0.5))/2
```

```
2 print(ratio)
```

Do you now see how we can use Python to calculate complex values?

5. Module Review

5.1. First Steps Wrap Up

Congrats! You made it to the end of the first module. Great job. You've taken the first steps to learning a new programming language, and growing your IT skillset. Getting there shows real determination and a will to learn.

We've covered a lot of topics, and many might be new to you if you've never learned about programming before. You've discovered what scripting is, what the syntax and semantics of a programming language are all about, and how they relate to automation.

We've got to grip some small blocks of Python code, talked about why Python is relevant to IT, and explored what other programming languages are available. We've had our first approach to how to input data, and write a script that puts this data to use, and we've seen how you can use Python to perform typical math calculations. Not bad for your first Python steps, right? This is just the beginning of an exciting journey, learning to code, and we hope you're eager to learn more.

Coming up, get ready for your **first graded** These assessments help you check whether you've understood all the concepts and that you're ready to move on to the next stage. Now, don't worry. If at any point you're not sure about a question, you can always review the videos, cheat sheets and any section of this book to remind yourself of the answer.

Remember, that everybody learns at different speeds. So take your time, really get familiar with the concepts. Once you feel ready, the assessment is right below waiting for you. I'll get back to you once you've nailed it.

5.2. Module 1 Graded Assessment - 10 Questions

It's time for your first graded assessment. Open the "Graded Assessments" folder you downloaded earlier. It contains the pdf files of all the graded assessments in this course. Here's the file name to search:

Module 1 Graded Assessment – *file name*

5.2.1. Solutions to Module 1 Graded Assessment

I (or any member of my team) is available to help you grade your assessments. You can use my help link (email) at the end of chapter 25 to send your assessment for grading. We will get back to you in 12 to 24 hours with your result.

However, if you cannot wait, you can open the “Graded Assessments” folder you downloaded earlier. It contains the pdf formats of the solutions to all graded assessments in this course. You can use them to grade your assessments by yourself. Just be honest as you grade. Here’s the file name to search:

Module 1 Graded Assessment Solutions – *file name*

Module 2

Programming is like clothing. It is an art, a way to express yourself artistically in this world.

“You might not think that programmers are artists, but programming is an extremely creative profession. It’s logic-based creativity” – John Romero

6. Expressions and Variables

6.1. Basic Python Syntax introduction

Welcome back, and well done for completing your first graded assessment. You're doing a great job making it this far. Chances are some topics we've covered may have been a little tricky at times, especially if you're completely new to programming.

Don't worry if something wasn't obvious right away. We went through a lot of new concepts and it might take several passes until you feel comfortable with them. And that's totally normal. We all went through when we were learning how to code.

In the previous module we explored some basic concepts, like programming and automation. We called out that each programming language has a specific syntax, which we need to learn so we can tell the computer what to do. We then got a sneak preview of some of the things we could do with Python.

Up next, we'll dive deeper into some basic building blocks of Python syntax, things like variables, expressions, functions, and conditional blocks. At first glance these pieces may seem pretty simple, but when we start to combine them they become a lot more powerful.

Understanding of programming languages syntax isn't too different from learning a spoken language. For example, the best way to learn Spanish is to visit a Spanish speaking country, immerse yourself in the culture, listen to the people. Then figure out how

to arrange the words to form a sentence that another speaker can understand.

The same is true for programming. When you immerse yourself in Python programming you'll learn how to formulate statements of code that the computer can understand. This is called syntax.

Okay, so as you go through the next few sections keep in mind that our main goal is to learn the language's syntax. So we'll focus on how to tell our computer what to do, not on how to get it to do complicated tasks. Like before, we'll run through some simple exercises to help you see the concepts in action.

And as you pick up the new skills and get to grips with different tools we'll start to write more advanced scripts that tackle more challenging problems. Again, if at any point you feel confused or that something just isn't clear, remember you can watch the videos and take the practice quizzes as many times as you need.

The key to getting good at programming is practice, practice and practice. You have to keep working your programming muscles in order to get strong, just like building muscles in the gym. Train hard, train regularly, and you'll be tackling more weighty coding problems in no time.

All right, ready to jump back in? In the next section we're going to learn all about data types. Let's get started.

6.2. Data Types

In earlier videos, we called out that text written between quotes in Python is called a string. In programming terminology, a string is known as a data type, whether it's a mobile game or a script used to automatically create user accounts. Most programs need to manipulate some kind of data, and that data can come in a lot of different forms, or like we call them data types.

A string is only one kind of data type found in Python. There's a bunch of others, like an **integer** which represents whole numbers without a fraction, like 1. We also have **float** which represents real numbers or in other words, a number with a fractional part like 2.5. Other examples are **dict** (for dictionary), **complex** (for complex numbers), **bool** (for boolean), etc.

In Python programming, data type is an important concept. We can use variables (see section 6.4) to store data of different types, and different data types can do different things. The following data types and their categories are built in by default in Python:

Python:	Python:	Python:	Python:	Python:	Python:
Python:		Python:	Python:		Python:
		Python:	Python:		

Python:	Python:	Python:	Python:
---------	---------	---------	---------

Generally, your computer doesn't know how to mix different data types. Watch the next video to understand data type better:

Video 5 (02:04 *Data Types*)

Adding two integers together makes perfect sense to computers.

```
print(2+3)
```

5

Adding together two strings also makes sense. We just end up with the longer strings that contains the two.

```
print("Python programming " + "is fun")
```

Python programming is fun

But your computer doesn't know how to add an integer and a string. If you tell it to mix these two different data types, your computer isn't going to know what to do and will raise an error as shown in **Video**

Errors are a common part of programming, and you'll probably have to deal with them a lot. The trick is to think of errors as little clues from your computer to help you improve your programming skills. Read the errors carefully, understand what

they're telling you, and then use that new knowledge to help you fix the mistake.

In the example shown in Video 5, the last line of the error message in the Python interpreter shows us that we've encountered something called a *TypeError*. When we get a bit of explanatory texts, that tells us that the plus sign can't be used between an *int* type and a *str* type, which are short names for integer and string. I tried this code in Pycharm and got this error which is similar.

```
1 print(7 + "8")
```

```
File "C:\user\Documents\python_programs.py", line 1
```

```
    print(7 + "8")
```

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

```
Process finished with exit code 1
```

Thinking about what we've already learned about strings, integers, and mixing data types, can you guess what the error is trying to tell us? The message "unsupported operand type(s)" in Pycharm tells us that we can't add the integer **7** and the string **"8"** because they're different data types. But what if you didn't have an instructor to helpfully point that out?

How would you know? You'd need to use your research skills and the resources we called out earlier in the course to do some investigating. For example, you could look for information about the error by pasting the `TypeError` message into the search bar of your favorite search engine.

This is a common trick used by almost everyone learning to code, and even by experienced developers. You'll usually find that other people on the Internet have reported similar errors and solved them too.

Back to our example. Maybe you're thinking, aren't we adding two numbers 7 and 8 here? Looks a bit like it. Well, look carefully and remember that anything wrapped in **quotation marks** is considered a string in Python.

So, "8" is a string here, while 7 is an integer. To the computer, adding 7 plus "8" is just as strange as adding 7 plus A is to us, and seven plus A equals no sense at all. It might be helpful to think about data types in terms of information they can represent.

For example, the name of a file would be represented as a string data types, while the size of that file might be an integer data type. If you're ever not 100 percent sure what data types a certain value is, Python gives you a handy way to find out. You can use the "type" function, to have the computer tell you the type. This might come in handy when dealing with code that someone else wrote and you're not sure what data types it's using.

For example,

```
print(type("8"))
```

```
'str'>
```

This tells us that “8” belongs to *str* class, which like we said earlier is short for string.

```
print(type(2))
```

```
'int'>
```

The number 2, belongs to the *int* class, which is short for integer. We'll talk more about what we mean by “class” later in the course. For now, you can just use it as a synonym for data type. So now you know three very common data types in Python.

There are plenty of others you'll be using soon, but don't worry about them at the moment. As we continue through the course, we'll come across more data types and learn how to interact with each of them.

For now, just remember, mixing your data types will get your computer, well, all mixed up. So keep your strings with your strings, your integers with your integers, and your floats with your floats, and you shouldn't get in too much of a tangle.

6.3. Data Types Recap

In Python, text in between quotes - either single or double quotes - is a string data type. An integer is a whole number, without a fraction, while a float is a real number that can contain a fractional part. For example, 1, 7, 342 are all integers, while 5.3, 3.14159 and 6.0 are all floats. When attempting to mix incompatible data types, you may encounter a You can always check the data type of something using the *type()* function.

6.4. Variables

When we ask a computer to perform an operation for us, we usually need to store values and give them names so that we can refer to them later. This is where variables come in handy.

Variables are names that we give to certain values in our programs. Those values can be of any data type; numbers, strings or even the results of operations.

We already used variables in some of our initial examples like using them to store a name or a value. Now we are going to learn exactly how they work and how to make the most of them. Think of variables as containers for data.

When you create a variable in your code, your computer reserves a chunk of its own memory to store that value. This lets the computer access the variable later to read or modify the value. You can see this in action in this video:

Video 6 (01:47 *Variables*)

Here's another example. Imagine a simple script that calculates the area of a triangle using the formula

$$\text{Area} = (\text{base} \times \text{height}) / 2$$

Area, base and height can all be represented by variables as shown in this script:

```
base = 5
height = 3
area = (base*height)/2
print(area)
```

7.5

In the above script we are creating three variables and storing different values in each. The process of storing a value inside a variable is called Here we assign the base variable the value of 5. We assign the width variable the value of 3, and the area variable with the result of the expression (base x height) / 2.

An **expression** is a combination of numbers, symbols or other variables that produce a result when evaluated. In this example, we are multiplying the value of two variables and the dividing the result by 2 to arrive at the value that we want. Finally, we use our old friend the **print** function to display the value of the area on the screen.

All right. We have just seen how to assign values to variables, use expressions to calculate more complex values and then print the contents of a variable. Variables are important in programming because they let you perform operations on data that may change.

For example, if we extended our triangle script to accept any input as the value of the base and height variables, we could calculate the area of a triangle of any size.

To give a more IT focused example, say we have a script that performs a specific operation on a file. We can extend that script to perform the same operation on any file but only if the program used a variable to store the file name.

You might have noticed that we assign a value to a variable by using the = sign in the form of

variable =

Generally, you can name variables whatever you like but there are some restrictions. Using these **reserved terms** will make your program confusing to read and will result in errors.

6.4.1. Variable Name Restrictions

Don't use Keywords or functions that Python reserves for its own use, like

Don't use spaces

You must start with a letter or underscore (_)

Variable names must be made up of only letters, numbers and underscores (_)

Let's check out some examples of valid and invalid variable names to understand this better:

I_am_a_variable is a **valid** variable name.

I_am_a_variable2 is also a **valid** variable name.

1_is_a_number is an **invalid** variable name because variable names must start with a letter or underscore.

apples_&_oranges is **invalid** because it uses the special character & (ampersand).

Last thing, remember that precision is important when programming. Python variables are case sensitive, so capitalization matters. Lowercase name, uppercase name and all caps name are all valid and *different* variable names, and that **rule on variables is**

6.5. Expressions, Numbers and Type Conversions

Earlier, we saw how we can't use the + operator between an integer and a string because they're different data types. But what happens when we try to operate with an integer and a float instead, let's find out in this video:

Video 7 (01:26 *Expressions Numbers and Type Conversions*)

```
Print(7+8.5)
```

```
15.5
```

Error-free! Python has no problem performing this operation. But what's up with that, aren't integer and a float two different data types? They sure are but there's a lot happening under the hood here. Behind the scenes the computer is busy automatically converting our integer seven into a float 7.0.

This lets Python then add together the values to return results that is also a float. We call this process, **implicit** The interpreter automatically converts one data type into another. We've called this out before, but it's worth highlighting again that Python operations aren't just restricted to numbers. You can also use the plus operator to add together strings. This lets you do things like create sentences from individual words.

Just don't forget to add spaces to each words. Otherwise, the computer will run them all together.

```
print("This" + "is" + "not" + "neat") # No space afer each word
```

Thisisnotneat

```
print("This " + "is " + "pretty " + "neat!") # Space added after  
each word except the last
```

This is pretty neat!

So what if you really want to combine a string and a number, is it possible? It sure is, but only with an **explicit** In Python, to convert between one data type and another, we call a function with the name of the type we're converting to. Let's see how this works (try writing this script in your own Python interpreter or IDE).

```
base = 6  
height = 3  
area = (base*height)/2  
print("The area of the triangle is: " + str(area))
```

Now, things are getting a little bit more complex. Let's take a moment to unpack this to make sure it all makes sense. In this script, we're first calculating the area of a triangle, and when printing it we're adding it to a string. To do this, we need to call

the **str()** function to convert a number into a string. Let's execute it and check out what happens:

The area of the triangle is: 9.0

Our number got converted to a string and print it together with the message. We've learned a little bit about variables, values, expressions, and conversions. Once we finish section 6.5, we've got a practice quiz to help you solidify your knowledge. As always, take your time and review the content if you need. You've totally got this.

6.6. Implicit versus Explicit Conversion

As we saw earlier, some data types can be mixed and matched due to implicit conversion. Implicit conversion is where the interpreter helps us out and automatically converts one data type into another, without having to explicitly tell it to do so.

By contrast, explicit conversion is where we manually convert from one data type to another by calling the relevant function for the data type we want to convert to. We used this in our previous example when we wanted to print a number alongside some text.

Before we could do that, we needed to call the *str()* function to convert the number into a string. Once the number was explicitly converted to a string, we could join it with the rest of our textual string and print the result.

6.7. Practice Quiz 4: 5 Questions

1. In this scenario, two friends are eating dinner at a restaurant. The bill comes in the amount of 47.28 dollars. The friends decide to split the bill evenly between them, after adding 15% tip for the service. Calculate the tip by filling the blanks (_), the total amount to pay, and each friend's share, then output a message saying "Each person needs to pay: " followed by the resulting number. - 1 point

```
1 bill = 47.28
```

```
2 tip = bill _ 0.15
```

```
3 total = bill + _
```

```
4 share = total/2
```

```
5 print("Each person needs to pay: " + _(share))
```

2. This code is supposed to take two numbers, divide one by another so that the result is equal to 1, and display the result on the screen. Unfortunately, there is an error in the code. Find the error and fix it, so that the output is correct. - 1 point

```
1 numerator = 10
```

```
2 denominator = 10
```

```
3 result = numerator / denominator
```

```
4 print("result")
```

3. Combine the variables to display the sentence "How do you like Python so far?" - *1 point*

```
word1 = "How"
```

```
word2 = "do"
```

```
word3 = "you"
```

```
word4 = "like"
```

```
word5 = "Python"
```

```
word6 = "so"
```

```
word7 = "far?"
```

4. This code is supposed to display "2 + 2 = 4" on the screen, but there is an error. Find the error in the code and fix it, so that the output is correct. - *1 point*

```
print("2 + 2 = " + (2 + 2))
```

5. What do you call a combination of numbers, symbols, or other values that produce a result when evaluated? – *1 point*

An explicit conversion

An expression

A variable

An implicit conversion

6.7.1. Answers to Practice Quiz 4

1.

```
bill = 47.28
```

```
tip = bill * 0.15
```

```
total = bill + tip
```

```
share = total/2
```

```
print("Each person needs to pay: " + str(share))
```

2.

```
1 numerator = 10
```

```
2 denominator = 10
```

```
3 result = numerator / denominator
```

```
4 print(result)
```

3. `print(word1 + " " + word2 + " " + word3 + " " + word4 + " " + word5 + " " + word6 + " " + word7)`

4. `print("2 + 2 = " + str(2 + 2))`

5. B.

7. Functions

7.1. Defining Functions

You made it through another quiz. You are doing awesomely, keep it up! So far we've been looking at variables, expressions and operations which are the smallest components of scripts. Up next, we're going to look at functions which are another crucial programming building block.

We've come across a few Python functions in our examples so far: the *print()* function that writes text on the screen, the *type()* function which tells us the type of a certain value and the *str()* function which converts a number into a string.

All those functions come as a part of the language, and we'll look into a bunch of other built-in Python functions throughout this course. But now we're going to see how to **define our own functions** to tell the computer to do things that the language's built-in functions don't.

Video 8 (02:13 *Defining Functions*)

Let's start with a simple example. In the following piece of code, we're defining a function:

```
def greeting(name):  
    print("Welcome, " + name)
```

Our function takes the **parameter** “name” and prints a greeting for that name. This snippet is small but it already shows a lot of important points about how we define functions in Python. Let's go through this step-by-step.

To define a function, we use the **def** keyword. The name of the function is what comes *after* the keyword. In this example, the function's name is “greeting”. So to call the function later in the script, we'll use the word `greeting`. After the name, we have the parameter of the function which is written between parentheses `()`.

In this example, we only have one parameter, `name`, followed by a `:` (colon) at the end of the line. After the colon, we have the **body** of the function. That's where we state what we want our function to do. Note how the body is **indented** to the right. This is a key characteristic of Python and we'll come across to the bunch.

For now, just keep in mind that the body of the function must be to the *right* of the definition. In this example, the body contains just one line that calls the `print` function. Looks simple, right? But creating functions can actually be super powerful.

The body of a function can have as many lines as we want it to and do all sorts of fun stuff. We'll find out exactly what in later sections. But for now, let's execute our function twice and see what happens. Type this in your editor (don't copy and paste in the editor):


```
greeting("kay") # First example.  
Welcome, kay
```

```
greeting("Cameron") # Second example.  
Welcome, Cameron
```

That's nice. But it's not too interesting yet. Let's make it do a little more as shown in this example:

```
def greeting(name, department):  
    print("Welcome, " + name) # This is indented 2 spaces  
    print("You are part of " + department) # This is also indented  
2 spaces
```

Our function now receives **two parameters** instead of one, *name* and *department* and it writes two separate messages. Again, notice the indentation. We can add as many lines as we'd like to the body of the function but each line must be indented the same number of spaces to the right. In this example, we're using 2 spaces. We could use 4 or 8 or, any other number as long as they're all consistent.

Let's try calling our new and improved greeting function.

```
greeting("Blake", "IT support") # We are calling our function here.  
Welcome, Blake  
You are part of IT support
```

```
greeting("Ellis", "Software Engineering") # We are calling our  
function again.
```

```
Welcome, Ellis
```

```
You are part of Software Engineering
```

Nice results. That's more useful, and we're only just scratching the surface of what we can do with functions. Remember that these are just simple examples but a function can do a lot more than just print messages. In this course and throughout the upcoming courses, we'll explore a bunch of other tasks that we can do with Python and usually we'll write them inside functions.

How are you feeling so far? These new concepts are coming fast and furious now. Are you starting to get to grips with it all? If so, awesome, and if some things are still a little fuzzy, now is a great moment to go back and review everything we've covered up till now. Once you're feeling good, meet me on over in the next section.

7.2. Defining Functions Recap

We looked at a few examples of built-in functions in Python, but being able to define your own functions is incredibly powerful. We start a function definition with the `def` keyword, followed by the name we want to give our function. After the name, we have the parameters, also called arguments, for the function enclosed in parentheses.

A function can have no parameters, or it can have multiple parameters. Parameters allow us to call a function and pass it data, with the data being available inside the function as variables with the same name as the parameters. Lastly, we put a colon at the end of the line.

After the colon, the function body starts. It's important to note that in Python the function body is delimited by indentation. This means that all code indented to the right following a function definition is part of the function body.

The first line that's no longer indented is the boundary of the function body. It's up to you how many spaces you use when indenting - just make sure to be consistent. So if you choose to indent with 4 spaces, you need to use 4 spaces everywhere in your code.

7.3. Returning Values

We've seen how we can pass values into a function as parameters by passing values like the name or department in the example earlier. But what about getting values out of a function? This is where the concept of **return** values comes to play. The work that functions do can produce *new* results. Sure, we can print the results on the screen, but what if we wanted to use those results later in our script or didn't want to print them at all?

We can do this by returning values from the functions we defined ourselves. Let's go back to calculating the area of a triangle.

Video 9 (02:49 *Returning Values*)

Do you remember this triangle example from our earlier exercise?

```
base = 6
height = 3
area = (base*height)/2
```

The area of the triangle is calculated as base times height divided by 2. Imagine we need to calculate this value several times in our code. It would be useful to have a function that does this for us. Check out how this would look:

```
def area_triangle(base, height):
    return base*height/2
```

We use the keyword `return` to tell Python that this is the return value of a function. When we call the function, we store that value in a variable. Let's say we have the two triangles and we want to add up the sum of both areas. Here's what we would do. First, we calculate the two areas separately. Then, we add the sum of both areas together:

```
area_a = area_triangle(5, 4)
area_b = area_triangle(7, 3)
sum = area_a + area_b
```

Finally, we print the result converting it to a string.

```
print("The sum of both areas is: " + str(sum))
```

The sum of both areas is: 20.5

The second line of code is the result that Python generates. As you can see in this example, the `area_of_triangle` function returns a value which is not surprisingly the area of the triangle. We store that value in a different variable for each call to the function, in this case, *area_a* and *area_b*. Then we operate with those values adding them into the variable called *sum* and only printing this final result.

This shows the power of the `return` statement. It allows us to combine calls to functions and to more complex operations which makes your code more readable. Return statements in Python are even

more interesting because we can use them to return more than one value.

Let's say you have a duration of time in seconds and you want to convert that to the equivalent number of hours, minutes, and seconds. Here's how to do that in Python:

```
1 def convert_seconds(seconds):
2     hours = seconds//3600
3     minutes = (seconds - hours*3600)//60
4     remaining_seconds = seconds - hours * 3600 - minutes * 60
5     return hours, minutes, remaining_seconds
```

Did you spot the new operator in this function? That // (double slash) operator is called **floor**. A floor division divides a number and takes the **integer part** of the division as the result. For example, `5 // 2` is 2 instead of 2.5.

In our example, the first operation (line 1) is calculating how many hours are in a given amount of seconds, while the second line (line 2) works out how many minutes are left once we subtract the hours. The third line then works out how many seconds remain after subtracting minutes. We end up with three numbers as a result: **return**

So the function returns all three of them. Let's see what this looks like when we're calling our function:

```
hours, minutes, seconds = convert_seconds(5000)
```

```
print(hours, minutes, seconds)
```

```
1 23 20
```

As you can see from the result above, we have 1 hour, 23 minutes, 20 seconds in 5000 seconds. Because we know that the function returns three values, we assign the result of the function to three different variables. There's one last thing we should call out about returning values. It is possible to return nothing and that's perfectly okay. Let's look at an example from section 7.1.

```
def greeting(name):  
    print("Welcome, " + name)
```

Here the function just printed a message and didn't return anything. What do you think would happen if we try to assign the value of this function to a variable? Let's try it out and see.

```
result = greeting("John")
```

```
Welcome, John
```

```
print(result)
```

```
None
```

Here when we call the function, it printed a message just like we expected. We stored the return value in the result variable, but there was no return statement in the function. So the value of results is

None is a very special data type in Python used to indicate that things are empty or that they return nothing.

Wow! That was a lot to learn about functions and the return values. Remember that the key to getting this right is to practice writing the code you've just learned as many times as you need. Functions and return values can be tricky concepts to master, but they let us do a bunch of cool stuff. So put the time and effort into learning for some really valuable returns!

7.4. Returning Values Using Functions

Sometimes we don't want a function to simply run and finish. We may want a function to manipulate data we passed it and then return the result to us. This is where the concept of return values comes in handy.

We use the return keyword in a function, which tells the function to pass data back. When we call the function, we can store the returned value in a variable. Return values allow our functions to be more flexible and powerful, so they can be reused and called multiple times.

Functions can even return multiple values. Just don't forget to store all returned values in variables! You could also have a function return nothing, in which case the function simply exits.

7.5. The Principles of Code Reuse

As we've called out before, functions are powerful because you can create your own. You can use them to organize the code in your scripts into logical blocks, which makes the code you write easier to use and reuse. Check out this example:

Video 10 (01:20 *The Principles of Code Reuse*)

```
name = "Kay"
number = len(name) * 9
print("Hello " + name + ". Your lucky number is " + str(number))
```

```
name = "Cameron"
number = len(name) * 9
print("Hello " + name + ". Your lucky number is " + str(number))
```

As you can see, this code is written in two parts (3 lines for each). Here's our results:

```
Hello Kay. Your lucky number is 27
Hello Cameron. Your lucky number is 63
```

This script uses the **len** function, which returns the length of a string. In this example the script then uses that length to calculate a number, which we're calling the lucky number here. Finally, it prints a message with the name and the number.

Each time you want to perform the calculation, we change the values of the variables and write the formula. Then, print a greeting followed by the lucky number.

See how the following line is duplicated in the first and second part of the code:

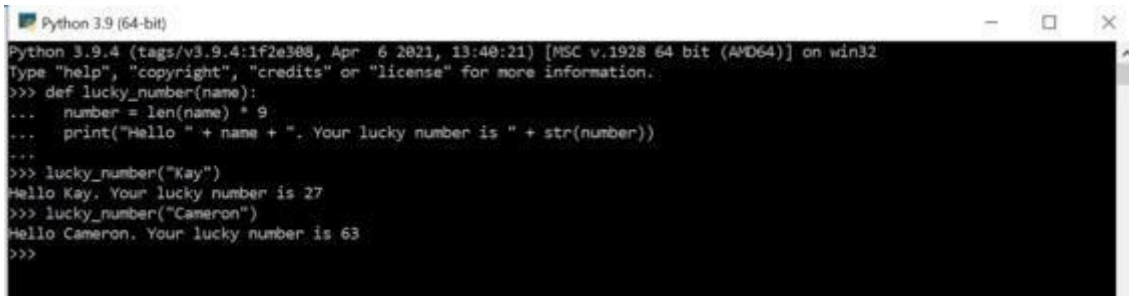
```
number = len(name) * 9
```

When you find **code duplication** in your scripts, it's a good idea to check if you can clean things up a bit by using a function. How about we rewrite this code creating a function to group all the duplicated code into just one line, like this (try to write and run this code in your Python terminal 3.9):

```
def lucky_number(name):  
    number = len(name) * 9  
    print("Hello " + name + ". Your lucky number is " +  
str(number))
```

```
lucky_number("Kay")  
lucky_number("Cameron")
```

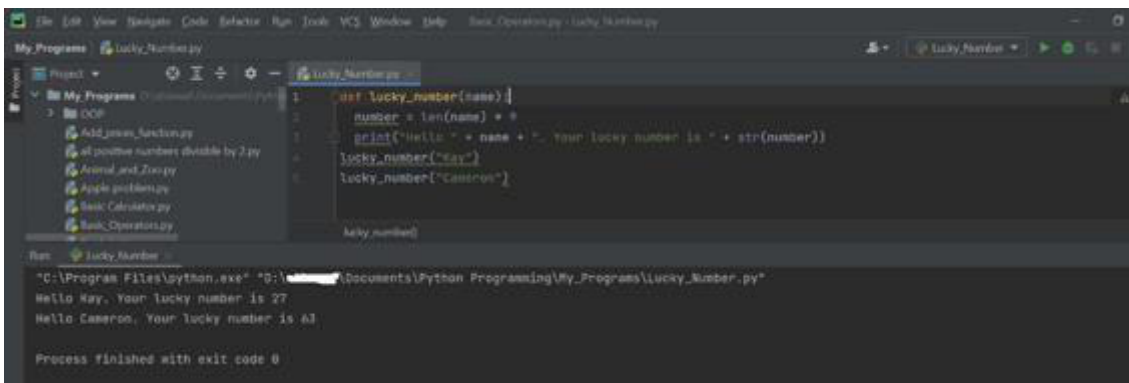
The updated script gives us the exact same result as the original one, but it looks a lot cleaner. See Fig. 7.5.1.



```
Python 3.9 (64-bit)
Python 3.9.4 (tags/v3.9.4:1f2e388, Apr 6 2021, 13:40:21) [MSC v.1928 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license()" for more information.
>>> def lucky_number(name):
...     number = len(name) * 9
...     print("Hello " + name + ". Your lucky number is " + str(number))
...
>>> lucky_number("Kay")
Hello Kay. Your lucky number is 27
>>> lucky_number("Cameron")
Hello Cameron. Your lucky number is 63
>>>
```

Fig. 7.5.1: Lucky_Number script inside Python 3.9.6 Terminal

You should get exactly the same result if you write and run this script in your Pycharm. See Fig. 7.5.2.



```
def lucky_number(name):
    number = len(name) * 9
    print("Hello " + name + ". Your lucky number is " + str(number))
lucky_number("Kay")
lucky_number("Cameron")
```

```
"C:\Program Files\python.exe" "D:\Documents\Python Programming\My_Programs\Lucky_Number.py"
Hello Kay. Your lucky number is 27
Hello Cameron. Your lucky number is 63
Process finished with exit code 0
```

Fig. 7.5.2: Lucky_Number script inside Pycharm 2021.1.2

First, we've defined a function called lucky number, which carries out our calculation and prints it for us. Then we call the function twice, once with each name. Since we've grouped the calculation and print statements into a function, our code is not only easier to read but it's also now reusable. We can execute the code inside the lucky number function as many times as we need it, by just calling it with a different name. So we don't have to write it out and again and again for each new name. Does that make sense?

Hopefully, these examples have helped explain how functions are used and defined, and also demonstrated how useful they can be. Did you notice that we're feeding information into our functions through their parameters? This is one of the many ways that we can input data into our code.

The values for those parameters may come from different places, like a file on our computer or through a form on a website, but that doesn't impact our code. The result of the function is still the same, no matter where the parameters come from.

Functions are your friends. They can help clean up your code and do a math so you don't have to. You'll be using them a lot both in this course and in your programming life. So get ready to get real friendly with functions.

7.6. Code Style

So far, we've looked into how the Python syntax is used for variables, expressions, and defining and using functions. There's a lot more syntax to come but before we dive into that, let's talk a bit about a different side of programming:

On the whole, having good or bad style when you write code doesn't make much difference between a script succeeding or crashing, but it can make a *big* difference for the people who use it and contribute to it. Poor programming style can make life difficult for the IT specialists or system administrators who have to read the script after it's written or make changes to it so it works with a new system.

Bad style can even give the scripts author a headache if it's been awhile since they wrote it. Imagine having to rewrite your own code because it's too messy to understand. Yikes!

On the flip side, good style can make a script look almost like natural human language. It can make the scripts intent and construction immediately clear to the reader. Good style makes life easier for people who have to maintain the code and helps them understand what it does and how it does it. It can also reduce errors since it makes updating the code easier and more straightforward. Most importantly good style makes you look cool, right?

So we agree, our code should be stylish. But what makes the style of a piece of code good or bad? Although there are no hard and fast rules that apply to every programming language and situation, keeping a few principles in mind will go a long way to creating good well styled code.

7.6.1. Principles for Creating Well-styled Code

First off, you want your code to be self-documenting as possible. Self-documenting code is written in a way that's readable and doesn't conceal its intent. This principle can be applied to all aspects of writing code, from picking your variable names to writing clear concise expressions. This is explained further in this video:

Video 11 (0:46 *The Principles of Code Reuse*)

Take this code snippet for example:

```
def calculate(d):  
    q = 3.14  
    z = q* (d**2)  
    print(z)  
calculate(5)  
78.5
```

It's hard to determine the purpose of this code by just looking at it. The names of the variables don't give the reader much information and although you can likely work out the result of the calculation, there are no clues to what that result (78.5) might be used for.

In programming lingo, when we re-write code to be more self-documenting, we call this process This is how it would look if we

refactored this code:

```
def circle_area(radius):  
    pi = 3.14  
    area = pi * (radius**2)  
print(circle_area(5))  
78.5
```

With this refactored code, the intent should now be more clear. The names of the variables and the function reflect their purpose, which helps the reader understand the code more quickly.

You should always aim for your code to be self-documenting. But even then, sometimes you may need to use a particularly tricky bit of code in your script. When good naming and clean organization can't make the code clear, you can add a bit of explanatory texts to the code.

You do this by adding what we call a comment. As already discussed in section 4.1, Python comments are indicated by the # (hash) character. When your computer sees a # character and understands that it should ignore everything that comes after that character on that line. Check out how this looks:

```
# This is how you write a comment in Python!
```

Using comments lets you explain why a function does something a certain way. It also allows you to leave notes to your future self

or other programmers to remind you of what needs to be improved and why. Obviously, it's much easier to read your own code than someone else's.

In my job, I work on code that was written by lots of different people and everybody designs things a little differently. This is why it's so important to comment and document your code well. More often than not, your code will eventually be used by someone other than you. So be a good neighbor.

Use the style guide to structure your code in a way that's readable by others or by you in six months when you've forgotten why you wrote that code in the first place. In upcoming exercises in this course, we'll use comments to let you know what you need to do with the code. You can always write as many extra comments as you need.

Coming up, a quiz to consolidate your newly acquired knowledge about functions. Don't worry. You've got this.

7.7. Practice Quiz 4: 5 Questions

1. This function converts miles to kilometers - 1 point

```
def convert_distance(miles):
```

```
    km = miles * # approximately 1.6 km in 1 mile
```

```
    return
```

```
my_trip_miles = 55
```

```
# 2) Convert my_trip_miles to kilometers by calling the function  
above
```

```
my_trip_km = convert_distance(_)
```

```
# 3) Fill in the blank to print the result of the conversion  
distance in kilometers is " +
```

```
# 4) Calculate the round-trip in kilometers by doubling the result,  
# and fill in the blank to print the result
```

```
round-trip in kilometers is " +
```

Complete the function to return the result of the conversion.

Call the function to convert the trip distance from miles to kilometers.

Fill in the blank to print the result of the conversion.

Calculate the round-trip in kilometers by doubling the result, and fill in the blank to print the result.

2. This function compares two numbers and returns them in increasing order. Fill in the blanks, so the print statement displays the result of the function call in order. – 1 point

if a function returns multiple values, don't forget to store these values in multiple variables.

```
# This function compares two numbers and returns them  
# in increasing order.
```

```
def order_numbers(number1, number2):  
    if number2 > _:  
        return _, _  
    return number2, number1
```

```
# 1) Fill in the blanks so the print statement displays the result  
# of the function call.
```

```
smaller, bigger = order_numbers(100, 99)  
bigger)
```

3. What are the values passed into functions as input called?
Select the correct response – 1 point

Variables

Return values

Parameters

Data types

4. Let's revisit our lucky_number function. We want to change it, so that instead of printing the message, it returns the message.

This way, the calling line can print the message, or do something else with it if needed. Fill in the blanks to complete the code to make it work. – 1 point

```
def lucky_number(name):  
    number = * 9  
    message = "Hello " + name + "Your lucky number is " +  
return _
```

5. What is the purpose of the def keyword? – 1 point

Used to define a new function

Used to define a return value

Used to define a new variable

Used to define a new parameter

7.7.1. Answers to Practice Quiz 5

1.

```
def convert_distance(miles):
```

```
    km = miles * # approximately 1.6 km in 1 mile
```

```
    return
```

```
my_trip_miles = 55
```

```
# 2) Convert my_trip_miles to kilometers by calling the function  
above
```

```
my_trip_km = convert_distance(my_trip_miles)
```

```
# 3) Fill in the blank to print the result of the conversion  
distance in kilometers is " +
```

```
# 4) Calculate the round-trip in kilometers by doubling the result,  
# and fill in the blank to print the result
```

```
round-trip in kilometers is " +
```

2.

```
# This function compares two numbers and returns them  
# in increasing order.
```

```
def order_numbers(number1, number2):
```

```
if number2 > number1:  
    return number1, number2  
return number2, number1
```

```
# 1) Fill in the blanks so the print statement displays the result  
# of the function call.  
smaller, bigger = order_numbers(100, 99)
```

3. C. A parameter, also sometimes called an argument, is a value passed into a function for use within the function.

4.

```
def lucky_number(name):  
    number = * 9  
    message = "Hello " + name + "Your lucky number is " +  
return message
```

5. A. When defining a new function, we must use the def keyword followed by the function name and properly indented body.

8. Conditionals

8.1. Comparing Things

We've seen a few arithmetic expressions so far, like addition, subtraction, and division. Remember when we turned Python into a calculator? Well, Python can also compare values. This lets us check whether something is smaller than (<), equal to (=), or bigger than (>) something else. This allows us to take the result of our expressions and use them to make decisions.

Check out these examples:

Video 12 (3:05 *Comparing Things*)

```
print(10>1)
```

True

In this example, 10 is greater than 1, so the value **True** is printed as a result. True is a value that belongs to another data type called the

Booleans represent one of two possible states, either true or false.

Every time you compare things in Python the result is a Boolean of the appropriate value.

```
print("cat" == "dog")
```

False

In this example we can see our very first **equality operator** (`==`), which is formed by putting two equal signs together. We use this operator to test whether two things are equal to each other. In this example the string `cat` is *not equal* to the string `dog`, so the Boolean that's printed is

```
print(1 != 2)
```

True

In this example we're doing the opposite comparison. By pairing **!** (exclamation mark) and an equal sign, we're using the **not equals operator** (`!=`), which is the *negated* form of the equality operator. In this particular line of code the operator checks that `1` is not equal to `2`.

We call out before that the plus operator doesn't work between integers and strings. What do you think will happen if we try to compare an integer and string? Let's find out by seeing if the number `1` is taller than the string `1`.

```
print(1 < "1")
```

Traceback (most recent call last):

File "", line 1, in

TypeError: '<' not supported between instances of 'int' and 'str'

As you can see, we get a *type* error. That's the same error we got before. This happens because Python doesn't know how to check if a number is smaller than a string. And what about the equality operator?

```
print(1 == "1")
```

False

In this case the Interpreter has no problem telling us that the integer 1 and the string 1 aren't the same. Basically although they may seem similar to us because they both contain the same number, it's clear to the computer that one is a number and the other is the string. For the computer it's obvious that they are completely different entities.

On top of the comparison and equality operators, Python also has a set of **logical** operators. These operators allow you to connect multiple statements together and perform more complex comparisons. In Python, the logical operators are the words `and` and `or`. Let's look at some examples:

```
print("Yellow" > "Cyan" and "Brown" > "Magenta")
```

False

To evaluate as true, the *and* operator would need *both* expressions to be true at the same time here.

Here, we're comparing strings, and the > and < operators refer to alphabetical order. Yellow comes after cyan, but brown doesn't come after magenta. So this means that the **first statement is** but the **second one is not** which makes the **result of the whole expression**

If we use the **or** operator instead, the expression will be true if **either** of the expressions are true, and false only when **both** expressions are false. Let's try it out.

```
print(25 > 50 or 1 != 2)
```

True

25 is definitely not bigger than 50, but 1 is different than 2. So in the end the whole expression is true.

Last up, the **not** operator **inverts** the value of the expression that's in front of it. If the expression is true, it becomes false. If it's false, it becomes true, just like this:

```
print(not 42 == "Answer")
```

True

Logical operators are important because they help us write more complex expressions. We'll see this in action in the next few sections. If this is the first time you've come across these operators it might seem like there's a lot to remember. But don't worry, you'll learn most of them very quickly just by practicing.

In the section 8.9, we have a cheat sheet that lists all the operators available and what each one does. It's a handy resource you're sure to find useful when writing your own scripts.

8.2. Comparison Operators Recap

In Python, we can use comparison operators to compare values. When a comparison is made, Python returns a boolean result, or simply a True or False.

To check if two values are the same, we can use the equality operator: `==`

To check if two values are not the same, we can use the not equals operator: `!=`

We can also check if values are greater than or lesser than each other using `>` and `<`. If you try to compare data types that aren't compatible, like checking if a string is greater than an integer, Python will throw a

We can make very complex comparisons by joining statements together using logical operators with our comparison operators. These logical operators are `and` and `or`. When using the **and** operator, both sides of the statement being evaluated must be true for the whole statement to be true.

When using the **or** operator, if either side of the comparison is true, then the whole statement is true. Lastly, the **not** operator simply inverts the value of the statement immediately following it. So if a statement evaluates to True, and we put the **not** operator in front of it, it would become False.

8.3. Branching with IF statements

Now that we're armed with knowledge of Python's expressions, comparators, and variables, we can dive right into how to use them in our scripts to perform different actions based on their values. The ability of a program to alter its execution sequence is called and it's a key component in making your scripts useful.

You probably use the idea of branching a bunch in your everyday life. For example, if it's before noon, you might greet someone by saying good morning instead of good afternoon or good evening. If it's raining outside, you might choose to take an umbrella. If it's cold, you probably wear a jacket.

In your scripts, you can instruct your computer to make decisions based on inputs too. Let's take a look at an IT-focused example. In many companies, new employees can choose the username they'll use to access the company's systems, and usually, the chosen username needs to fit with a given set of guidelines. Companies can set different criteria for what a valid username looks like.

Video 13 (1:14 *Branching with if statements*)

For now, let's assume that at your company, a valid username has to have at least three characters. You've been tasked with writing a program that will tell the user if their choices valid or not. To do that, you could write a function like this:

```
def hint_username(username):  
  
    if len(username) < 3:  
  
        print("Invalid username. Must be at least 3 characters long")
```

This function checks whether the length of the username is smaller than 3. If it is, the function prints a message saying that the username is invalid. Look closely at how the if statement is written. We write the keyword **if** followed by the **condition** that we want to check for, and then followed by a *After that*, comes the body of the if block, which is **indented** further to the right.

You may notice that there are some **similarities** between how an if block and the function are defined. The keyword, either **def** or **if**, indicates the start of a special block. At the end of the first line, we use a colon, and then the body of the function or the if block is indented to the right.

But there's also an important **difference** between how an if block and a function are defined: The body of the if block will only execute when the condition evaluates to true; otherwise, it is skipped.

Of course, you can do a lot more things inside the body of the if block than just printing stuff. As we expand our programming abilities, we'll learn how to do things like shorten texts that's too long, delete a file if it exists, start a service if it's not running,

and a bunch more. If your code is inside a function, you could also choose to return a value depending on whether a certain condition is met. Can you imagine how that would look?

By now, you know how to define functions, and inside those functions, you can now make your program do something only when certain conditions are met. Ready to branch out and make our branches even more interesting with **else** statements?

Then hop on over to the next section, or else, you'll miss out!

8.4. If Statements Recap

We can use the concept of **branching** to have our code alter its execution sequence depending on the values of variables. We can use an *if* statement to evaluate a comparison. We start with the *if* keyword, followed by our comparison. We end the line with a colon. The body of the *if* statement is then indented to the right. If the comparison is the code inside the *if* body is executed. If the comparison evaluates to then the code block is skipped and will not be run.

8.5. Else Statements

The if statement is already a pretty useful construct, but we can extend it to make it even more powerful. Think about the username example from section 8.3.

Video 14 (1:36 *Branching with if statements*)

What if we also wanted to print a message when the username was valid? We can include an **else** statement to achieve this:

```
def hint_username(username):
    if len(username) < 3:
        print("Invalid username. Must be at least 3 characters long")
    else:
        print("Valid username")
```

The program can now go in one of two directions depending on the length of the username. If it's not long enough, we get a message indicating that the username is invalid. But if the program verifies that the username is long enough, it will print a message saying it is valid.

Pay attention to how the else statement is written. It uses the *else* keyword followed by a colon to indicate the beginning of the else block. Once again, the body of the block is further indented to the right.

As we've called out before these blocks can contain multiple lines and do more than just print messages. They can do calculations, modify values, return values, and a lot more.

And remember that you can choose to use as many or as few spaces as you want for the indentation, but you always need to indent and you always need to use the same number of spaces.

The `else` statement is very useful, but we don't always need it. Say we want to have a function that checks if a value is even or odd. We could do that with a piece of code like this:

```
def is_even(number)
    if number % 2 == 0:
        return True
    return False
```

Here, we're using a new operator so let's first explain that. The **modulo** operator is represented by the `%` (percentage) sign. It returns the **remainder** of the integer division between two numbers. The integer division is an operation between integers that yields two results which are both integers, the quotient and the remainder. So if we do an integer division between 5 and 2, the quotient is 2 and the remainder is 1.

If we do an integer division between 11 and 3, the quotient is 3 and the remainder is 2. Even numbers are all multiples of 2 which means the remainder of the integer division between an even number and 2 is always going to be 0.

In this function, we're using this principle to decide whether a number is even or not. So how come we have these two return statements, one below the other, without an else statement? The trick is that when a return statement is executed, the function exits so that the code that follows doesn't get executed. This means that if the number is even, the computer will reach the "return True" statement and exit the function.

Anything that comes after that will only be executed if the condition in the if statement was false. In other words, once the function reaches the "return False" line, we know for sure that the if condition was false which means the number was odd.

At first, you might feel more comfortable including the else statement, even if it's not needed and that's totally okay. It's important to know that both ways of writing this are correct. Remember that this technique can only be used when you're returning a value inside the if statement.

To recap, the *if* statement allows us to branch the execution based on a specific condition being true. The *else* statement lets us set a piece of code to run only when the condition of the if statement was false. If you return a value inside an if block then the code after the block will only be executed if the condition was false. All make sense?

If all these if's and else's are starting to get a little confusing, that's okay. There's a lot to soak up here and the best way to do

that is yeah, you guessed it, So review the content and practice on your own as much as you need. Once you're done, go to the next section.

8.6. Else Statements and Modulo Operator Recap

We just covered the *if* statement, which executes code if an evaluation is true and skips the code if it's false. But what if we wanted the code to do something different if the evaluation is false? We can do this using the *else* statement.

The *else* statement follows an *if* block, and is composed of the keyword *else* followed by a colon. The body of the *else* statement is indented to the right, and will be executed if the above *if* statement doesn't execute.

We also touched on the modulo operator, which is represented by the percent sign: This operator performs integer division, but only returns the remainder of this division operation.

If we're dividing 5 by 2, the quotient is 2, and the remainder is 1. Two 2s can go into 5, leaving 1 left over. So $5\%2$ would return 1. Dividing 10 by 5 would give us a quotient of 2 with no remainder, since 5 can go into 10 twice with nothing left over. In this case, $10\%2$ would return 0, as there is no remainder.

8.7. Elif Statements

The *if* and *else* blocks allow us to branch execution depending on whether a condition is true or false. But what if there are more conditions to take into account? This is where the **elif** statement, which is short for **else if** comes into play.

But before we jump into how to use it, let's take a look at why we need it in the first place. Let's go back to our trusty username validation example.

Video 15 (1:04 *Elif Statements*)

```
def hint_username(username):
    if len(username) < 3:
        print("Invalid username. Must be at least 3 characters long")
    else:
        print("Valid username")
```

Now, what if your company also had a rule that usernames longer than 15 characters aren't allowed? How could we let the user know if their chosen username was too long? We could do it like this:

```
def hint_username(username):
    if len(username) < 3:
        print("Invalid username. Must be at least 3 characters long")
    else:
```



```
    if len(username) > 15:
        print("Invalid username. Must be at most 15 characters
long")
    else:
print("Valid username")
```

In this case, we're adding an extra **if** block *inside* the **else** block (shown in red color). This works, but the way the code is **nested** makes it kind of hard to read. To avoid unnecessary nesting and make the code clearer, Python gives us the *elif* keyword, which lets us handle more than two comparison cases. Take a look.

```
def hint_username(username):
    if len(username) < 3:
        print("Invalid username. Must be at least 3 characters long")
    elif len(username) > 15:
print("Invalid username. Must be at most 15 characters long")
    else:
        print("Valid username")
```

The *elif* statement looks very similar to the *if* statement (shown in blue color). It's followed by a condition and a colon, and a block of code indented to the right that forms the body. The condition must be true for the body of the *elif* block to be executed.

The main **difference between elif and if statements** is we can only write an *elif* block as a companion to an *if* block. That's because the condition of the *elif* statement will only be checked if the condition of the *if* statement wasn't true.

So in this example, the program first checks whether the username is less than 3 characters long, and prints a message if that's the case. If the username has at least 3 characters, the program then checks if it's longer than 15 characters. If it is, we get a message to tell us that.

Finally, if none of the above conditions were met, the program prints a message indicating that the username is valid.

There's **no limit** to how many conditions we can add, and it's easy to include new ones. For example, say the company decided that the username shouldn't include numbers. We could easily add an extra `elif` condition to check for this. Cool, right?

You now know how to compare things and use those comparisons for your `if`, `elif`, and `else` statements, and you are using all of them inside functions. Using branching to determine your program's flow opens up a whole new realm of possibilities in your scripts. You can use comparisons to pick between executing different pieces of code, which makes your script pretty flexible.

Branching also helps you do all kinds of practical things like

only backing up files with a certain extension, or only allowing login access to a server during certain times of the day.

Any time your program needs to make a decision, you can specify its behavior with a branching statement. Are you starting to notice tasks in your day-to-day that could be made more efficient with scripting? There are so many possibilities, and we're only just getting started with all the cool stuff Python programming can help you do.

Wow! we've covered a lot in these last few sections. Remembering all these concepts can take some time, and the best way to learn them is to use them (learn by doing). So I've put together a cheat sheet for you:

8.8 Cheat Sheet 2: Conditionals

You'll find all these operators and branching blocks listed in your *Cheat Sheets* folder in one handy resource. It's super useful when you need a quick refresher.

8.9. More Complex Branching with `elif` Statements

Building off of the `if` and `else` blocks, which allow us to branch our code depending on the evaluation of one statement, the `elif` statement allows us even more comparisons to perform more complex branching.

Very similar to the `if` statements, an `elif` statement starts with the `elif` keyword, followed by a comparison to be evaluated. This is followed by a colon, and then the code block on the next line, indented to the right. An `elif` statement must follow an `if` statement, and will only be evaluated if the `if` statement was evaluated as false. You can include multiple `elif` statements to build complex branching in your code to do all kinds of powerful things!

8.10. Practice Quiz 6: 5 Questions

1. What's the value of this Python expression: `(2**2) == 4`? - 1 point

4

`2**2`

True

False

2. Complete the script by filling in the missing parts. The function receives a name, then returns a greeting based on whether or not that name is "Taylor". - 1 point

```
_ "Hello there, " + _
```

3. What's the output of this code if number equals 10? - 1 point

4. Is "A dog" smaller or larger than "A mouse"? Is $9999+8888$ smaller or larger than $100*100$? Replace the plus sign in the following code to let Python check it for you and then answer. - 1 point

5. If a filesystem has a block size of 4096 bytes, this means that a file comprised of only one byte will still use 4096 bytes of storage. A file made up of 4097 bytes will use $4096*2=8192$ bytes of storage. Knowing this, can you fill in the gaps in the `calculate_storage` function below, which calculates the total number of bytes needed to store a file of a given size?

Use floor division to calculate how many blocks are fully occupied.

_ block_size

```
# Use the modulo operator to check whether there's any remainder
```

```
# Depending on whether there's a remainder or not, return
```

```
# the total number of bytes required to allocate enough blocks
```

```
# to store your data.
```

```
+ _) * block_size
```

```
* full_blocks
```


8.10.1. Answers to Practice Quiz 6

1. C. The conditional operator `==` checks if two values are equal. The result of that operation is a boolean: either True or False.

2.

3. 2. Our number is 10, which is < 12 , so it matches that condition.

4.

5.

```
# Use floor division to calculate how many blocks are fully occupied.
```

```
// block_size
```

```
# Use the modulo operator to check whether there's any remainder
```

```
# Depending on whether there's a remainder or not, return
```

```
# the total number of bytes required to allocate enough blocks
```

```
# to store your data.
```

```
+ 1) * block_size
```

```
* full_blocks
```

```
# Answer Should be 4096.
```

```
# Answer Should be 4096.
```

```
# Answer Should be 8192.
```

Answer Should be 8192.

9. Module Review

9.1. Basic Syntax Wrap Up

You just completed your second module and learned a whole lot about Python syntax. Congrats!

Video 16 (0:18 *Basic Syntax Wrap Up*)

We've learned how to operate with different data types and how to create our own variables and expressions. We've defined our first functions and learned how to make them return values so that they're more reusable.

We then dove into creating branches in our scripts which lets them act in different ways depending on the values of our variables. We learned a lot of new and very powerful stuff. Knowing how to structure your code and functions and how to make your code act in different ways depending on the values is what allows us to tell our computer what to do. We'll keep using these tools throughout the course as we move on to more complex and interesting things.

Next step, you can put everything you've learned to the test in the next graded assessment. Don't worry if you don't feel ready yet. Remember that you can re-watch the videos and do the practice quizzes as many times as you need to make sure you fully understand everything we've covered.

When you're ready for the test, take your time and best of luck.
I'll catch you after you finished in the next module, where we'll
learn all about loops. See you there!

9.2. Why I Like Python

My team is responsible for maintaining the operating system of a bunch of computers in the fleet of Google. And many of the things we do are done through Python scripts. For example, we have a script that keeps the computer up to date. The software is updated everyday and is written in Python.

We also have a script that ensures the computer doesn't have any specific problems, and if there is a problem, raises an alert for the user so that the he can take action. That one is also written in Python.

We have a bunch of other scripts like that run in the computers of our users that are all written in Python. One of the things I like the most about Python is that the code is really readable. You can give a piece of Python code to someone that doesn't even know how to program and most of the time, they will have at least some idea of what is going on.

Another thing I like about Python is that it comes with a lot of modules that do a lot of the things we want to do already. Python has been around for a while. There's a lot of people that have contributed all of these modules. So it's very likely that the thing you want to do, is already in a module, and you only have to import it and use it.

9.3. What I Don't Like About Python

No computer language is perfect. Every computer language has its advantages and disadvantages. In the case of Python, the one thing that I find the most annoying is that, because it is not a compiled language, there could be errors in the code that only gets detected very late in the development process.

And well, it's not good that all these hidden errors could be in the code. If you write tests codes that can test all of your code, then you can be assured that your code works successfully. That's why I think Python is great for writing small scripts that are self-contained and sometimes for big software projects that have a lot of infrastructure on them.

9.4. Module 2 Graded Assessment - 10 Questions

It's time again for your second graded assessment. Open the "Graded Assessments" folder you downloaded earlier. It contains the pdf files of all the graded assessments in this course. Here's the file name to search:

Module 2 Graded Assessment – *file name*

9.4.1. Solutions to Module 2 Graded Assessment:

I (or any member of my team) is available to help you grade your assessments. You can use my help link (email) at the end of chapter 25 to send your assessment for grading. We will get back to you in 12 to 24 hours with your result.

However, if you cannot wait, you can open the “Graded Assessments” folder you downloaded earlier. It contains the pdf formats of the solutions to all graded assessments in this course. You can use them to grade your assessments by yourself. Just be honest as you grade. Here’s the file name to search:

Module 2 Graded Assessment Solutions – *file name*

Module 3

“Happiness is the longing for repetition” – Milan Kundera

10. While Loops

10.1. Introduction to Loops

Welcome back. Before we dive back in, I want to first say well done. You've learned a lot of new skills in a short amount of time and tackled some pretty tricky concepts. None of this stuff is easy and you're doing so great.

So we've got some fun concepts lined up for the next few sections. So far we've seen how to organize our code and functions. We've also made our code branch in two different paths depending on certain conditions.

In this module we'll learn how to get computers to do repetitive tasks, which is another cornerstone of programming. As we've called out before computers are great at **repeating the same task over and over** and they never get bored or make a mistake.

You could ask a computer to do the same calculation a thousand times in the first result would be just as accurate as the last, which isn't something we can say about us humans. Have you ever tried to do something at thousand times in a row? It could be enough to drive you loopy, which is why in this course, we're going to learn how to leave the loops up to the computer.

The ability to accurately perform repetitive tasks and never get tired is why computers are so great for automation. The automated task could be anything like copying files to a bunch of computers on a network, sending personalized emails to a list of users, or verifying that a process is still running.

It doesn't matter how complex the task is your computer will do it as many times as you tell it to, which leaves you time for more interesting things like planning future hardware needs, or managing software roll out.

In the next few sections we'll explore three techniques for automating repetitive tasks. These are **while** **for** and **Each** of these techniques are used to tell the computer to repeat a task, but each takes a slightly different approach.

We're going to learn how to write the code for each, and how to know when to use one technique instead of the others. So, are you ready? Let's get started!

10.2. What is a While loop?

First off, we're going to talk about *while* loops.

While Loops instruct your computer to continuously execute your code based on the value of a condition.

This works in a similar way to branching if statements. The difference here is that the body of the block can be executed **multiple times** instead of just once. Check out this video:

Video 17 (2:20 *What is a while loop?*)

Type these 5 lines of code in your Pycharm IDE in the order it is written. Do not copy and paste:

```
1 x = 0
2 while x < 5:
3     print("Not there yet, x = " + str(x))
4     x = x + 1
5     print("x=" + str(x))
```

Can you guess what it does? Before we execute it to find out, let's go through it together line by line.

In the first line (1), we're assigning the value of 0 to the variable `x`. We call this action which means giving an initial value to a variable. In the line two, we're starting the while loop. We're

setting a condition for this loop that X needs to be smaller than 5. Right now we know that x is 0 since we've just initialized it, so this condition is currently true.

On the next two lines (3 and 4), we have a block that's indented to the right. Here, we can use what we learned about functions and conditionals to identify that this is the while loop's

There are two lines in the body of the loop. In the first line (3), we print a message followed by the current value of X. In the second line (4), we **increment** the value of X. We do this by adding 1 to its current value and assigning it back to X.

So after the first execution of the body of the loop, X will be 1 instead of 0. Because this is a loop, the computer doesn't just continue executing with the next line in the script. Instead, it **loops back** around to re-evaluate the condition for the while loop.

Because 1 here is still smaller than 5, it executes the body of the loop. It then prints the message and, once more, increments X by 1. So the X is now 2.

The computer will keep doing this until the condition isn't true anymore. In this example, the condition will be false when X is no longer smaller than 5. Once the condition is false, the **loop** and the next line is executed. Finally, the last line of our code prints the last value of X.

Now that this code makes a bit more sense, what do you think will happen when we execute it? Ready to find out? Let's execute the code and see what happens:

```
Not there yet, x = 0  
Not there yet, x = 1  
Not there yet, x = 2  
Not there yet, x = 3  
Not there yet, x = 4  
x=5
```

So we had five lines with the message, Not there yet, and then at the end of the script the value of X was 5. This was a simple example of how a while loop behaves.

As we've said before, we're learning the building blocks of programming. Once you know those building blocks, you can combine them to create more complex expressions. As an IT specialist, while loops can be super helpful. You can use them to keep asking for a username if the one provided isn't valid, or maybe try an operation until it succeeds.

Knowing how to construct these expressions can help you get your computer to do a whole lot with only a little bit of code. It's pretty powerful stuff we're learning here. Now that you've got an idea of how a while loop works, we will spice it up with another example in section 10.4.

10.3. Anatomy of a While Loop

A *while* loop will continuously execute code depending on the value of a condition. It begins with the keyword *while*, followed by a comparison to be evaluated, then a colon. On the next line is the code block to be executed, indented to the right.

Similar to an *if* statement, the code in the body will only be executed if the comparison is evaluated to be true. What sets a *while* loop apart, however, is that this code block will keep executing as long as the evaluation statement is true. Once the statement is no longer true, the loop exits and the next line of code will be executed.

10.4. More While Loop Examples

In section 10.2, we saw a very simple example of a while loop. We looked at a basic syntax of the loop and how it works. Let's now apply this knowledge to a similar example, but this time with a while loop inside a function.

Video 18 (2:04 *More while loop Examples*)

Type this code in your Pycharm IDE in the order it is written. Do not copy and paste:

```
def attempts(n):  
    x = 1  
    while x <= n:  
        print("Attempt " + str(x))  
        x += 1  
    print("Done")
```

```
attempts(5)
```

Can you work out what this function does? In this example, we start out by initializing a variable called X. In this case, we initialize it with a value of 1. Then, we enter our while loop which checks to see if the value inside of the X variable is less than the parameter n that the function received.

If that comparison evaluates to true, then the code inside the while block is executed. Say we pass a value of 5 as a parameter to this function.

In the first pass through the loop, X is always equal to 1, so the comparison 1 smaller than or equal to 5 would be true and we then enter the body of the loop.

In the body, we first print a message indicating *that* current attempt number, and then we increase the value of X by 1. To increment the number we're using a slightly different expression than before.

`x +=1` is a **shorthand version** of `x =` You can use *either* expression since they both mean the same thing.

The process continues until the result of the comparison isn't true anymore, which happens when X is bigger than n. In our current example, this would be when the value of x is 6. Let's run this script to see it in action.

```
Attempt 1
Attempt 2
Attempt 3
Attempt 4
Attempt 5
Done
```

In these past examples, we've used the simple conditions of a number being smaller, or smaller or equal than another number. These are common conditions, but they're by no means the only conditions you can have in a while loop. It's common for example to call a separate function that evaluates the condition, like this (not complete):

```
Username = get_username()
while not valid_username(username):
    print("Invalid username")
    username = get_username()
```

In this case, there's a lot of code hidden behind functions and it's doing stuff we don't see.

There's a **get_username()** function that asks the user for a username and a function that validates that username. All this is happening in just a handful of characters.

As you can see, you can pack a lot of punch into just a short line of code. In this case, the body of the while loop will be executed until the user enters a valid username.

The important thing to remember is that the condition used by the while loop needs to evaluate to true or false. It doesn't matter if this is done by using comparison operators or calling additional functions.

The conditions used in while loops can also become more complex if we use the logical operators that we encountered when looking into branching, and This lets us combine the values of several expressions to get the result we want.

Okay, we've now covered what a while loop is and learned its syntax and basic behavior. Some of this stuff can be a bit tricky and you're doing great. Keep sticking with it.

Next, we're going to do a rundown of some of the most common pitfalls that you may come across when writing your own loops. Head on over to the next section to get started.

10.5. Why Initializing Variables Matters

As we've called out earlier writing loops allows us to get our computer to do repetitive work for us. So one of the main benefits of writing scripts in IT is to save time by automating repetitive tasks. Loops are super useful. So let's make sure you avoid some of the most common mistakes people make when writing loops.

One of the most common errors is **forgetting to initialize variables with the right** We've all made this mistake when starting to code. Remember how in the earlier examples we initialized our variable X to 0 in one case and to 1 in the other.

When we forget to initialize a variable two different things can happen as explained in this video:

Video 19 (2:10 *Why Initializing Variables Matters*)

The first possible outcome and the easiest to catch is that Python might raise an error telling us that we're using a variable we haven't defined, which looks like this:

```
while my_variable < 10:
```

```
    my_variable += 1
```

Traceback (most recent call last):

```
File Programming\My_Programs\Book Jotter.py", line 1, in
    while my_variable < 10:
```

```
NameError: name 'my_variable' is not defined
```

As we've done with other errors we've come across, we can look at the last line to understand what's going on. This error type is a *NameError* and the message that comes after it says we're using an undefined variable. It's straightforward to fix, we just need to initialize the variable before using it like this:

```
my_variable = 5
while my_variable < 10:
    print("Hello")
    my_variable += 1
```

```
Hello
Hello
Hello
Hello
Hello
```

As you can see we now have our output. So the error is fixed.

Now, there's a second issue we might face if we forget to initialize variables with the right value. We might have *already used* the variable in our program. In this case, if we **reuse** the variable without setting the correct value from the start, it will still have the value from before. This can lead to some pretty unexpected behavior. Try this script:


```
x = 1
sum = 0
while x < 10:
    sum += x
    x += 1

product = 1
while x < 10:
    product = product * x
    x += 1

print(sum, product)
```

Can you spot the problem?

In the first block, we correctly initialize X to 1 and sum to 0, and then iterate until x equals 10, summing up all the values in between. So by the end of that block, sum equals the result of adding all the numbers from 1 to 10 and X is 10.

In the second part of the code, the original intention was to get the *product* of all the numbers from 1 to 10, but if you look closely, you can see that we're initializing *product* but **forgetting to initialize** So X is still 10.

This means that when the *while* condition gets checked, X is **already 10** at the start of the iteration. The *while* condition is false before it even starts and the body never executes.

If you run this script yourself, you can confirm that this is the output:

```
45 1
```

In this case, it might be harder to catch the problem because python doesn't raise an error. The problem here is that our *product* variable has the wrong value (1). If you have a loop that's gone rogue and not behaving as expected, it's a good idea to check if all the variables are correctly initialized.

In this example, we need to set X back to 1 before starting the second loop, like this:

```
x = 1
product = 1
while x < 10:
    product = product * x
    x += 1
```

```
print(sum, product)
```

```
45 362880
```

Fix this script by your self and confirm that the correct value of *product* is 362880.

As always, the best way to learn is to practice it yourself. Makes sense? Remember, if you ever feel stuck or a little unsure about something you can always ask for help in the discussion forums. These forums are there to let you get the help you need when you need it, so don't forget to use them.

So, to recap, whenever you're writing a loop check that you're initializing all the variables you want to use before you use them. Don't worry if you don't get it right the first time, we've all been there when learning how to code.

As we've called out before, the way to master programming is to practice, practice and practice. Keep practicing until you're comfortable and even then it's still okay to make mistakes. So don't feel like you can't loop back around to review and practice everything we've covered so far!

10.6. Common Pitfalls with Variable Initialization

You'll want to watch out for a common mistake: **forgetting to initialize**. If you try to use a variable without first initializing it, you'll run into a `NameError`. This is the Python interpreter catching the mistake and telling you that you're using an undefined variable. The fix is pretty simple: initialize the variable by assigning the variable a value before you use it.

Another common mistake to watch out for that can be a little trickier to spot is forgetting to initialize variables with the correct value. If you use a variable earlier in your code and then reuse it later in a loop without first setting the value to something you want, your code may wind up doing something you didn't expect. Don't forget to initialize your variables before using them!

10.7. Infinite Loops and How to Break Them

You may remember by now that while loops use the condition to check when to exit. The body of the while loop needs to make sure that the condition being checked will change. If it doesn't change, the loop may never finish and we get what's called an **infinite** a loop that keeps executing and never stops. Check out the example in this video:

Video 20 (1:21 *Infinite Loops and How to Break Them*)

```
while x % 2 == 0:  
    x = x / 2
```

It uses the **modulo** operator that we saw a while back. This cycle will finish for positive and negative values of X. But what would happen if X was 0 (zero)? The remainder of 0 divided by 2 is 0, so the condition would be true. The result of dividing 0 by 2 would also be zero, so the value of X wouldn't change.

This loop would go on for ever, and so we'd get an infinite loop. If our code was called with X having the value of zero, the computer would just waste resources doing a division that would never lead to the loop stopping.

The program would be stuck in an infinite loop circling back endlessly, and we don't want that. All that looping might make

your computer “dizzy”. To avoid this, we need to think about what needs to happen for a loop to be successful. Look at this one:

```
if x != 0:  
    while x % 2 == 0:  
        x = x / 2
```

In this example, we said that X needs to be different than zero. So we could nest this while loop inside an if statement just like this. With this approach, the while loop is executed only when X *is not* zero.

Alternatively, we could add the condition directly to the loop using a logical operator like in this example:

```
while x != 0 and x % 2 == 0:  
    x = x / 2
```

This makes sure we only enter the body of the loop for values of X that are both different than zero and even. Talking about infinite loop reminds me of one of the first times I used while loops myself.

I wrote a script that emailed me as a way of verifying that the code worked, and while some condition was true, I forgot to exit the loop. It turned out those e-mails got sent faster than once per second. As you can imagine, I got about 500 e-mails before I realized what was going on. I am infinitely grateful for that little lesson!

When you're done laughing at my story, remember, when you're writing loops, it's a good idea to take a moment to **consider the different values a variable can**. This helps you make sure your loop won't get stuck. If you see that your program is running forever without finishing, have a second look at your loops to check there's no infinite loop hiding somewhere in the code.

While you need to watch out for infinite loops, they are not always a bad thing. Sometimes you actually want your program to execute continuously until some external condition is met. If you've used the **ping** utility on Linux or macOS system, or **ping-t** on a Windows system, you've seen an infinite loop in action.

This tool will keep sending packets of data and printing the results to the terminal unless you send it the interrupt signal, usually pressing Ctrl + C. If you were looking at the program source code, you'll see that it uses an infinite loop to do this with a block of code with instructions to keep sending the packets forever.

One thing to call out is **it should always be possible to break the loop** by sending a certain signal. In the ping example, that signal is the user pressing Ctrl + C. In other cases, it could be that the user pressed the button on a graphical application, or that another program sent a specific signal, or even that a time limit was reached.

In your code, you could have an infinite loop that looks something like this:

```
while True:
    do_something_cool()
    if user_requested-to_stop():
        break
```

In Python, we use the **break** keyword which you can see in the above script to signal that the current loop should stop running. We can use it not only to stop infinite loops but also to stop a loop early if the code has already achieved what's needed. So a quick refresh.

How do you avoid the most common pitfalls when writing while loops?

First, remember to initialize your variables, and second, check that your loops won't run forever.

10.8. Infinite loops and Code Blocks

Another easy mistake that can happen when using loops is introducing an infinite loop. An infinite loop means the code block in the loop will continue to execute and never stop. This can happen when the condition being evaluated in a *while* loop doesn't change.

Pay close attention to your variables and what possible values they can take. Think about unexpected values, like zero.

In one of my code blocks, I sometimes see an error message that reads "Evaluation took more than 5 seconds to complete." This means that the code encountered an infinite loop, and it timed out after 5 seconds. Then I take a closer look at the code and variables to spot where the infinite loop is.

Wow! All this talk of loops is making me feel a little dizzy right now. I'm going to have to go and lie down while you do the next practice quiz. Best of luck! Meet me over in the next section when you're done.

10.9. Practice Quiz 7: 5 Questions

1. What are while loops in Python? – 1 point

While loops let the computer execute a set of instructions while a condition is true.

While loops instruct the computer to execute a piece of code a set number of times.

While loops let us branch execution on whether or not a condition is true.

While loops are how we initialize variables in Python.

2. Fill in the blanks to make the `print_prime_factors` function print all the prime factors of a number. A prime factor is a number that divides another without a remainder. - 1 point

Start with two, which is the first prime factor.

Keep going until the factor is larger than the number.

Check if factor is a divisor of number.

If it is, print it and divide the original number.

```
number = number / factor
```

```
# If it's not, increment the factor by one.
```

```
# Should print 2,2,5,5.
```

```
# DO NOT DELETE THIS COMMENT.
```

3. The following code can lead to an infinite loop. Fix the code so that it can finish successfully for all numbers. Note: Try running your function with the number 0 as the input, and see what you get! - 1 point

```
# Check if the number can be divided by two without a remainder
```

```
# If after dividing by two the number is 1, it's a power of two
```

4. Fill in the empty function so that it returns the sum of all the divisors of a number, without including it. A divisor is a number that divides into another without a remainder. - 1 point

—

Return the sum of all divisors of n, not including n

—

—

—

—

0

sum of 1

1

sum of 1+2+3+4+6+9+12+18

55

114

5. The `multiplication_table` function prints the results of a number passed to it multiplied by 1 through 5. An additional requirement is that the result is not to exceed 25, which is done with the `break` statement. Fill in the blanks to complete the function to satisfy these conditions.

```
# Initialize the starting point of the multiplication table
```

```
# Only want to loop through 5
```

```
    result = number * multiplier
```

```
# What is the additional condition to exit out of the loop?
```

```
break
```

```
# Increment the variable for the loop
```

```
# Should print: 3x1=3 3x2=6 3x3=9 3x4=12 3x5=15
```

```
# Should print: 5x1=5 5x2=10 5x3=15 5x4=20 5x5=25
```

```
# Should print: 8x1=8 8x2=16 8x3=24
```

10.9.1. Answers to Practice Quiz 7

1. A. Using while loops we can keep executing the same group of instructions until the condition stops being true.

2.

```
# Start with two, which is the first prime
```

```
# Keep going until the factor is larger than the number
```

```
# Check if factor is a divisor of number
```

```
# If it is, print it and divide the original number
```

```
    number = number / factor
```

```
# If it's not, increment the factor by one
```

```
# Should print 2,2,5,5  
# DO NOT DELETE THIS COMMENT
```

3.

```
# Check if the number can be divided by two without a remainder  
r
```

```
# If after dividing by two the number is 1, it's a power of two
```

4.

```
# Return the sum of all divisors of n, not including n
```

```
# 0  
# 1  
# 55  
# 114
```

5.

```
# Initialize the starting point of the multiplication table
```

```
# Only want to loop through 5
```

```
    result = number * multiplier
```

```
# What is the additional condition to exit out of the loop?
```

```
break
```

```
# Increment the variable for the loop
```

```
# Should print: 3x1=3 3x2=6 3x3=9 3x4=12 3x5=15
```

```
# Should print: 5x1=5 5x2=10 5x3=15 5x4=20 5x5=25
```


Should print: $\delta x_1=8$ $\delta x_2=16$ $\delta x_3=24$

11. For Loops

11.1. What is a For Loop?

Okay, how are you doing? If all this talk of loops is starting to make your head spin, remember that there's nothing wrong with looping back around and reviewing what you've learned. It's the quickest way to stop feeling like you're running in circles. All right, feeling good? Great, Then we're ready for a different type of loop. In this video, we're going to meet the *for* loop.

A for loop **iterates over a sequence of**

A very simple example of a for loop is to iterate over a sequence of numbers, like this:

Video 21 (3:03 *What is a for loop?*)

```
for x in range(5):
```

Notice how the structure is kind of similar to the structures we've already seen. The first line indicates the distinguishing keyword. In this case, that's `for` and it ends with a colon. The body of the loop is indented to the right, like we saw in the **while** loop, the **if** block, and the **function** definitions.

The difference in this case is that we have the keyword `in`. Also, between the *for* keyword and *in* keyword, we have the name of a

variable. This variable will take each of the values in the sequence that loop iterates through. So in this example, it'll iterate through a sequence of numbers generated using the range function.

There are two important things I want to call out about this range function. First, in Python and a lot of other programming languages, a range of numbers will start with the value 0 by default. Second, the list of numbers generated will be one less than the given value.

In the simple example here, x will take the values 0, 1, 2, 3, and 4. Let's check this out (run this program yourself):

```
for x in range(5):  
    print(x)
```

```
0  
1  
2  
3  
4
```

So there, we have a very basic for loop. It iterates over a sequence of numbers generated by the range function. When using a for loop, we point the variable defined between *for* and *in* in this case, x, at each element of the sequence. This means on the first iteration x points at 0.

On the second iteration, it points at 2, and so on. Whatever code we put in the body of the loop will be executed on each of the values, one value at a time. As we said earlier, the loops body can do a lot of things with the values it iterates.

For example, you could have a function to calculate the square of a number, and then use a for loop to sum the squares of the numbers in a range. Iterating over numbers looks very similar to the while loop examples we showed before. So you may be wondering why have two loops that look like they do the same thing?

Well, the power of the **for loop** is that we can use it to iterate over a sequence of **values of any** not just a range of numbers. Think all the way back to our very first Python example in this course. Remember our trusty hi friends script? In it, we saw a for loop that iterated over a list of strings. It looks like this.

```
1 friends = ["Taylor", "Alex", "Pat", "Eli"]  
  
2 for friend in friends:  
3     print("Hi " + friend)
```

We'll talk a lot more about lists later on. But for now, you only need to know that we can construct lists using square brackets, and **separate the elements in them with** In this example, we're iterating a list of strings, and for each of the strings in the list, were printing a greeting.

The sequence that the for loop iterates over could contain any type of element, not just strings. For example, we could iterate over a list of numbers to calculate the total sum and average. Here's one way of doing this:

```
values = [23, 52, 59, 37, 48]
sum = 0
length = 0
for value in values:
    sum += value
    length += 1
print("total sum: " + str(sum) + " - Average: " + str(sum/length))
```

Here, we're defining a list of values. After that, we're initializing two variables, **sum** and **length**, that will update in the body of the for loop. In the for loop, we're iterating over each of the values in the list, adding the current value to the sum of values, and then also adding 1 to length, which calculates how many elements there are in the list.

Once we've gone through the whole list, we print out the sum and the average. We'll keep using for loops in our examples every time we want to iterate over the elements of any sequence and operate with them.

Some examples of sequences that we can iterate are:

- the files in a directory
- the lines in a file

the processes running on a machine.

And there's a bunch of others. So as an IT specialist, you'll use for loops to automate tons of stuff. For example, you might use them to

copy files to machines
process the contents of files
automatically install software

and a lot more.

A few weeks ago, I had to update a lot of files with different values depending on their contents. So I used a for loop in a script to iterate over all the files. Then, my script took different actions based on an **if** condition and updated all of those files for me.

It would have taken me forever if I had done this manually file by file. If you're wondering when you should use for loops and when you should use while loops, here's a way to tell:

Use **for** loops when there's a **sequence of elements** that you want to iterate.

Use **while** loops when you want to **repeat an action** until a condition changes.

And if whatever you're trying to do can be done with either for or while loops, just use whichever one's your favorite. I'm more of a while gal myself, but it's totally your call.

In section 11.3, I've put together more examples to help get you more practice with for loops and discover some of the cool things you could do with them.

11.2. For Loops Recap

For loops allow you to iterate over a sequence of values. Let's take the example from the beginning of video 21 (section 11.1):

```
for x in range(5):
```

```
    print(x)
```

Similar to *if* statements and *while* loops, *for* loops begin with the keyword ***for*** with a colon at the end of the line. Just like in function definitions, *while* loops and *if* statements, the body of the *for* loop begins on the next line and is indented to the right.

But what about the stuff in between the *for* keyword and the colon? In our example, we're using the *range()* function to create a sequence of numbers that our *for* loop can iterate over. In this case, our variable *x* points to the current element in the sequence as the *for* loop iterates over the sequence of numbers.

Keep in mind that in Python and many programming languages, a range of numbers will start at 0, and the list of numbers generated will be *one less* than the provided value. So *range(5)* will generate a sequence of numbers from 0 to 4, for a total of 5 numbers.

Bringing this all together, the *range(5)* function will create a sequence of numbers from 0 to 4. Our *for* loop will iterate over

this sequence of numbers, one at a time, making the numbers accessible via the variable `x` and the code within our loop body will execute for each iteration through the sequence. So for the first loop, `x` will contain 0, the next loop, 1, and so on until it reaches 4. Once the end of the sequence comes up, the loop will exit and the code will continue.

The power of *for* loops comes from the fact that it can iterate over a sequence of any kind of data, not just a range of numbers. You can use *for* loops to iterate over a list of strings, such as usernames or lines in a file.

Not sure whether to use a *for* loop or a *while* loop? Remember that a *while* loop is great for performing an action over and over until a condition has changed. A *for* loop works well when you want to iterate over a sequence of elements.

11.3. More for Loop Examples

In the last section, we talked about the range function, and how it generates a sequence of numbers starting with zero. Sometimes, though, we don't want to start with zero. For these situations, the range function also allows us to specify the first element of the list to generate. We do that by passing two parameters to the function instead of one, like in this example:

Video 22 (2:01 *What is a for loop?*)

```
product = 1
for n in range(1, 10):
    product = product * n
print(product)
362880
```

In this example, we're calculating the products of all numbers from 1 to 10. For this operation, it's important that we start with 1 and not with 0. If we'd started with 0, the whole product would be zero.

Additionally, we can specify a third parameter to change the size of each step. This means that instead of going one by one, we could have a larger difference between the elements. Let's check out this example when you might want to do something like this.

```
def to_celsius(x):
```

```
return (x-32)*5/9
```

```
for x in range(0,100,10):  
    print(x, to_celsius(x))
```

First, we're defining a function that converts a temperature value from Fahrenheit to Celsius, and we're simply using a conversion formula to do that. Then we have a *for* loop that starts at zero, and goes up to 100 in steps of 10. Notice that we're using 101 for the upper limit instead of 100. We're doing this because the range never includes the last element, and we want to include 100 in our range.

The body of the for-loop prints the value in Fahrenheit and the value in Celsius, creating a conversion table. Let's run the program to see this in action:

```
0 -17.77777777777778  
10 -12.222222222222221  
20 -6.666666666666667  
30 -1.1111111111111112  
40 4.4444444444444445  
50 10.0  
60 15.555555555555555  
70 21.111111111111111  
80 26.666666666666668  
90 32.222222222222222  
100 37.77777777777778
```

That example got you feeling the heat? Don't worry, there's a quick rundown of what we've learned. The range function can receive one, two or three parameters. If it receives one parameter, it will create a sequence one by one from zero until one less than the parameter received. If it receives two parameters, it will create a sequence one by one from the first parameter until one less than the second parameter.

Finally, if it receives three parameters, it will create a sequence starting from the first number and moving towards the second number. But this time, the jumps between the numbers will be the size of the third number, and again, it will stop before the second number.

Sound like a lot to remember, but don't panic. As we've said before, you don't have to try to memorize it all, just keep practicing. It'll soon become second nature. To help you practice, we've included all of this in a handy cheat sheet to refer to whenever you need it. You'll find that in section 11.7.

11.4. A Closer Look at the `Range()` Function

Previously we had used the `range()` function by passing it a single parameter, and it generated a sequence of numbers from 0 to one less than we specified. But the `range()` function can do much more than that.

We can pass in two parameters: the first specifying our starting point, the second specifying the end point. Don't forget that the sequence generated won't contain the last element; it will stop one before the parameter specified.

The `range()` function can take a third parameter, too. This third parameter lets you alter the size of each step. So instead of creating a sequence of numbers incremented by 1, you can generate a sequence of numbers that are incremented by 5.

To quickly recap the `range()` function when passing one, two, or three parameters:

One parameter will create a sequence, one-by-one, from zero to one less than the parameter.

Two parameters will create a sequence, one-by-one, from the first parameter to one less than the second parameter.

Three parameters will create a sequence starting with the first parameter and stopping before the second parameter, but this

time increasing each step by the third parameter.

11.5. Nested For Loops

You're doing great getting your head around all these loops. I think you're ready for something a little bit more complex. We're going to explore what happens when you get loops inside of loops. Does that make your head spin? Don't worry, we're about to break it down for you with a couple of examples.

Have you ever played dominoes before? There's a bunch of fun games you can play with these tiles. In case you're not familiar, each Domino Tiles has two numbers represented by a collection of dots carved on each half of the tile. The numbers go from zero to six. Tiles can be rotated so that each combination of numbers is represented only once in a set of Domino Tiles. In other words, the two three tile is the same as the three two tile, and there's only one per set.

Now, imagine we wanted to write a program that prints each Domino Tile in a set. If we take all of the tiles that have zero on the left, we can print tiles with numbers from zero to six on the right. That should be easy to do with a four loop.

But what about tiles that have one on the left? Well, we need to skip the one zero tile, because that one was already printed as zero one. So we can print a list of tiles with one on the left and numbers from one to six on the right.

When we look at two, we would need to skip both zero and one, and so on. Are you following along? How you think we'd write

the code for this? Turning this into code means that we'd need to write two *for* loops, one inside the other. This is what we call ***nested for*** loops. Check out how this looks on Python code:

Video 23 (3:26 *Nested for loops*)

```
for left in range(7):
```

```
    for right in range (left, 7):
```

```
        print("[ " + str(left) + " | " + str(right) + " ]", end= "
```

```
print()
```

```
[0|0] [0|1] [0|2] [0|3] [0|4] [0|5] [0|6]
```

```
[1|1] [1|2] [1|3] [1|4] [1|5] [1|6]
```

```
[2|2] [2|3] [2|4] [2|5] [2|6]
```

```
[3|3] [3|4] [3|5] [3|6]
```

```
[4|4] [4|5] [4|6]
```

```
[5|5] [5|6]
```

```
[6|6]
```

In this code, we're using a new parameter that we passed to the print function. This parameter is called Normally, once print has taken the content we passed and written it to the screen, then it writes a special character that creates a new line called the **newline** character. If we want print to write something else instead of the newline character, we use the **end** parameter, like we see in this example.

Notice how the second for loop iterates over a different number of elements each time it's called as the value of left changes.

Depending on what you want to achieve with your nested loops, you may want both loops to always go through the same number of elements, or you might want the second loop to connect to the first one. Let's look at a different example.

Let's say you run a local girl's basketball league in your town. You have four teams that will play against each other in the league, both at home and away. You've stored the names of the teams in a list, like this:

```
teams = ['Dragons', 'Wolves', 'Pandas', 'Unicorns']
```

We want to write a script that will output all possible team pairings. For this, the order of the names matters because for each game, the first name will be the **home** team and the second name is the **away** team. Of course, what we don't want to do is have a team playing against itself. So what statement do we need to use to avoid that?

To do this, we need to use a conditional that makes sure we only print the pairing when the names are different. Check out what this looks like:

```
teams = ['Dragons', 'Wolves', 'Pandas', 'Unicorns']
for home_team in teams:
    for away_team in teams:
        if home_team != away_team:
            print(home_team + " vs " + away_team)
```

Dragons vs Wolves
Dragons vs Pandas
Dragons vs Unicorns
Wolves vs Dragons
Wolves vs Pandas
Wolves vs Unicorns
Pandas vs Dragons
Pandas vs Wolves
Pandas vs Unicorns
Unicorns vs Dragons
Unicorns vs Wolves
Unicorns vs Pandas

Success! As you can see, nested loops are super useful for solving certain problems, like pairing teams. What it doesn't solve is the question, who would win in a face-off between dragons and unicorns? If only there were some code for that!

Anyway, we've seen that nested loops are a handy tool, but we need to be careful not to just blindly apply them to any problem. Why? Well, because the longer the list your code needs to iterate through, the longer it takes your computer to complete the task.

Let's say your manager asks you to do an operation that will run through a list of 10,000 elements. If the operation takes one millisecond per element, the whole loop would take one millisecond times 10,000 to complete, which is 10 seconds.

Now, imagine we add a nested loop that has to go over the same 10,000 elements. This means that each iteration of the

outside loop would do a full iteration of the inside loop, which again, would take ten seconds to go through the whole list. So, now the whole iteration takes 10,000 times 10 seconds, which is 100,000 seconds, that's over 27 hours! I have the patience of a gnat, so that would definitely not work for me.

This doesn't mean we shouldn't use nested loops. They are a useful tool when solving problems that require them, but we need to be careful of where and how we use them. Throughout this course, and one is coming up (Part 2), we'll look at a lot of techniques that can help us pick the right tool to use for each type of problem.

Up next, we'll look into some common errors that you might come across when writing your for loops and what to do about them.

11.6. Common Errors in For Loops

We've now seen how to write for loops, combine them with functions, nest a for loop inside a different loop, and even combine a nested loop with conditionals. Nice job, you're chugging right along. But before we're done with for loops, let's check out some common mistakes you may come across while trying this yourself.

As we've called out are ready, for loops iterate over sequences. The interpreter will refuse to iterate over a single element. As you see here:

Video 24 (1:57 *Common Errors in for Loops*)

```
for x in 25:  
    print(x)
```

Traceback (most recent call last):

```
File "D:\ajwright\Documents\Python  
Programming\My_Programs\Book Jotter.py", line 1, in  
    for x in 25:  
TypeError: 'int' object is not iterable
```

In this example, we're trying to iterate over the number 25. Python prints a type error telling us that integers are not iterable. There are two solutions to this problem, depending on what we're trying

to do. If we want to go from 0 to 25, then we use the range function, so,

```
for x in range(25):  
    print(x)
```

0

1

2

3

4

5

6

7

8

9

10

11

12

13

14

15

16

17

18

19

20

21

22

23

24

But if we're trying to iterate over a list that has 25 as the *only* element, then it needs to be a list, and that means writing it between square brackets as follows.

```
for x in [25]
    print(x)
```

25

You might be wondering why would you ever want to iterate over a list of one element, and that's a good question. Well, this kind of issue usually happens when you have a function with a for loop inside it, which is iterating over the elements of a list received by parameter. Say for example, you have a function that fixes the permissions of a list of files received by parameter, and you want to call this function to fix the permissions of just one specific file.

To do that, you need to pass the file as the single element of a list. Let's check this out with some code we're familiar with, our friendliest of Python examples. We're going to modify it to have the greetings inside a function:

```
def greet_friends(friends)
    for friend in friends:
        print("Hi " + friend)
greet_friends(['Taylor', 'Luisa', 'Jamaal', 'Eli'])
```

Hi Taylor

Hi Luisa

Hi Jamaal

Hi Eli

We've defined a **greet_friends** function that receives a list by parameter and iterates over that list, greeting each friend. But what if we only want to greet one friend instead of four? Well, we still need to define a list, but with only one element.

But first, let's see what would happen if we don't do that:

```
greet_friends("Barry")
```

Hi B

Hi a

Hi r

Hi r

Hi y

Not what we expected, right? Well, what's going on here? This happens because strings are iterable, the for loop will go over each letter of the string and do the operation we asked it to do, which in this case is print a greeting. Depending on what you're trying to do, you may actually want to iterate through the letters of a string. But in this case, we don't.

So to sum it up, if you get an error that a certain type is not iterable, you need to make sure the for loop is using a sequence

of elements and not just one, and if you find your code iterating through each letter of a string when you want it to do it for the whole string, you probably want to have that string be a part of a list.

We've now learned how to write while loops and for loops. You might remember, for loops are best when you want to iterate over a known sequence of elements but when you want to operate while a certain condition is true, while loops are the best choice.

Next up, we've got a super useful cheat sheet for you that puts all this into one handy resource. After that, head over to the practice quiz 8 to test your knowledge and check in on how you're doing.

11.7 Cheat Sheet 3: Loops

To locate the *Loops Cheat Sheet* file, open the “Cheat Sheets” folder. It’s one of the folders inside the course resources folder you downloaded earlier.

11.8. Practice Quiz 8: 4 Questions

1. How are *while* loops and *for* loops different in Python? – 1 point

While loops can be used with all data types, for loops can only be used with numbers.

For loops can be nested, but while loops can't.

While loops iterate while a condition is true, for loops iterate through a sequence of elements.

While loops can be interrupted using `break`, for loops using `continue`.

2. Fill in the blanks to make the factorial function return the factorial of `n`. Then, print the first 10 factorials (from 0 to 9) with the corresponding number. Remember that the factorial of a number is defined as the product of an integer and all integers before it. For example, the factorial of five (5!) is equal to $1*2*3*4*5=120$. Also recall that the factorial of zero (0!) is equal to 1. 1 point

3. Write a script that prints the first 10 cube numbers (x^{**3}), starting with $x=1$ and ending with $x=10$. - 1 point

4. Write a script that prints the multiples of 7 between 0 and 100. Print one multiple per line and avoid printing any numbers that aren't multiples of 7. Remember that 0 is also a multiple of 7. - 1 point

11.8.1. Answers to Practice Quiz 8

1. C. We can use while loops when we want our code to execute repeatedly while a condition is true, and for loops when we want to execute a block of code for each element of a sequence.

2.

3.

4.

12. Recursion (Optional)

12.1. What is recursion?

Welcome back. How are you feeling after the last quiz? You're starting to learn some pretty cool things that you could do in your code. Who knew loops can be so fascinating?

You've now discovered two looping techniques that you could use in Python: **while loops** and **for**. We use while loops when we want to do an operation repeatedly while a certain condition is true. We use for loops when we want to iterate over the elements of a sequence.

Now, we're going to check out a third technique called **recursion**. But before we dive in, you may have noticed that this chapter is marked as optional. That's because while recursion is a very common technique used in software engineering, it's not used that much in automation.

Still, we think it's valuable for you to know about recursion and to have an idea of how to use it. You may see it in code written by others or you may face a problem where recursion is the best way to solve it. So while the next few sections are optional, it's still super valuable stuff. Of course, feel free to skip them if you'd just rather focus on the other concepts. Let's dive in.

Video 25 (1:11) *What is Recursion?*

Recursion is the repeated application of the same procedure to a smaller

Have you ever played with a Russian nesting doll? They are a great visual example of recursion. Each doll has a smaller doll inside it. When you open up the doll to find the smaller one inside, you keep going until you reach the smallest doll which can't be opened.

Recursion let's us tackle complex problems by reducing the problem to a simpler

Take our Russian nesting dolls, all nested inside each other. Imagine we want to find out how many dolls there are in total. We would need to open each doll one by one until we got to the last one and then count how many dolls we've opened. That's recursion in action.

Here's another example with a more complex problem. Imagine you're in a line of people and you want to know how many people are in front of you. If the line is long, it might be hard to count the people without leaving the line and losing your place.

Instead you can ask the person in front of you how many people are in front of them. Since this person will be in the same situation as you, they'll have to ask the same question to the person in front of them and so on and so on until the question reaches the first person in the line. This person can confidently reply that there are no people in front of them. So then the

second person in line can reply one, the person behind them replies two, and so on until the answer reaches you.

Okay. I know the chances are pretty small that all of those people would play along just so you can know where you are in line, but it's a useful way to visualize how recursion works. How does this translate into programming? Well, in programming, recursion is a way of doing a repetitive task by having a function call itself.

A **recursive function** calls itself usually with a modified parameter until it reaches a specific condition. This condition is called the **base**. In our earlier examples, the base case would be the smallest Russian doll or the person at the front of the queue. Let's check out an example of recursive function to understand what we're talking about.

```
def factorial(n):  
    if n < 2:  
        return 1  
    return n*factorial(n-1)
```

Here, we're defining a function called factorial. At the beginning of the function, we have a conditional block defining the base case, where n is smaller than 2. It simply returns the value 1. After the base case, we have a line where the factorial function is calling itself with n minus 1. This is called the **recursive**. This creates a loop.

Each time the function is executed, it calls itself with a smaller number until it reaches the base case. Once it reaches the base case, it returns the value 1. Then the previously called function multiplies that by two and the previously called function multiplies it by three and so on.

This loop will keep going until the first factorial function called returns the desired result. It's a bit complex. Let's add a few print statements in our Pycharm IDE to see exactly how this works:

```
def factorial(n):
    print("Factorial called with " + str(n))
    if n < 2:
        print("Returning 1")
        return 1
    = n*factorial(n-1)
    print("Returning " + str(result) + " for factorial of " + str(n))
    return result
factorial(4)
```

```
Factorial called with 4
Factorial called with 3
Factorial called with 2
Factorial called with 1
Returning 1
Returning 2 for factorial of 2
Returning 6 for factorial of 3
Returning 24 for factorial of 4
```

So the result of factorial (4) is 24. Here we can see the function kept calling itself until it reached the base case. After that, each function returned the value of the previous function multiplied by n until the original function returned. Cool, huh?

Next up, we're going to check out some more examples of when to use recursion and when it's best to avoid it.

12.2. Recursion in Action in the IT Context

By now you've seen what a recursive function looks like, how to write a base case and the recursive case. You might be wondering why do we need recursive functions if I can just use a for or while loop? Well, solutions to some specific problems are easier to write and understand when using recursive functions.

A lot of math functions like the factorial or the sum of all the previous numbers are good examples of this. If a math function is already defined in recursive terms, it's straightforward to just write the code as a recursive function. But it's not just about math functions. Let's check out a couple of examples of how this could help an IT specialist trying to automate tasks.

Let's say that you need to write a tool that goes through a bunch of directories (folders) in your computer and calculates how many files are contained in each. When listing the files inside a directory, you might find subdirectories inside them and you'd want to count the files in those subdirectories as well. This is a great time to use recursion.

The base case would be a directory with no subdirectories. For this case, the function would just return the amount of files. The recursive case would be calling the recursive function for each of the contained subdirectories. The return value of a given function call would be the sum of all the files in that directory plus all the files in the contained subdirectories.

A directory of files that can contain other directories is an example of a recursive

Because directories can contain subdirectories that contain subdirectories that contain subdirectories, and so on. When operating over recursive structure, it's usually easier to use recursive functions than *for* or *while* loops.

Another IT-focused example of a recursive structure is anything that deals with groups of users that can contain other groups. We see this situation a lot when using administrative tools like active directory or LDAP (Lightweight Directory Access Protocol).

Say your group management software allows you to create groups that have both users and other groups as their members, and you want to list all human users that are part of a given group. Here you would use a recursive function to go through the groups.

The base case would be a group that only includes users listing all of them. The recursive case would mean going through all the groups contained listing all the users in them and then listing any users contained in the current group.

It's important to call out that in some languages there's a maximum amount of recursive calls you can use. In Python by default, **you can call a recursive function 1,000 times until you reach the** That's fine for things like subdirectories or user groups that aren't thousands of levels deep.

But it might not be enough for mathematical functions like the ones we saw in section 12.1. Let's go back to our factorial example from section 12.1 and try to call it with n equals 1,000:

```
def factorial(n):
    print("Factorial called with " + str(n))
    if n < 2:
        print("Returning 1")
        return 1
    result = n*factorial(n-1)

    print("Returning " + str(result) + " for factorial of " + str(n))
    return result
factorial(1000)
```

Traceback (most recent call last):

```
File "D:\ajwright\Documents\Python
Programming\My_Programs\Book Jotter.py", line 9, in
    factorial(1000)
File "D:\ajwright\Documents\Python
Programming\My_Programs\Book Jotter.py", line 6, in factorial
    result = n*factorial(n-1)
File "D:\ajwright\Documents\Python
Programming\My_Programs\Book Jotter.py", line 6, in factorial
    result = n*factorial(n-1)
File "D:\ajwright\Documents\Python
Programming\My_Programs\Book Jotter.py", line 6, in factorial
    result = n*factorial(n-1)
[Previous line repeated 993 more times]
File "D:\ajwright\Documents\Python
Programming\My_Programs\Book Jotter.py", line 2, in factorial
```

```
print("Factorial called with " + str(n))
```

RecursionError: maximum recursion depth exceeded while calling a Python object

Do you see that recursion error (last line) above? It's telling us that we've reached the maximum limit for recursive calls. So while you can use recursion in a bunch of different scenarios, we only recommend using it when you need to go through a recursive structure that won't reach a thousand nested levels.

All right, I've just added recursion to your growing scripting tool box. They're ready for you whenever the situation calls for it.

12.3. Additional Recursion Sources

In the past sections, we visited the basic concepts of recursive functions. A recursive function must include a **recursive case** and **base**. The recursive case calls the function again, with a different value. The base case returns a value without calling the same function.

A recursive function will usually have this structure:

```
recursive_function(modified_parameters)
```

For more information on recursion, check out this resource:

[Wikipedia Recursion](#)

12.4. Practice Quiz 9: 5 Questions

1. What is recursion used for? – 1 point

Recursion is used to create loops in languages where other loops are not available.

We use recursion only to implement mathematical formulas in code.

Recursion is used to iterate through sequences of files and directories.

Recursion lets us tackle complex problems by reducing the problem to a simpler one.

2. Which of these activities are good use cases for recursive programs? Select all that apply. - 1 point

Going through a file system collecting information related to directories and files.

Creating a user account.

Installing or upgrading software on the computer.

Managing permissions assigned to groups inside a company, when each group can contain both subgroups and users.

Checking if a computer is connected to the local network.

3. Fill in the blanks to make the `is_power_of` function return whether the number is a power of the given base.

base is assumed to be a positive number. for functions that return a boolean value, you can return the result of a comparison.

- 1 point

```
# Base case: when number is smaller than base.
```

```
# If number is equal to 1, it's a power (base**0).
```

```
number/=base # This is equivalent to number = number / base
```

```
# Recursive case: keep dividing number by base.
```

```
should be True
```

```
should be True
```

```
should be False
```

4. The `count_users` function recursively counts the amount of users that belong to a group in the company system, by going through each of the members of a group and if one of them is a group, recursively calling the function and counting the members. But it has a bug! Can you spot the problem and fix it? - 1 point

```
# count += 1
```

```
count += count_users(member)
```

5. Implement the `sum_positive_numbers` function, as a recursive function that returns the sum of all positive numbers between the number `n` received and 1. For example, when `n` is 3 it should return $1+2+3=6$, and when `n` is 5 it should return $1+2+3+4+5=15$. -

1 point

12.4.1. Answers to Practice Quiz 9

1. D. By reducing the problem to a smaller one each time a recursive function is called, we can tackle complex problems in simple steps.

2. A, D.

A is a correct answer because directories can contain subdirectories that can contain more subdirectories, going through these contents is a good use case for a recursive program.

D is also a correct answer because the groups can contain both groups and users, this is the kind of problem that is a great use case for a recursive solution.

3.

Base case: when number is smaller than base.

If number is equal to 1, it's a power ($\text{base}^{**}0$).

```
number/=base
```

```
# Recursive case: keep dividing number by base.
```

```
should be True
```

```
should be True
```

```
should be False
```

4.

```
# count += 1
```

```
count += count_users(member)
```

5.

13. Module Review

13.1. Loops Wrap Up

Wow, we've come a long way and you've learned a lot already. Now's a good time to stop and give yourself a big pat on the back. In this module, we've looked at **ways we can use to tell a computer to do an action** Python gives us three different ways to perform repetitive tasks: while loops, for loops, and recursion.

We use **while loops** when we want to **do an operation while a certain condition is true or alternatively until it becomes**

We use **for loops** when we want to **iterate over the elements of the sequence or a range of**

We use **recursion** when the **problem is best solved in smaller steps and then combining those steps towards a larger**

If you're still not sure which is the best tool to choose for a specific problem don't worry, that's normal. As you keep practicing your automation skills, choosing between one option and another will become natural.

So next time you find yourself doing the same or similar things over and over again, that's your call to see if you can use a loop to get your computer to do the work for you.

Up next it's test time again, with the next graded assessment. Like always remember you can take as much time as you need

before taking the assessment. Go at your own pace, review everything we've covered, and practice the examples. So there's no chance loops will ever throw you for a loop!

13.2. Module 3 Graded Assessment – 10 Questions

It's time again for your next graded assessment. Open the "Graded Assessments" folder you downloaded earlier. It contains the pdf files of all the graded assessments in this course. Here's the file name to search:

Module 3 Graded Assessment – *file name*

13.2.1. Solutions to Module 3 Graded Assessment:

I (or any member of my team) is available to help you grade your assessments. You can use my help link (email) at the end of chapter 25 to send your assessment for grading. We will get back to you in 12 to 24 hours with your result.

However, if you cannot wait, you can open the “Graded Assessments” folder you downloaded earlier. It contains the pdf formats of the solutions to all graded assessments in this course. You can use them to grade your assessments by yourself. Just be honest as you grade. Here’s the file name to search:

Module 3 Graded Assessment Solutions – *file name*

Module 4

If you're not a string, at least your name is! I'm sure I'm glad I did not lose you. You're still in my list and dictionary.

14. Strings

14.1. Basic Structures Introduction

Welcome back and congratulations on getting this far. I'm sure I'm glad we didn't lose you and all those loops we covered in the last module. You're doing great and making tons of progress.

In earlier sections, we covered the basic elements of Python syntax. We talked about how to define functions, how to make your computer act differently based on conditionals, and how to make it perform operations repeatedly using while, and for loops, and recursion.

Now that we have the basics of syntax out of the way, we can start growing our Python knowledge which will let us do more and more interesting operations. Remember, one of our main goals in this course is to help you learn to write short Python scripts that automate actions, you've made big steps towards getting there.

In the upcoming sections, you're going to learn a bunch of new super useful skills to add to your programming toolbox. We'll check out some datatypes provided by the Python language to help us solve common problems with our scripts. In particular, will do a deep dive into and

Heads-up, while we've used strings in our scripts already, we barely scratched the surface of all the things we can do with them in Python. We also ran into a few lists in some examples but there's a lot more of them we haven't seen yet.

Dictionaries are a whole **new datatype** for us to dig our teeth into. These are all data types or data structures that are super flexible. We're going to use them to write all kinds of scripts in Python. So it's a good idea to spend some time getting to know them, and learning when to use them, and how to make the most out of them.

We've got a lot of new and exciting concepts to discover. So let's get right to it.

14.2. What is a string?

By now, we've used strings in a lot of examples, but we haven't spent time looking at them in detail yet. Before we dive into the nitty-gritty though, let's go over what we've seen so far and add a few more points. First, a quick refresher.

A string is a data type in Python that's used to represent a piece of text. It's written between quotes, either double quotes or single quotes, your

Video 26 (1:01 *What is a string?*)

It doesn't matter which type of quotes you use as long as they match. If we mix up double and single quotes, Python won't be too happy:

```
name = 'Sasha'  
color = 'gold'  
place = "Cambridge"
```

```
File "", line 1 place = "Cambridge"  
SyntaxError: EOL while scanning string literal.
```

As you can see it returns a syntax error, telling us it couldn't find the end of the string.

A string can be as short as zero characters, usually called an **empty string** or really long. We also learned that we can use strings to build longer strings using the plus sign and action called like this:

```
Print("Name: " + name + ", Favorite color: " + color)
```

```
Name: Sasha, Favorite color: gold
```

A less common operation is to multiply the string by a number, which multiplies the content of the string that many times, like this:

```
"example" * 3
```

```
'exampleexampleexample'
```

If we want to know how long a string is, we can use the **len** function which we saw in earlier sections. The len function tells us the number of characters contained in the string, for example, since name = Sasha,

```
len(name)
```

5

We can use strings to represent a lot of different things. They can hold a username, the name of a machine, an email address, the

name of a file, and any other text. A lot of the data that we'll interact with will be stored in strings, so it's important to know how to use them.

There are tons of things we could do with strings in our scripts. For example, we can check if files are named a certain way by looking at the filename and seeing if they match our criteria, or we can create a list of emails by checking out the users of our system and concatenating our domain.

I recently wrote a script that worked with a bunch of files and took different actions according to the name of each file. So the file ended in a certain extension say, .TXT , then my script would print it. If the file had a certain string and the name, say, test, then my script would ignore it and move on to the next thing and so on. The contents of a text file are also strings.

A few months ago, I had to change the default values for a bunch of configuration options from true to false. So I wrote a function that would find the string “true” in a file and replace it with “false”.

You can probably think of more examples where your code needs to handle strings, but to use strings effectively, we need to know what options are available to us in Python.

In the next few sections, we'll cover some of the operations we can perform over strings, including how to access parts of them and modify them.

14.3. The Parts of a String

When we first came across the for loop, we called out that we can iterate over a string character by character. But what if we want to access just a specific character or characters? We might want to do this, for example, if we have a text that's too long to display and we want to show just a portion of it, or if we want to make an acronym by taking the first letter of each word in a phrase.

We can do that through an operation called **string**. This operation lets us access the character in a given position or index using square brackets and the number of the position we want. Like this:

Video 27 (2:57 *The parts of a String*)

```
name = 'Jaylen'  
print(name[1])
```

a

This might seem confusing at first, like Python is acting up. We're asking for the first character, and it's giving us the second. What gives Python? Well, what's happening here is that **Python starts counting indexes from 0 not 1**. Just like it does with the range function. So if we want the first character, we need to access the one at index 0.

```
print(name[0])
```

```
}
```

Knowing that indexes start at 0, which one do you think will be the last index in the string? It'll always be one less than the length of the string. In this case, our string has six characters, so the last index will be 5. Let's try it out.

```
print(name[5])
```

```
n
```

We see that the character in position five is the last character of the string. If we try to access index six, we get an index error telling us that it's out of range. So, we can only go up to length minus 1.

What if you want to print the last character of a string but you don't know how long it is? You can do that using **negative**. Let's see that in a different example:

```
text = "Random string with a lot of characters"  
print(text[-1])
```

```
s
```

```
print(text[-2])
```

r

In the above example, we don't know the length of the string, but it doesn't matter. Using negative indexes lets us access the positions in the string **starting from the** Nice, right?

On top of accessing individual characters, we can also access a **slice of a** A slice is the portion of a string that can **contain more than one** also sometimes called a We do that by creating a range using a colon as a separator. Let's see an example of this:

```
color = "Orange"  
color[1:4]
```

ran

The range we use when accessing a slice of a string works just like the one created by the range function. It **includes the first** but goes up to **one less than the last** In this case, we start with indexed one, the second letter of the string, and go up to index three, the fourth letter of the string.

Another option for the range is to include **only one of the two** In that case, it's assumed that the other index is either 0 for the first value or the length of the string for the second value. Check this out:

```
fruit = "Pineapple"  
print(fruit[:4])
```

Pine

Accessing the slice from nothing to 4 takes the first four characters of the string, indexes 0 to 3. Accessing the slice from 4 to nothing takes everything from index 4 onward.

```
print(fruit[4:])
```

apple

All of this indexing might seem confusing at first. Don't worry, we all took time to wrap our heads around it. Just like all the challenges we've come across so far, the key is to **keep practicing until you master** There are a bunch of exercises ahead to help you with that.

Now that we know how to select, slice, and access the parts of the string we want, we're going to learn how to modify them. That's coming up next.

14.4. String Indexing and Slicing Recap

String indexing allows you to access individual characters in a string. You can do this by using square brackets and the location, or index, of the character you want to access. It's important to remember that Python starts indexes at 0.

So to access the first character in a string, you would use the index [0]. If you try to access an index that's larger than the length of your string, you'll get an error. This is because you're trying to access something that doesn't exist! You can also access indexes from the end of the string going towards the start of the string by using negative values. The index [-1] would access the last character of the string, and the index [-2] would access the second-to-last character.

You can also access a portion of a string, called a **slice** or a **substring**. This allows you to access multiple characters of a string. You can do this by creating a range, using a colon as a separator between the start and end of the range, like [2:5].

This range is similar to the **range()** function we saw previously. It includes the first number, but goes to one less than the last number. For example:

```
>>> fruit = "Mangosteen" >>> fruit[1:4] 'ang'
```

The slice *includes* the character at index 1, and *excludes* the character at index 4. You can also easily reference a substring at the start or end of the string by only specifying one end of the range. For example, only giving the end of the range:

```
>>> fruit[:5] 'Mango'
```

This gave us the characters from the start of the string through index 4, *excluding* index 5. On the other hand this example gives is the characters *including* index 5, through the end of the string:

```
>>> fruit[5:] 'steen'
```

You might have noticed that if you put both of those results together, you get the original string back!

```
>>> fruit[:5] + fruit[5:] 'Mangosteen'
```

Cool!

14.5. Creating New Strings

In the last section, we saw how to access certain characters inside a string. Now, what if we wanted to change them? Imagine you have a string with a character that's wrong and you want to fix it, like the one shown in this one:

Video 28 (4:44 *Creating New Strings*)

```
message = "A kong string with a silly typo"
```

Taking into account what you learned about string indexing, you might be tempted to fix it by accessing the corresponding index and changing the character. Let's see what happens if we try that:

```
message[2] = "l"
```

```
TypeError: 'str' object does not support item assignment
```

We get a type error, right? In this case, we're told that strings don't support item assignment. This means that we can't change individual characters because **strings in Python are** which is just a fancy word, **meaning they can't be** What we can do is create a new string based on the old one, like this:

```
new_message = message[0:2] + "l" + message[3:]  
print(new_message)
```

“A long string with a silly typo”

Nice, we fixed the typo. But does this mean the message variable can never change? Not really. We can assign a new value to the same variable. Let's do that a couple of times to see how it works.

```
message = “This is a new message”  
print(message)  
This is a new message
```

```
message = “And another one”  
print(message)  
And another one
```

What we're doing here, is giving the message variable a whole new value. We're not changing the underlying string that was assigned to it before. We're assigning a whole new string with different content. If this seems a bit complex, that's okay. You don't need to worry about this right now.

We'll call this out whenever it's relevant for the programmer writing. So, we figured out how to create a new message from the old one. But how are we supposed to know which character to change? Let's try something different:

```
pets = “Cats & Dogs”  
pets.index(“&”)
```

In this case, we're using a **method** to get the index of a certain character. **A method is a function associated with a specific** We'll talk a lot more about classes and methods later. For now, what you need to know is that this is a function that applies to a variable, and we can call it by following the variable with a dot. Let's try this a few more times:

```
pets.index("C")
```

0

```
pets.index("Dog")
```

7

So the index method returns the index of the given inside the string. The substring that we pass, can be as long or as short as we want. What if there's more than one of the substring?

```
pets.index("S")
```

3

In this example, we know there are two **s** characters, but we only get one value. That's because the index method returns just the **first position that** What happens if the string doesn't have the substring we're looking for?

```
pets.index("x")
```

ValueError: Substring not found

The index method can't return a number because the substring is not there, so we get a *ValueError* instead. We said, that if the substring isn't there, we would get an error. So how can we know if a substring is contained in a string to avoid the error? Let's check this out:

“Dragons” in pets

False

“Cats” in pets

True

We can use the keyword *in* to check if a substring is contained in a string. We came across the keyword when using four loops. In that case, it was used for iteration. Here, it's a **conditional that can be either true or false**. It will be true if the substring is part of the string, and false if it is not. So here, the Dragons substring is not part of the string, and sadly, we can't have a Dragon as a pet!

All right, we just covered a bunch of new things and you're doing awesome. Let's put all the stuff together to solve a real-world problem.

Imagine that your company has recently moved to using a new domain, but a lot of the company email addresses are still using the old one. You want to write a program that replaces this old domain with the new one in any outdated email addresses. The function to replace the domain would look like this:

```
1 def replace_domain(email, old_domain, new_domain):
2     if "@" + old_domain in email:
3         index = email.index("@")
4         new_email = email[:index] + "@" + new_domain
5         return new_email
6     return email
```

This function is a bit more complex than others, so let's go through it line by line. First, we define the `replace_domain` function (line 1) which accepts three parameters: the email address to be checked, the old domain, and the new domain.

Having all these values as parameters instead of directly in the code, makes our function We are not just changing one domain to the other. We have a function that will work with all domains. Pretty sweet.

In the first line of the body of the function (line 2), we check if the concatenation of the `@` sign and the old domain are contained in the email address, using the keyword `in`. We check this to make sure the email has old domain on the portion that comes after the `@` sign.

If the condition is true, the email address needs to be updated. To do that, we first find out in line 3 the index where the old domain, including the `@` sign, starts. We know that this index will be a valid number because we've already checked that the substring was present.

So, using this index, we create the new email (line 4). This is a string that contains the first portion of the old email, up until the index we had calculated, followed by the @ sign and the new domain.

Finally, in line 5, we return this new email. If the email didn't contain the new domain, then we can just return it, which is what we do in the last line 6.

Wow, that was a really complex function with a lot of new things in it. So don't worry if you're finding it a bit tricky. Re-watch the video and take your time. If there's a specific part that's tripping you up, remember, you can always ask for help in the discussion forum. You may even find that someone has asked and got the answer to the same question already.

When you feel ready to move on, meet me over in the next section, where we're going to learn a lot more handy string methods.

14.6. Basic String Methods

In Python, strings are immutable. This means that they can't be modified. So if we wanted to fix a typo in a string, we can't simply modify the wrong character. We would have to create a new string with the typo corrected. We can also assign a new value to the variable holding our string.

If we aren't sure what the index of our typo is, we can use the string method *index* to locate it and return the index. Let's imagine we have the string "**lions tigers and bears**" in the variable `animals`. We can locate the index that contains the letter **g** using `animals.index("g")` which will return the index; in this case 8.

We can also use substrings to locate the index where the substring begins. `animals.index("bears")` would return 17, since that's the start of the substring. If there's more than one match for a substring, the index method will return the first match. If we try to locate a substring that doesn't exist in the string, we'll receive a **ValueError** explaining that the substring was not found.

We can avoid a `ValueError` by first checking if the substring exists in the string. This can be done using the *in* keyword. We saw this keyword earlier when we covered *for* loops. In this case, it's a conditional that will be either `True` or `False`. If the substring is found in the string, it will be `True`. If the substring is not found in the string, it will be `False`.

Using our previous variable we can do **"horses" in animals** to check if the substring "horses" is found in our variable. In this case, it would evaluate to False, since horses aren't included in our example string. If we did **"tigers" in** we'd get True, since this substring is contained in our string.

14.7. More String Methods

We said earlier that we had a lot of new exciting concepts coming up. Well, I'm not going to “string” you along anymore. We're going to tie up our lessons on strings with a bunch of fun methods for transforming our string text.

So far, we've seen ways you can access portions of strings using the indexing technique, create new strings by slicing and concatenating, find characters and strings using the index method, and even test if one string contains another. On top of all this string processing power, the string class provides a bunch of other methods for working with text.

Now, we'll show you how to use some of these methods. Remember, the goal is not for you to memorize all of this. Instead, we want to give you an idea of what you can do with strings in Python.

Video 29 (2:38 *More String Methods*)

Some string methods let you perform transformations or formatting on the string text, like and its opposite,

```
“Mountains”.upper()  
‘MOUNTAINS’  
“Mountains”.lower()  
‘mountains’
```

These methods are really useful when you're handling user input. Let's say you wanted to check if the user answered yes to a question. How would you know if the user typed it using upper or lower case? You don't need to, you just transform the answer to the case you want. Like this example:

```
answer = 'YES'  
if answer.lower() == "yes":  
  
    print("User said yes")
```

User said yes

Another useful method when dealing with user input is the **strip** method. This method will get rid of surrounding spaces in the string:

```
"yes ".strip()  
"yes"
```

If we ask the user for an answer, we usually don't care about any surrounding spaces. So it's a good idea to use the strip method to get rid of any white space. This means that **strip** doesn't just remove spaces, it **also removes tabs and new line** which are all characters we don't usually want in user-provided strings.

There are two more versions of this method, **rstrip** and to get rid of the whitespace characters just to the **left** or to the **right** of the string instead of both sides.

```
“ yes ”.lstrip()  
‘yes ‘
```

```
“ yes ”.rstrip()  
‘ yes’
```

Other methods give you information about the string itself. The method **count()** returns how many times a given substring appears within a string.

```
“The number of times e occurs in this string is 4”.count(“e”)  
4
```

The method **endswith()** returns whether the string ends with a certain substring.

```
“Forest”.endswith(“rest”)  
True
```

The method **isnumeric()** returns whether the string's made up of just numbers.

```
“Forest”.isnumeric()  
False
```

```
“12345”.isnumeric()  
True
```

Adding to that, if we have a string that is numeric, we can use the **int** function to convert it to an actual number.

```
int("12345") + int("54321")  
66666
```

In earlier videos, we showed that we can concatenate strings using the plus sign.

The **join** method can also be used for concatenating. To use the join method, we have to call it on the string that'll be used for joining.

```
" ".join(["This", "is", "a", "phrase", "joined", "by", "spaces"])  
'This is phrase joined by spaces'
```

In this case, we're using a string with a space in it. The method receives a list of strings and returns one string with each of the strings joined by the initial string. Let's check out another example:

```
"...".join(["This", "is", "a", "phrase", "joined", "by", "tripple", "dots"])  
'This...is...phrase...joined...by...tripple...dots'
```

Finally, we can also split a string into a list of strings. The **split** method returns a list of all the words in the initial string and it automatically splits by any whitespace.

```
“This is another example”.split()  
['This', 'is', 'another', 'example']
```

Are you starting to see how these string methods could be useful in your IT job? Okay, so we've just learned a bunch of new methods. But there are tons more that you can use on strings. We've included a list with the ones we talked about, and some new ones in the next cheat sheet. You'll also find a link to the full Python documentation there, which gives you all the info on each available method.

As we've said before, don't worry about trying to memorize everything. You'll pick these concepts up **with** and the documentation is always there if you need it. All right, last up in our string of string tutorials, we're going to check out how to format strings in section 14.9.

14.8. Advanced String Methods

We've covered a bunch of String class methods already, so let's keep building on those and run down some more advanced methods.

The string method **lower** will return the string with all characters changed to lowercase. The inverse of this is the **upper** method, which will return the string all in uppercase. Just like with previous methods, we call these on a string using dot notation, like `"this is a".lower()` This would return the string `"THIS IS A"` This can be super handy when checking user input, since someone might type in all lowercase, all uppercase, or even a mixture of cases.

You can use the **strip** method to remove surrounding whitespace from a string. Whitespace includes spaces, and **newline** You can also use the methods **lstrip** and **rstrip** to remove whitespace only from the left or the right side of the string, respectively.

The method **count** can be used to return the number of times a substring appears in a string. This can be handy for finding out how many characters appear in a string, or counting the number of times a certain word appears in a sentence or paragraph.

If you wanted to check if a string ends with a given substring, you can use the method `endswith()` This will return `True` if the substring is found at the end of the string, and `False` if not.

The **isnumeric** method can check if a string is composed of only numbers. If the string contains only numbers, this method will return True. We can use this to check if a string contains numbers before passing the string to the **int()** function to convert it to an integer, avoiding an error. Useful!

We took a look at string concatenation using the plus sign, earlier. We can also use the **join** method to concatenate strings. This method is called on a string that will be used to join a list of strings. The method takes a list of strings to be joined as a parameter, and returns a new string composed of each of the strings from our list joined using the initial string. For example, "**.join(["This","is","a","sentence"])**" would return the string "**This is a**

The inverse of the join method is the **split** method. This allows us to split a string into a list of strings. By default, it splits by any whitespace characters. You can also split by any other characters by passing a parameter.

14.9. Formatting Strings

Up to now we've been making strings using the plus sign (+) to just concatenate the parts of the string we wanted to create. We've also used the `str` function to convert numbers into strings so that we can concatenate them, too. This works, but it's not ideal, especially when the operations you want to do with the string are on the tricky side. There's a better way to do this using the **format** method. Let's see a couple of examples.

Video 30 (3:23 *Formatting Strings*)

```
name = "Manny"
number = len(name) * 3
print("Hello {}, your lucky number is {}".format(name, number))
Hello Manny, your lucky number is 15
```

In this example, we have two variables, **name** and **number**. We generate a string that has those variables in it by using the curly brackets `{}` placeholder to show where the variables should be written. We then pass the variables as a parameter to the `format` method.

See how it doesn't matter that `name` is a string and `number` is an integer? The `format` method deals with that, so we don't have to. Pretty neat, right?

The curly brackets aren't always empty. By using certain expressions inside those brackets, we can take advantage of the

full power of the format expression. Heads up, this can get complex fast. If at any point, you get confused, don't panic, you only really need to understand the basic usage of the format method we just saw.

One of the things we can put inside the curly brackets is the name of the variable we want in that position to make the whole string more readable. This is particularly relevant when the text can get rewritten or translated and the variables might switch places.

In this example, we could rewrite the message to make the variables appear in a different order. In that case, we'd need to pass the parameters to *format* in a slightly different way:

```
print("Your lucky number is {number},  
{name}.".format(name=name, number=len(name)*3))  
Your lucky number is 15, Manny.
```

Because we're using **placeholders with variable** the order in which the variables are passed to the format function doesn't matter. But for this to work, we need to set the names we're going to use and assign a value to them inside the parameters to

That's just the tip of the iceberg of what we can do with the format method. Want to dive a little deeper? Great, let's keep on going. We're going to check out a different example. Let's say you want to output the price of an item with and without tax.

Depending on what the tax rate is, the number might be a long number with a bunch of decimals.

```
price = 7.5
with_tax = price * 1.09
print(price, with_tax)
7.5, 8.175
```

So if something costs \$7.5 without tax and the tax rate is 9%, the price with tax would be \$8.175. Since there's no such thing as half a penny anymore, that number doesn't make sense. So to fix this we can make the format function print only two decimals, like this:

```
print("Base price: ${:.2f}. With Tax: ${:.2f}".format(price, with_tax))
Base price: $7.50. With Tax: $8.18
```

In this case between the curly brackets we're writing a **formatting** expression. There are a bunch of different expressions we can write. These expressions are needed when we want to tell Python to format our values in a way that's different from the default.

The expression starts with a **colon to separate it from the field name** that we saw before. After the colon, we write `.2f`. This means we're going to format a **float** number and that there should be **two digits** after the decimal dot. So no matter what the price is, our function always prints two decimals.

Remember when we did the table to convert from Fahrenheit to Celsius temperatures? Our table looked kind of ugly because it was full of float numbers that had way too many decimal digits. Using the format function, we can make it look a lot nicer:

```
def to_celcius(x):
    return (x - 32) * 5 / 9
for x in range(0, 101, 10):
    print("{:>3} F | {:>6.2f} C".format(x, to_celcius(x)))
# Align text to the right with a total of 3 spaces for x
# and 6 spaces for to_celcius
0 F | -17.78 C
10 F | -12.22 C
20 F | -6.67 C
30 F | -1.11 C
40 F | 4.44 C
50 F | 10.00 C

60 F | 15.56 C
70 F | 21.11 C
80 F | 26.67 C
90 F | 32.22 C
100 F | 37.78 C
```

In these two expressions we're using the greater than operator to align text to the right so that the output is neatly aligned. In the first expression we're saying we want the numbers to be aligned to the right for a total of three spaces. In the second expression we're saying we want the number to always have exactly two decimal places and we want to align it to the right at six spaces.

We can use string formatting like this to make the output of our program look nice and also to generate useful logging and debugging messages. Over the course of my sysadmin career, I've grown used to formatting strings to create more informative error messages. They help me understand what's going on with a script that's failing.

There's a ton more formatting options you can use when you need them. But don't worry about learning them all at once, we'll explain any others as they come along and we'll put everything in a cheat sheet that you can refer to whenever you need a formatting expression.

We will take a look at that in section 14.12 and then we'll have a quiz in section 14.13 to help you get more familiar with all this new knowledge.

14.10. String Formatting Recap

You can use the **format** method on strings to concatenate and format strings in all kinds of powerful ways. To do this, create a string containing curly brackets, as a placeholder, to be replaced. Then call the format method on the string using *.format()* and pass variables as parameters. The variables passed to the method will then be used to replace the curly bracket placeholders. This method automatically handles any conversion between data types for us.

If the curly brackets are empty, they'll be populated with the variables passed in the order in which they're passed. However, you can put certain expressions inside the curly brackets to do even more powerful string formatting operations. You can put the name of a variable into the curly brackets, then use the names in the parameters. This allows for more easily readable code, and for more flexibility with the order of variables.

You can also put a formatting expression inside the curly brackets, which lets you alter the way the string is formatted. For example, the formatting expression **{:.2f}** means that you'd format this as a float number, with two digits after the decimal dot. The colon acts as a separator from the field name, if you had specified one. You can also specify text alignment using the greater than operator: For example, the expression **{:>3.2f}** would align the text three spaces to the right, as well as specify a float number with two decimal places. String formatting can be very handy for outputting easy-to-read textual output.

14.11. Cheat Sheet 4: String Reference

To locate the *String Reference Cheat Sheet* pdf file, open the “Cheat Sheets” folder. It’s one of the folders inside the course resources folder you downloaded earlier.

14.12. Cheat Sheet 5: Formatting Strings

To locate the *Formatting Strings Cheat Sheet* pdf file, open the “Cheat Sheets” folder. It’s one of the folders inside the course resources folder you downloaded earlier.

14.13. Practice Quiz 10: 5 Questions

1. The `is_palindrome` function checks if a string is a palindrome. A palindrome is a string that can be equally read from left to right or right to left, omitting blank spaces, and ignoring capitalization. Examples of palindromes are words like kayak and radar, and phrases like "Never Odd or Even". Fill in the blanks in this function to return `True` if the passed string is a palindrome, `False` if not. – 1 point

```
# We'll create two strings, to compare them
```

```
# Traverse through each letter of the input string
```

```
# Add any non-blank letters to the
```

```
# end of one string, and to the front
```

```
# of the other string.
```

```
# Compare the strings
```


2. Using the format method, fill in the gaps in the `convert_distance` function so that it returns the phrase "X miles equals Y km", with Y having only 1 decimal place. For example, `convert_distance(12)` should return "12 miles equals 19.2 km".

3. If we have a string variable named `Weather = "Rainfall"`, which of the following will print the substring or all characters before the "f"?

```
print(Weather[:4])  
print(Weather[4:])  
print(Weather[1:4])  
print(Weather[:"f"])
```

4. Fill in the gaps in the `nametag` function so that it uses the format method to return `first_name` and the first initial of `last_name` followed by a period. For example, `nametag("Jane", "Smith")` should return "Jane S."

```
# Should display "Jane S."  
# Should display "Francesco R."
```

```
# Should display "Jean-Luc G."
```

5. The `replace_ending` function replaces the old string in a sentence with the new string, but only if the sentence ends with the old string. If there is more than one occurrence of the old string in the sentence, only the one at the end is replaced, not all of them. For example, `replace_ending("abcabc", "abc", "xyz")` should return `abcxyz`, not `xyzxyz` or `xyzabc`. The string comparison is case-sensitive, so `replace_ending("abcabc", "ABC", "xyz")` should return `abcabc` (no changes made).

```
# Check if the old string is at the end of the sentence
```

```
# Using i as the slicing index, combine the part
```

```
# of the sentence up to the matched string at the
```

```
# end with the new string
```

```
# Return the original sentence if there is no match
```

```
# Should display "It's raining cats and dogs"  
# Should display "She sells seashells by the seashore"  
# Should display "The weather is nice in May"  
  
# Should display "The weather is nice in April"
```

14.13.1. Answers to Practice Quiz 10

1.

```
# We'll create two strings, to compare them
```

```
# Traverse through each letter of the input string
```

```
# Add any non-blank letters to the
```

```
# end of one string, and to the front
```

```
# of the other string.
```

```
# Compare the strings
```

2.

3. A. Formatted this way, the substring preceding the character "f", which is indexed by 4, will be printed.

4.

```
# Should display "Jane S."  
# Should display "Francesco R."  
# Should display "Jean-Luc G."
```

5.

```
# Check if the old string is at the end of the sentence
```

```
# Using i as the slicing index, combine the part
```

```
# of the sentence up to the matched string at the
```

```
# end with the new string
```

```
i = sentence.rindex(old)
```

```
# Return the original sentence if there is no match
```

```
# Should display "It's raining cats and dogs"
```

```
# Should display "She sells seashells by the seashore"
```

```
# Should display "The weather is nice in May"
```

```
# Should display "The weather is nice in April"
```

15. Lists

15.1. What is a List?

As you know by now, Python comes with a lot of ready-to-use data types. We've seen integers, floats, Booleans, and strings in detail. But those data types can only take you so far. Eventually in your scripts, you want to develop code that manipulates collections of items like a list of strings representing all the file names in a directory or a list of integers representing the size of network packets.

This is where the list data type comes in handy. You can think of lists as long boxes with the space inside the box divided up into different slots.

Video 31 (3:06 *What is a List?*)

Each slot can contain a different value. Like we mentioned earlier when we first came across the list, in Python, we use square brackets [] to indicate where the list starts and ends. Let's check out an example:

```
x = ["Now", "we", "are", "cooking!"]
```

Here, we've created a new variable called `x` and set its contents to be a list of strings. We can check the type of `x` using the `type` function we saw a little while ago:

```
type(x)
```



```
'list'>
```

Nice. Python tells us this is a list. In the same way, we've done with other variables, we can show the contents of the whole list using the *print* function:

```
print(x)
['Now', 'we', 'are', 'cooking!']
```

The **length** of the list is how many elements it has. To get that value, we'll use the same *len* function we used for strings:

```
len(x)
4
```

That's right. Our list has four elements. When calling *len* for the list, it doesn't matter how long each string is on its own. What matters is how many elements the list has. To check if a list contains a certain element, you can use the keyword *in* like in these examples:

```
"are" in x
```

```
True
```

```
"Today" in x
```

```
False
```

Again, like when we use this with strings, the result of this check is a Boolean, which we can use as a condition for branching or looping. We can also use indexing to access individual elements depending on their position in the list. To do that, we use the

square brackets and the index we want to access, exactly like we did with strings:

```
print(x[0])
```

Now

```
print(x[3])
```

cooking!

Remember that the first element is given the index zero. This means the last index of the list will be the length of the list minus one. What happens if we try to access an element after the end of the list?

```
print(x[4])
```

IndexError: List index out of range

You might have seen this coming. We get an index error. We can't go over the end of the list. Remember that because list indexes start at zero, accessing the item at index 4 means we're trying to access the 5th element in the list. There are only 4 elements. So we're out of range if we try to access the index number 4.

Does this seem a bit confusing? If it does, the visualization in the video might help you out. As you can see in the video, index 4 doesn't point at anything since there's no slot 4 in our list.

As with strings, we can also use indexes to create a **slice** of the list. For this, we use ranges of two numbers separated by a

colon:

```
x[1:3]  
['we', 'are']
```

Again, the second element is not included in the slice. So the range goes to the second index minus one. Here's another example:

```
x[:2]  
['Now', 'we']
```

Here, we start at index one and go up to 1 less than 3, which is 2. We can also leave out one of the range indexes empty:

```
x[2:]  
['are', 'cooking!']
```

The first value defaults to 0 and the second value to the length of the list. Makes sense? If all this sounds really familiar to what we said about strings, then this course is working as intended. That's because strings and lists are very similar data types.

In Python, **strings** and **lists** are both **examples of** There are other sequences too, and they all share a bunch of operations like iterating over them using *for* loops, indexing using the *len* function to know the length of the sequence, using *+* to concatenate two sequences and using *in* to verify if the sequence contains an element. So this is great news.

While understanding indexing is hard, once you know it for one data type, you've pretty much mastered it for every data type. So you actually know way more than you thought.

Wow! Now, we're really cooking! Next up, we're going to look at some more list operations. This time, actually specific to lists.

15.2. Lists Defined

Lists in Python are defined using square brackets, with the elements stored in the list separated by commas: `list = ["This", "is", "a"]`. You can use the `len()` function to return the number of elements in a list: `len(list)` would return

You can also use the `in` keyword to check if a list contains a certain element. If the element is present, it will return a `True` boolean. If the element is not found in the list, it will return `False`. For example, `"This" in list` would return `True` in our example.

Similar to strings, lists can also use indexing to access specific elements in a list based on their position. You can access the first element in a list by doing `list[0]` which would allow you to access the string

In Python, lists and strings are quite similar. They're both examples of sequences of data. Sequences have similar properties, like (1) being able to iterate over them using `for` (2) support indexing; (3) using the `len` function to find the length of the sequence; (4) using the plus operator `+` in order to concatenate; and (5) using the `in` keyword to check if the sequence contains a value. Understanding these concepts allows you to apply them to other sequence types as well.

15.3. Modifying the Contents of a List

One of the ways that lists and strings are different is that **lists are** which is another fancy word to say that they can change. This means we can add, remove, or modify elements in a list. Let's go back to our example of thinking of a list as a long box.

Video 32 (3:10 *Modifying the Contents of a List?*)

Changing the list means we keep the same box and we add, remove, or change the elements inside that box. We'll now go through the methods that let us modify the list one by one. If all these details seem a little overwhelming, that's okay. As usual, there will be a cheat sheet at the end and you'll have lots of chances to practice each of these methods as we go along. You don't need to learn all those by heart, and of course you can always review anything that isn't clear. So don't worry, I got your back.

We'll start with the simplest change; adding an element to a list using the **append** method. Let's check this out in the tastiest example yet:

```
fruits = ["Pineapple", "Banana", "Apple", "Melon"]
fruits.append("Kiwi")
print(fruits)
["Pineapple", "Banana", "Apple", "Melon", "Kiwi"]
```

The `append` method adds a new element at the end of the list. No matter how long the list is, the element always gets added to the end. You could start with an empty list and add all of its items using `append`.

If you want to insert an element in a different position, instead of at the end, you can use the `insert` method:

```
fruits.insert(0, "Orange")
print(fruits)
["Orange", "Pineapple", "Banana", "Apple", "Melon", "Kiwi"]
```

The `insert` method takes an index as the first parameter and an element as the second parameter. It adds the element at that index in the list. To add it as the first element, we use index zero. We can use any other number. What happens if we use a number larger than the length of the list?

```
fruits.insert(25, "Peach")
print(fruits)
["Orange", "Pineapple", "Banana", "Apple", "Melon", "Kiwi",
"Peach"]
```

No errors. You can say that it even worked just peachy! If we use an index higher than the current length, the element just gets added to the end. You can pass any number to `insert` but usually, you either add at the beginning using `insert` at the zero index or at the end using

We can also remove elements from the list. We can do it using the value of the element we want to remove. Can you guess what method we would use? You got it! Use the *remove* method.

```
fruits.remove("Melon")
print(fruits)
["Orange", "Pineapple", "Banana", "Apple", "Kiwi", "Peach"]
```

The *remove* method removes from the list the first occurrence of the element we pass to it. What happens if the element is not in the list?

```
fruits.remove("Pear")
ValueError: list.remove(x): x not in the list
```

We got a value error, telling us the element isn't in the list. Another way we can remove elements is by using the **pop** method, which receives an index.

```
fruits.pop(3)
'Apple'
print(fruits)
["Orange", "Pineapple", "Banana", "Kiwi", "Peach"]
```

The *pop* method returns the element that was removed at the index that was passed.

The last way to modify the contents of a list is to change an item by **assigning something else** to that position, like this:


```
fruits[2] = "Strawberry"  
print(fruits)  
["Orange", "Pineapple", "Strawberry", "Kiwi", "Peach"]
```

Wow, the contents of our fruits variable have changed a lot since we started this section. But it's always the same variable, the same box. We've just modified what's inside. Modifying the contents of lists will come up in tons of scripts as we operate with them.

If the list contains hosts on a network, you could add or remove hosts as they come online or offline. If the list contains users authorized to run a certain process, you could add or remove users when permissions are granted or removed and so on.

You've now seen a number of methods that let us modify the contents of a list, adding, removing, and changing the elements that are stored inside the list. Whenever you need to write a program that will handle a variable amount of elements, you'll use a list.

What if you need a sequence of a fixed amount of elements?
That's coming up in the next section.

15.4. Modifying Lists

While lists and strings are both sequences, a big difference between them is that **lists** are mutable. This means that the contents of the list can be changed, unlike strings which are immutable. You can add, remove, or modify elements in a list.

You can add elements to the end of a list using the **append** method. You call this method on a list using dot notation, and pass in the element to be added as a parameter. For example, **list.append("New data")** would add the string "New data" to the end of the list called list.

If you want to add an element to a list in a specific position, you can use the **insert** method. The method takes two parameters: the first specifies the index in the list, and the second is the element to be added to the list. So **list.insert(0, "New data")** would add the string "New data" to the front of the list. This wouldn't overwrite the existing element at the start of the list. It would just shift all the other elements by one. If you specify an index that's larger than the length of the list, the element will simply be added to the end of the list.

You can remove elements from the list using the **remove** method. This method takes an element as a parameter, and removes the first occurrence of the element. If the element isn't found in the list, you'll get a **ValueError** error explaining that the element was not found in the list.

You can also remove elements from a list using the **pop** method. This method differs from the remove method in that it takes an index as a parameter, and returns the element that was removed. This can be useful if you don't know what the value is, but you know where it's located. This can also be useful when you need to access the data and also want to remove it from the list.

Finally, you can change an element in a list by using indexing to overwrite the value stored at the specified index. For example, you can enter **list[0] = "Old data"** to overwrite the first element in a list with the new string "Old data".

15.5. Lists and Tuples

As we called out before, there are a number of data types in Python that are all sequences.

Video 33 (2:25 *Lists and Tuples*)

Strings are sequences of characters and are immutable. Lists are sequences of elements of any type and are mutable. A third data type that's a sequence and also closely related to lists is the tuple. **Tuples** are sequences of elements of any type that are We write tuples in parentheses instead of square brackets. For example,

```
fullname = ('Garce', 'M', 'Hopper')
```

You might be wondering, why do we even need another sequence type? Weren't lists great? Yes, lists are great. They can hold any number of elements and we can add, remove and modify their contents as much as we want, but there are cases when we want to make sure an element in a certain position or index refers to one specific thing and won't change. In these situations, lists won't help us.

In our example, we have a tuple that represents someone's full name. The first element of the tuple is the first name (Grace). The second element is the middle initial, and the third element is

the last name. If we add another element somewhere in there, what would that element represent?

It would just be confusing and our code wouldn't know what to do with it, and that's why modifying isn't allowed. In other words, when using tuples the **position of the elements inside the tuple have**

Tuples are used for lots of different things in Python. One common example is the return value of functions. When a function returns more than one value, it's actually returning a tuple. Remember the function to convert seconds to hours, minutes, and seconds that we saw a while back?

```
def convert_seconds(seconds):
    hours = seconds//3600
    minutes = (seconds - hours*3600)//60
    remaining_seconds = seconds - hours * 3600 - minutes * 60
    return hours, minutes, remaining_seconds
```

Just to remind you, this function returns three values. In other words, it returns a tuple of three elements (hours, minutes and remaining_seconds). Let's give it a try:

```
result = convert_seconds(5000)
type(result)
'tuple'>
```

We see the result is a tuple. What if we print it?

```
print(result)
(1, 23, 20)
```

We see that it has the three elements we expect it to have. Remember, since this is a tuple, the order matters. The first element represents the hours, the second one represents the minutes, and the third represents the seconds. One interesting thing we can do with tuples is **unpack**. This means that we can **turn a tuple of three elements into three separate**

Because the order won't change, we know what those variables represent, like this.

```
hours, minutes, seconds = result
print(hours, minutes, seconds)
1, 23, 20
```

So now we've split the tuple into three separate values. We've seen before that we can also do this directly when calling the function without the intermediate result variable.

```
hours, minutes, seconds = convert_seconds(1000)
print(hours, minutes, seconds)
0, 16, 40
```

In Python, it's really common to **use tuples to represent data that has more than one value and that needs to be kept**. For example, you could use a tuple to have a filename and its size, or you

could store the name and email address of a person, or a date and time and the general health of the system at any point in time.

Can you see how these different data types could help you automate some of your IT work? Pretty cool, right? Knowing when to use tuples and when to use lists can seem a little fuzzy at first, but don't worry, it'll get clearer as we tackle more examples.

15.6. Tuples Recap

As we mentioned earlier, strings and lists are both examples of sequences. Strings are sequences of characters and are immutable. Lists are sequences of elements of any data type and are mutable. The third sequence type is the tuple. Tuples are like lists, since they can contain elements of any data type. But unlike lists, tuples are immutable. They're specified using parentheses instead of square brackets.

You might be wondering why tuples are a thing, given how similar they are to lists. Tuples can be useful when we need to ensure that an element is in a certain position and will not change. Since lists are mutable, the order of the elements can be changed on us. Since the order of the elements in a tuple can't be changed, the position of the element in a tuple can have meaning.

A good example of this is when a function returns multiple values. In this case, what gets returned is a tuple, with the return values as elements in the tuple. The order of the returned values is important, and a tuple ensures that the order isn't going to change. **Storing the elements of a tuple in separate variables is called** This allows you to take multiple returned values from a function and store each value in its own variable.

15.7. Iterating over Lists and Tuples

When we looked at *for* loops, we said they iterate over a sequence of elements. One of the examples we checked out was iterating over a list. Let's take a little trip to the zoo to see this in action:

Video 34 (5:25 *Iterating Over Lists and Tuples*)

```
animals = ["Lion", "Zebra", "Dolphin", "Monkey"]
chars = 0
for animal in animals:
    chars += len(animal)
print("Total characters: {}, Average length: {}".format(chars,
chars/len(animals)))
Total characters: 22, Average length: 5.5
```

In this code, we're iterating over a list of strings. For each of the strings, we get its length and add it to the total amount of characters. At the end we print the total and the average which we get by dividing the total by the length of the list.

You can see we're using the *len* function twice, once to get the length of the string and then again to get the amount of elements in the list. What if you wanted to know the index of an element while going through the list? You could use the *range* function and then use indexing to access the elements at the index that *range* returned. Or you could just use the **enumerate** function.

```
winner = ["Ashley", "Dylan", "Reese"]
for index, person in enumerate(winner):
    print("{} - {}".format(index + 1, person))
```

1 - Ashley

2 - Dylan

3 - Reese

The enumerate function returns a tuple for each element in the list. The first value in the tuple is the **index** of the element in the sequence. The second value in the tuple is the **element** in the sequence. You're the real winner with the enumerate function because it does all the work for you! Pretty useful, right?

Let's use all of this now to solve a slightly more interesting problem. Say you have a list of tuples containing two strings each. The first string is an **email address** and the second is the **full name** of the person with that email address.

You want to write a function that creates a new list containing one string per person including their name and the email address between angled brackets, the format usually used in emails, like this: [Terrance Ford](#)

So, what do we need to do? We'll start by defining a function that receives a list of people, that is takes the argument *people*. Remember, *people* is a list of tuples where the first element is the email address and the second one is the full name. So, in our

function, we first create the variable that we will use as a return value which will be a list and we'll call it

```
def full_emails(people):  
    result = []  
    for name, email in people:  
        result.append("{} <{}>".format(name, email))  
    return result
```

Will this work? Try it out yourself!

```
print(full_emails([("alex@example.com", "Alex Diego"),  
                  ("Shay@example.com", "Shay Brandt")]))
```

```
['Alex Diego ', 'Shay Brandt ']
```

Yes, this works as expected.

Before we move on, a quick word of caution about some common errors when dealing with lists in Python. Because we use the `range` function so much with *for* loops, you might be tempted to use it for iterating over indexes of a list and then to access the elements through indexing. You could be particularly inclined to do this if you're used to other programming languages before.

Because in some languages, the only way to access an element of a list is by using indexes. This works but looks ugly. It's more idiomatic in Python to iterate through the elements of the list directly or using *enumerate* when you need the indexes like we've done so far.

There are some specific cases that do require us to iterate over the indexes, for example, when we're trying to modify the elements of the list we're iterating. By the way, if you're iterating through a list and you want to modify it at the same time, you need to be very careful. If you remove elements from the list while iterating, you're likely to end up with an unexpected result. In this case, it might be better to use a copy of the list instead.

We've now seen a bunch of different things we can do with lists, and hopefully you're starting to see how they can be a very powerful tool in your IT specialist toolkit. Next up, we're going to learn a powerful technique for creating lists.

15.8. Iterating Over Lists Using Enumerate

When we covered *for* loops, we showed the example of iterating over a list. This lets you iterate over each element in the list, exposing the element to the *for* loop as a variable. But what if you want to access the elements in a list, along with the index of the element in question?

You can do this using the **enumerate()** function. The `enumerate()` function takes a list as a parameter and returns a tuple for each element in the list. The first value of the tuple is the index and the second value is the element itself.

15.9. List Comprehensions 1

We're almost done with our deep dive into Python list. But before we continue to our next data structure, let's talk about creating lists in a shorter way.

Video 35 (2:44 *List Comprehensions*)

Say we wanted to create a list with multiples of 7 from 7 to 70, we could do it like this.

```
multiples = []
for x in range(1,11):
    multiples.append(x*7)

print("Multiples of 7:", multiples)
Multiples of 7: [7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

This works fine and is a good way of solving it. But because creating lists based on sequences is such a common task, Python provides a technique called **list comprehension** that lets us do it in just one line. This is how we would do the same with list comprehension:

```
multiples = [x*7 for x in range(1,11)]

print("Multiples of 7:", multiples)
Multiples of 7: [7, 14, 21, 28, 35, 42, 49, 56, 63, 70]
```

The result is the same.

List comprehensions let us create new lists based on sequences or

So we can use this technique whenever we want to create a list based on a range like in this example, or based on the contents of a list, a tuple, a string or any other Python sequence. The syntax tries to copy how you would express these concepts with natural language, although it can still be confusing sometimes. Let's check out a different example.

Say we have a list of strings with the names of programming languages like this one,

```
languages = ["Python", "Perl", "Ruby", "Go", "Java", "C"]
```

and we want to generate a list of the length of the strings. We could iterate over the list and add them using *append* like we did before, or we could use a list comprehension like this:

```
lengths = [len(language) for language in languages]
print(lengths)
[6, 4, 4, 2, 4, 1]
```

List comprehensions also let us use a conditional clause. Say we wanted all the numbers that are divisible by 3 between 0 and a

100, we could create a list like this:

```
z = [x for x in range(0,101 ) if x % 3 == 0]
print(z)
[0, 3, 6, 9, 12, 15, 18, 21, 24, 27, 30, 33, 36, 39, 42, 45, 48, 51,
54, 57, 60, 63, 66, 69, 72, 75, 78, 81, 84, 87, 90, 93, 96, 99]
```

In this case we just want the element x to be a part of the list, but we only want the numbers where the remainder of the division by 3 is 0. So we add the conditional clause after the range.

Using list comprehensions when programming in Python is totally optional. Sometimes it can make the code look nicer and more readable, at other times it can have the opposite effect, especially if we try to pack too much information together. In general, it's a good idea to know that list comprehensions exist, especially when you're trying to understand someone else's code.

All right, we've now seen a bunch of different methods we can use to operate with lists and tuples, and Python provides even more of them that we didn't get to talk about.

In section 15.10, you'll find a list of the most common operations and links to the official documentation in case you want to learn more. After that you can practice your new skills in the next quiz.

15.10. List Comprehensions Recap

You can create lists from sequences using a for loop, but there's a more streamlined way to do this: **list** List comprehensions allow you to create a new list from a sequence or a range in a single line.

For example, `[x*2 for x in range(1,11)]` is a simple list comprehension. This would iterate over the range 1 to 10, and multiply each element in the range by 2. This would result in a list of the multiples of 2, from 2 to 20.

You can also use conditionals with list comprehensions to build even more complex and powerful statements. You can do this by appending an if statement to the end of the comprehension. For example, `[x for x in range(1,101) if x % 10 == 0]` would generate a list containing all the integers divisible by 10 from 1 to 100. The if statement we added here evaluates each value in the range from 1 to 100 to check if it's evenly divisible by 10. If it is, it gets added to the list.

List comprehensions can be really powerful, but they can also be super complex, resulting in code that's hard to read. Be careful when using them, since it might make it more difficult for someone else looking at your code to easily understand what the code is doing.

15.11. Cheat Sheet 6: Lists and Tuples Operations

To locate the *Lists and Tuples Operations Cheat Sheet* pdf file, open the “Cheat Sheets” folder. It’s one of the folders inside the course resources folder you downloaded earlier.

15.12. Practice Quiz 11: 6 Questions

1. Given a list of filenames, we want to rename all the files with extension `hpp` to the extension `h`. To do this, we would like to generate a new list called `newfilenames`, consisting of the new filenames. Fill in the blanks in the code using any of the methods you've learned thus far, like a *for* loop or a *list* – 1 point

```
# Generate newfilenames as a list containing the new filenames
# using as many lines of code as your chosen method requires.
newfilenames = []
```

```
# Should be ["program.c", "stdio.h", "sample.h", "a.out", "math.h", "
hpp.out"]
```

2. Fill in the blanks in the code to create a function that turns text into pig latin: a simple text transformation that modifies each word moving the first character to the end and appending "ay" to the end. For example, `python` ends up as `ythonpay`. - 1 point

```
# Separate the text into words
```

```
container = []
words = text.split()
```

```
# Create the pig latin word and add it to the list
```

```
# Turn the list back into a phrase
```

3. The permissions of a file in a Linux system are split into three sets of three permissions: read, write, and execute for the owner, group, and others. Each of the three values can be expressed as an octal number summing each permission, with 4 corresponding to read, 2 to write, and 1 to execute. Or it can be written with a string using the letters r, w, and x or - when the permission is not granted. For example: 640 is read/write for the owner, read for the group, and no permissions for the others; converted to a string, it would be: "rw-r-----" 755 is read/write/execute for the owner, and read/execute for group and others; converted to a string, it would be: "rwxr-xr-x" Fill in the blanks to make the code convert a permission in octal format into a string format. - 1 point

```
# Iterate over each of the digits in octal
```

```
# Check for each of the permissions values
```

```
x -= value
```

4. Tuples and lists are very similar types of sequences. What is the main thing that makes a tuple different from a list? - *1 point*

A tuple is mutable

A tuple contains only numeric characters

A tuple is immutable

A tuple can contain only one type of data at a time

5. The `group_list` function accepts a group name and a list of members, and returns a string with the format: `group_name: member1, member2, ...`. For example, `group_list("g", ["a","b","c"])` returns "g: a, b, c". Fill in the gaps in this function to do that. - *1 point*

```
members = []
```

6. The `guest_list` function reads in a list of tuples with the name, age, and profession of each party guest, and prints the sentence "Guest is X years old and works as ___." for each one. For example, `guest_list(('Ken', 30, "Chef"), ("Pat", 35, 'Lawyer'), ('Amanda', 25, "Engineer"))` should print out: Ken is 30 years old and works as Chef. Pat is 35 years old and works as Lawyer.

Amanda is 25 years old and works as Engineer. Fill in the gaps in this function to do that. - 1 point

#Click Run to submit code

"""

Output should match:

Ken is 30 years old and works as Chef

Pat is 35 years old and works as Lawyer

Amanda is 25 years old and works as Engineer

"""

15.12.1. Answers to Practice Quiz 11

1.

```
# Generate newfilenames as a list containing the new filenames  
# using as many lines of code as your chosen method requires.
```

```
newfilenames = []  
    eachname = filenames[counter]  
    newfilenames.append(tempnames)
```

```
# Should be ["program.c", "stdio.h", "sample.h", "a.out", "math.h", "  
hpp.out"]
```

2.

```
# Separate the text into words
```

```
container = []  
words = text.split()
```

```
# Create the pig latin word and add it to the list
```

```
# Turn the list back into a phrase
```

3.

```
# Iterate over each of the digits in octal
```

```
# Check for each of the permissions values
```

```
    result += letter  
    x -= value
```

4. C. Unlike lists, tuples are immutable, meaning they can't be changed.

5.

```
members = []
```


6.

#Click Run to submit code

"""

Output should match:

Ken is 30 years old and works as Chef

Pat is 35 years old and works as Lawyer

Amanda is 25 years old and works as Engineer

"""

16. Dictionaries

16.1. What is a Dictionary?

How are you feeling so far? Lists and strings are pretty cool, right? These tools let us do a ton of neat stuff in our code so they can be super fun to experiment with.

We're now going to learn about another data type, Like lists, dictionaries are used to organize elements into collections. Unlike lists, you don't access elements inside dictionaries using their position. Instead, **the data inside dictionaries take the form of pairs of keys and**

To get a dictionary value we use its corresponding key. Another way these two vary is while in a list the index must be a number, in a dictionary you can use a bunch of different data types as keys, like strings, integers, floats, tuples, and more.

The name dictionary comes from how they work in a similar way to human language dictionaries. In an English language dictionary, the word comes with a definition. In the language of a Python dictionary, the word would be the key and the definition would be the value. Make sense? Let's check out an example.

Video 36 (4:02 *What is a dictionary?*)

You can create an empty dictionary in a similar way to creating an empty list, except instead of square brackets dictionaries use **curly brackets** to define their content.

```
x = {}
```

Once again, we can use the `type` function to check that the variable we've just created is a dictionary:

```
type(x)  
'dict'>
```

Creating initialized dictionaries isn't too different from the syntax we used in earlier videos to create initialized lists or tuples. But instead of a series of slots with values in them, we have a series of keys that point at values. Okay, let's check out an example dictionary. We'll call it

```
file_counts = {"jpg": 10, "txt": 14, "csv": 2, "py": 23}  
print(file_counts)  
{'jpg': 10, 'txt': 14, 'csv': 2, 'py': 23}
```

In this *file_counts* dictionary, we've stored keys that are strings, like `jpg`, that point at integer values, like `10`. When creating the dictionary we use colons, and between the key and the value we separate each pair by commas. In a dictionary, it's perfectly fine to mix and match the data types of keys and values like this and can be very useful.

In this example, we're using a dictionary to store the number of files corresponding to each extension. It makes sense to encode

the file extension formatting in a string, while it's natural to represent a count as an integer number.

Let's say you want to find out how many text files there are in the dictionary. To do this, you would use the key `txt` to access its associated value.

```
file_counts["txt"]
```

```
14
```

The syntax to do this may look familiar, since we used something similar in our examples of indexing strings, lists, and tuples. You can also use the `in` keyword to check if a key is contained in a dictionary. Let's try a couple of keys.

```
"jpg" in file_counts
```

```
True
```

```
"html" in file_counts
```

```
False
```

Dictionaries are

You might remember what mutable means from an earlier section. That's right, it means we can add, remove and replace entries. To add an entry in a dictionary, just use the square brackets to create the key and assign a new value to it. Let's add a file count of eight for a new CFG file extension and dictionary.

```
file_counts ["cfg"] = 8
print(file_counts)
{'jpg': 10, 'txt': 14, 'csv': 2, 'py': 23, 'cfg': 8}
```

This brings up an interesting point about dictionaries. What do you think will happen if we try to add a key that already exists in the dictionary?

```
file_counts["csv"] = 17
print(file_counts)
{'jpg': 10, 'txt': 14, 'csv': 17, 'py': 23, 'cfg': 8}
```

When you use a key that already exists to set a value, the value that was already paired with that key is replaced. As you can see in this example, the value associated with the `csv` key used to be 2, but it's now 17. The keys inside of a dictionary are unique. If we try to store two different values for the same key, we'll just replace one with the other.

Last off, we can delete elements from a dictionary with the `del` keyword by passing the dictionary and the key to the element as if we were trying to access it.

```
del file_counts["cfg"]
print(file_counts)
{'jpg': 10, 'txt': 14, 'csv': 17, 'py': 23}
```

What do you think? Dictionaries seem pretty useful, right? We've now seen how to create a dictionary and how to add, modify, and delete elements stored in the dictionary. Up next, we'll discover some interesting things we can do with them.

16.2. Dictionaries Defined

Dictionaries are another data structure in Python. They're similar to a list in that they can be used to organize data into collections. However, data in a dictionary isn't accessed based on its position. Data in a dictionary is organized into pairs of keys and values. You use the key to access the corresponding value. Where a list index is always a number, a dictionary key can be a different data type, like a string, integer, float, or even tuples.

When creating a dictionary, you use curly brackets: When storing values in a dictionary, the key is specified first, followed by the corresponding value, separated by a colon. For example, **animals = { "bears":10, "lions":1, "tigers":2 }** creates a dictionary with three key value pairs, stored in the variable **animals**.

The key "bears" points to the integer value 10, while the key "lions" points to the integer value 1, and "tigers" points to the integer 2. You can access the values by referencing the key, like this: This would return the integer 10, since that's the corresponding value for this key.

You can also check if a key is contained in a dictionary using the **in** keyword. Just like other uses of this keyword, it will return True if the key is found in the dictionary; otherwise it will return False.

Dictionaries are mutable, meaning they can be modified by adding, removing, and replacing elements in a dictionary, similar to lists. You can add a new key value pair to a dictionary by

assigning a value to the key, like this: **animals["zebras"] = 2**. This creates the new key in the animal dictionary called zebras, and stores the value 2.

You can modify the value of an existing key by doing the same thing. So **animals["bears"] = 11** would change the value stored in the bears key from 10 to 11. Lastly, you can remove elements from a dictionary by using the **del** keyword. By doing **del animals["lions"]** you would remove the key value pair from the animals dictionary.

16.3. Iterating over the Contents of a Dictionary

It probably won't come as a surprise that, just like with strings lists and tuples, you can use for loops to iterate through the contents of a dictionary. Let's see how this looks in action.

Video 37 (3:49 *Iterating over the Contents of a Dictionary*)

```
file_counts = {"jpg":10, "txt":14, "csv":2, "py":23}
for extension in file_counts:
    print (extension)
```

```
jpg
txt
csv
pv
```

So if you use a dictionary in a *for* loop, the iteration variable will go through the keys in the dictionary. If you want to access the associated values, you can either use the keys as indexes of the dictionary or you can use the `items` method, which returns a tuple for each element in the dictionary. The tuple's first element is the key. Its second element is the value.

Let's try that with our example dictionary.

```
for ext, amount in file_counts.items():
```

```
print("There are {} files with the .{} extension".format(amount,
ext))
```

There are 10 files with the .jpg extension

There are 14 files with the .txt extension

There are 2 files with the .csv extension

There are 23 files with the .py extension

Sometimes you might just be interested in the keys of a dictionary. Other times you might just want the values. You can access both with their corresponding dictionary methods like this:

```
file_counts.keys()
dict_keys(['jpg', 'txt', 'csv', 'py'])
file_counts.values()
dict_values([10, 14, 2, 23])
```

These methods return special data types related to the dictionary, but you don't need to worry about what they are exactly. You just need to iterate them as you would with any sequence.

```
for value in file_counts.values():
    print(value)
```

10

14

2

23

So we can use items to get key value pairs, keys to get the keys, and values to get just the values. Not too hard, right?

Because we know that each key can be present only once, dictionaries are a great tool for counting elements and analyzing frequency. Let's check out a simple example of counting how many times each letter appears in a piece of text:

```
def count_letters(text):
    result = {}
    for letter in text:
        if letter not in result:
            result[letter] = 0
        result[letter] += 1
    return result
```

In this code, we're first initializing an empty dictionary, then going through each letter in the given string. For each letter, we check if it's not already in the dictionary. In that case, we initialize an entry in the dictionary with a value of zero. Finally, we increment the count for that letter in the dictionary.

To sum up, we've created a dictionary where the keys are each of the letters present in the string and the values are how many times each letter is present. Let's try out a few example strings.

```
count_letters("aaaaa")
{'a': 5}
```

```
count_letters("tenant")  
{'t': 2, 'e': 1, 'n': 2, 'a': 1}
```

```
count_letters("a long string with a lot of letters")  
{'a': 2, ' ': 7, 'l': 3, 'o': 3, 'n': 2, 'g': 2, 's': 2, 't': 5, 'r': 2, 'i': 2,  
'w': 1, 'h': 1, 'f': 1, 'e': 2}
```

Here you can see how the dictionary can have any number of entries and the pairs of key values always count how many of each letter there are in the string. Also, do you see how our simple code doesn't distinguish between actual letters and special characters like a space? To only count the letters, we'd need to specify which characters we're taking into account.

This technique might seem simple at first, but it can be really useful in a lot of cases. Let's say for example that you're analyzing logs in your server and you want to count how many times each type of error appears in the log file. You could easily do this with a dictionary by using the type of error as the key and then incrementing the associated value each time you come across that error type.

Are you starting to see how dictionaries can be a really useful tool when writing scripts? In section 16.5, we're going to learn how to tell when to use dictionaries and when to use lists.

16.4. Iterating Over Dictionaries Recap

You can iterate over dictionaries using a *for* loop, just like with strings, lists, and tuples. This will iterate over the sequence of keys in the dictionary. If you want to access the corresponding values associated with the keys, you could use the keys as indexes.

Or you can use the **items** method on the dictionary, like This method returns a tuple for each element in the dictionary, where the first element in the tuple is the key and the second is the value.

If you only wanted to access the keys in a dictionary, you could use the **keys()** method on the dictionary: If you only wanted the values, you could use the **values()** method:

16.5. Dictionaries versus Lists

Dictionaries and lists are both really useful and each have strengths in different situations. So when is it best to use a list and when is the dictionary the way to go? Think about **the kind of information you can represent in each data**

If you've got a list of information you'd like to collect and use in your script then the list is probably the right approach. For example, if you want to store a series of IP addresses to ping, you could put them all into a list and iterate over them.

Or if you had a list of host names and their corresponding IP addresses, you might want to pair them as key values in a dictionary. Because of the way dictionaries work, it's super easy and fast to search for an element in them.

Let's say you have a dictionary that has usernames as keys, and the groups they belong to as values. It doesn't matter if you have 10 users or 10,000 users, accessing the entry for a given user will take the same time. Amazing, but this isn't true for lists.

If you've got a list of 10 elements, and you need to check if one element is in the list, it'll be a very fast check but if your list has 10,000 elements it'll take significantly longer to check if the element you're looking for is there. So **in general, you want to use dictionaries when you plan on searching for a specific**

Another interesting difference is **the types of values that we can store in lists and** In you can store **any data** In we can store **any data type for the values** but the **keys are restricted to specific** The reasoning behind which types are allowed can get complex and I don't want to bog you down with unnecessary details.

So as a rule of thumb, you can use any immutable data type; numbers, booleans, strings and tuples as dictionary But you can't use lists or dictionaries for that.

On the flip side, like we said, the values associated with keys can be any type, including lists or even other dictionaries. You can use them to represent more complex data structures like directory trees in the file system.

There's a ton of different key value pairs that we need to work with in system administration. So I use dictionaries all the time. They're especially useful with large data sets. When I need to write a script that gets specific keys out of it to manipulate or modify the associated value.

But it doesn't always need to be that serious. One-time, just for fun, I wanted to be able to look up which Disney villain is associated with each protagonist. So I created a dictionary that stores a key like Snow White, with the value, evil queen. Pretty good.

There are even more data types available that we haven't checked out yet. One of these data types is a set which is a bit like a

cross between a list and a dictionary.

A set is used when you want to store a bunch of elements and be certain that there are only present once. Elements of a set must also be immutable. You can think of this as the keys of a dictionary with no associated values or you could see it as a list where what matters isn't the order of the elements but whether an element is in the list or not.

Wow, we've covered a lot and we've still only scratched the surface of what dictionaries can do in your scripts. As you progress in your IT career, you'll come across a lot of situations where dictionary is the easiest way to organize your data.

If you're interested, you can learn more about dictionaries in the official documentation. You'll find links to this in the next cheat sheet.

16.6. Cheat Sheet 7: Dictionary Methods

To locate the *Dictionary Methods Cheat Sheet* pdf file, open the “Cheat Sheets” folder. It’s one of the folders inside the course resources folder you downloaded earlier.

16.7. Practice Quiz 12: 5 Questions

1. The `email_list` function receives a dictionary, which contains domain names as keys, and a list of users as values. Fill in the blanks to generate a list that contains complete email addresses (e.g. `diana.prince@gmail.com`). – 1 point

```
emails = []
```

2. The `groups_per_user` function receives a dictionary, which contains group names with the list of users. Users can belong to multiple groups. Fill in the blanks to return a dictionary with the users as keys and a list of their groups as values. – 1 point

```
user_groups = {}
```

```
# Go through group_dictionary
```

```
# Now go through the users in the group
```

```
user_groups[user] = []
```

```
# Now add the group to the the list of  
# groups for this user, creating the entry  
# in the dictionary if necessary
```

3. The `dict.update` method updates one dictionary with the items coming from the other dictionary, so that existing entries are replaced and new entries are added. What is the content of the dictionary “wardrobe“ at the end of the following code? – *1 point*

```
wardrobe.update(new_items)
```

```
{'jeans': ['white'], 'scarf': ['yellow'], 'socks': ['black', 'brown']}  
{'shirt': ['red', 'blue', 'white'], 'jeans': ['white'], 'scarf': ['yellow'],  
'socks': ['black', 'brown']}  
{'shirt': ['red', 'blue', 'white'], 'jeans': ['blue', 'black', 'white'], 'scarf':  
['yellow'], 'socks': ['black', 'brown']}  
{'shirt': ['red', 'blue', 'white'], 'jeans': ['blue', 'black'], 'jeans':  
['white'], 'scarf': ['yellow'], 'socks': ['black', 'brown']}
```

4. What’s a major advantage of using dictionaries over lists? – *1 point*

Dictionaries are ordered sets

Dictionaries can be accessed by the index number of the element

Elements can be removed and inserted into dictionaries

It's quicker and easier to find a specific element in a dictionary

5. The `add_prices` function returns the total price of all of the groceries in the dictionary. Fill in the blanks to complete this function. – 1 point

```
# Initialize the variable that will be used for the calculation
```

```
# Iterate through the dictionary items
```

```
# Add each price to the total calculation
```

```
# Hint: how do you access the values of
```

```
# dictionary items?
```

```
# Limit the return value to 2 decimal places
```

16.7.1. Answers to Practice Quiz 12

1.

```
emails = []
```

2.

```
user_groups = {}
```

```
# Go through group_dictionary
```

```
# Now go through the users in the group
```

```
    user_groups[user] = []  
    user_groups[user].append(group)
```

```
# Now add the group to the the list of
```

```
# groups for this user, creating the entry  
# in the dictionary if necessary
```

3. B. The `dict.update` method updates the dictionary (`wardrobe`) with the items coming from the other dictionary (`new_items`), adding new entries and replacing existing entries.

4. D. Because of their unordered nature and use of key value pairs, searching a dictionary takes the same amount of time no matter how many elements it contains.

5.

```
# Initialize the variable that will be used for the calculation
```

```
# Iterate through the dictionary items
```

```
# Add each price to the total calculation
```

```
# Hint: how do you access the values of
```

dictionary items?

total += price

Limit the return value to 2 decimal places

17. Module Review

17.1. Basic Structures Wrap Up

In this module, we've covered the basic structures we can use to make the most of our Python scripts; strings, lists, and dictionaries. We've also called out a couple of associated data types like tuples and sets. Knowing your way around these structures lets you solve interesting problems with your programs.

As we keep saying, the key to mastering them and knowing when to use one or the other is practice. The more you write scripts that use these concepts, the easier it will become to pick the right one when you need it.

So how are you feeling? We just learned a lot of new concepts and it's totally normal to feel a little overwhelmed. If you're feeling confident that's awesome and if you're starting to think this is too hard for me I'll never get it, that's also completely normal.

We all felt like that at some point when learning how to code. First off, you will get this. Second, if you're feeling a little iffy on any of the content we've covered so far, now is the time to revise the previous sections or re-watch the videos.

Believe me, you'll be amazed by how much you've learned so far and a second review is usually all you need to understand what might seem a little tricky right now. Jobs in IT require problem-solving and perseverance. You wouldn't be here right now if you didn't have the grit to learn how to script. So stick with it.

I promised you that it'll only get easier and easier. To wrap up, we've got a graded assessment to help you put all your new knowledge to the test. Take it once you feel ready to. Take your time and remember, you've got this!

17.2. Module 4 Graded Assessment – 10 Questions

It's time again for your next graded assessment. Open the "Graded Assessments" folder you downloaded earlier. It contains the pdf files of all the graded assessments in this course. Here's the file name to search:

Module 4 Graded Assessment – *file name*

17.2.1. Solutions to Module 4 Graded Assessment

I (or any member of my team) is available to help you grade your assessments. You can use my help link (email) at the end of chapter 25 to send your assessment for grading. We will get back to you in 12 to 24 hours with your result.

However, if you cannot wait, you can open the “Graded Assessments” folder you downloaded earlier. It contains the pdf formats of the solutions to all graded assessments in this course. You can use them to grade your assessments by yourself. Just be honest as you grade. Here’s the file name to search:

Module 4 Graded Assessment Solutions – *file name*

Module 5

You are an object, at least in Python's view, and you belong to the human class...

18. Object-oriented Programming (OOP)

18.1. OOP Introduction

Welcome back and congrats on making it this far. Our journey together is getting more and more interesting, don't you think? Let's take a second to review what you've accomplished so far. We've now gone over all the basic syntax of Python and then checked out the most common data structures, strings, lists, and dictionaries.

These let us do a bunch of cool things like processing text, iterating through elements to do an operation on each, finding out the frequency of an element and a whole lot more.

In this chapter, we're going to focus on a bunch of new concepts. We're going to dive into **object-oriented programming** which is a way of thinking about and implementing our code. We'll discuss how to create our own objects and how to use many of Python's interesting capabilities.

We're going to learn a lot of new terminology too. As usual, we'll include cheat sheets in the references and in the recaps for you to refer to whenever you need a quick refresher. As always, if something isn't clear right away, remember that you can review the content and do the practice exercises as many times as you need.

Okay. Ready for your orientation on object-oriented programming? There's a lot to cover, so let's jump right in!

18.2. What is OOP?

Imagine you have to describe an apple to someone who's never seen one before, how would you do it? What would you say, besides that it's delicious? You might start off by saying that an apple is a type of fruit. You might talk about how there are lots of different kinds of apples, each with its own color, flavor, and name.

Well, when you're explaining concepts to your computer, it's a good idea to approach it in a similar way. Your computer has no idea what an apple is, or even what a fruit can be. If you want your computer to understand these things, you have to describe them in your programs and scripts.

Up to now, we've discussed elements of syntax, like variables, functions, loops, and some more complex data structures, like lists and dictionaries. These are powerful tools in an IT specialist's toolbox, but it can still be difficult to translate real-world concepts, like what's an apple, or what's a user account into programs.

To make it easier for computers to understand these new concepts, Python uses a programming pattern called **object-oriented** which models concepts using classes and objects. This is a flexible, powerful paradigm where **classes represent and define** while **objects are instances of**

In our apple example, we can have a class called apple that defines the characteristics of an apple. We could then have a

bunch of instances of that apple class, which are the individual objects of that class.

The idea of object-oriented programming might sound abstract and complex, but you've actually been using objects already without even realizing it. Almost everything in Python is an object, all of the numbers, strings, lists, and dictionaries we've seen so far, and have used in our exercises and quizzes, have been objects.

Each of them was an instance of a class representing a concept. The core, apple pun-intended concept of object-oriented programming comes down to attributes and methods associated with a type. The **attributes are the characteristics associated to a** and the **methods are the functions associated to a** In the apple example, the attributes are the color and flavor.

What would the methods be? Well, it depends on what we're going to do with apple. We could maybe have a cut method that turns one whole apple into four slices, or we could have an eat method that reduces the amount of apple available with every bite.

Let's think about a more IT focused example, like a file in our computer. A file has lots of attributes, it has a name, a size, the date it was created, permissions to access it, its contents, and a whole lot more. There are actually so many different file attributes, that Python has multiple classes to deal with files.

The typical file object focuses on the file's contents, and so this object has a bunch of methods to read and modify what's inside the file. Hopefully, these examples help make object-oriented programming a little clearer, but don't worry if you haven't fully wrapped your head around it. In the next section, we'll explore how to apply these concepts to some classes and objects we've already used in Python, which will help us dig a little deeper into how this all works.

18.3. Definition of OOP

In object-oriented programming, concepts are modeled as classes and objects. An idea is defined using a class, and an instance of this class is called an object. Almost everything in Python is an object, including strings, lists, dictionaries, and numbers.

When we create a list in Python, we're creating an object which is an instance of the list class, which represents the concept of a list. Classes also have attributes and methods associated with them. Attributes are the characteristics of the class, while methods are functions that are part of the class.

18.4. Classes and Objects in Python

Remember how we use the type function when checking what type a certain variable was? Let's do that again now:

Video 38 (2:36 *Classes and Objects in Python*)

```
type(0)
'int'>
type("")
'str'>
```

When we use the type function as we just did here, Python tells us which class the value or variable belongs to. Since this is a class, it has a bunch of attributes and methods associated with it. Let's take the string class 'str'> for an example. In this case, the only attribute is the content of the string.

What about the methods? Well, in earlier sections, we looked at a bunch of methods provided by the string class, like *upper()* to create an uppercase version of the string; and *isnumeric()* which checks whether or not the contents are all numeric.

Each string we've used in Python up to now has been a different instance of the string class. They all had the same methods, but the contents were different. This meant that the result of calling those methods was different also.

You can get your computer to list all the attributes and methods in a class. To do that just use the *dir* function. This gets the Interpreter to print to the screen a list of all the attributes and methods:

```
dir("")
```

```
['__add__', '__class__', '__contains__', '__delattr__', '__dir__',  
'__doc__', '__eq__', '__format__', '__ge__', '__getattr__',  
'__getitem__', '__getnewargs__', '__gt__', '__hash__', '__init__',  
'__init_subclass__', '__iter__', '__le__', '__len__', '__lt__', '__mod__',  
'__mul__', '__ne__', '__new__', '__reduce__', '__reduce_ex__',  
'__repr__', '__rmod__', '__rmul__', '__setattr__', '__sizeof__',  
'__str__', '__subclasshook__', 'capitalize', 'casefold', 'center', 'count',  
'encode', 'endswith', 'expandtabs', 'find', 'format', 'format_map',  
'index', 'isalnum', 'isalpha', 'isascii', 'isdecimal', 'isdigit', 'isidentifier',  
'islower', 'isnumeric', 'isprintable', 'isspace', 'istitle', 'isupper', 'join',  
'ljust', 'lower', 'lstrip', 'maketrans', 'partition', 'removeprefix',  
'removesuffix', 'replace', 'rfind', 'rindex', 'rjust', 'rpartition', 'rsplit',  
'rstrip', 'split', 'splitlines', 'startswith', 'strip', 'swapcase', 'title',  
'translate', 'upper', 'zfill']
```

Well, that's a lot of items! Let's break this down a little bit, so we all understand what's going on.

The first bunch, from `'__add__'` to are special methods that begin and end with double underscores. These methods are not usually called by those weird names. Instead, they're called by some of the internal Python functions.

For example, the `'__len__'` method is called by the `len` function that we've used before to find out the length of a string. The `'__ge__'` method is used to compare if one string is greater than or equal to another, when using the `>=` operator.

After the special methods, we see a lot of string methods that we've already come across. This list gives the names of all the methods, but it doesn't tell us how we can use them. There's a different function to tell us that, which is called `help`. Let's give that one a go.

```
help("")
```

Help on class str in module builtins:

```
class str(object)
```

```
| str(object='') -> str
```

```
| str(bytes_or_buffer[, encoding[, errors]]) -> str
```

```
|
```

```
| Create a new string object from the given object. If encoding  
or
```

```
| errors is specified, then the object must expose a data buffer
```

| that will be decoded using the given encoding and error handler.

| Otherwise, returns the result of `object.__str__()` (if defined)

| or `repr(object)`.

| encoding defaults to `sys.getdefaultencoding()`.

| errors defaults to

When we use the help function on any variable or value, we're showing all the documentation for the corresponding class. In this case, we're looking at the documentation for the *str* class, the class of the string object. As before, it starts with the special method. If we scroll down, we reach the ones we've already seen.

We can see the documentation for a bunch of methods, and it tells us the parameters that method receives and the type of return value. It also includes an explanation of what the method does.

For the `count(...)` method, we can see that it receives the substring that will be counted, and it has optional *start* and *end* arguments to indicate which slice of the string would be looked at. We know they're optional because they're written between square brackets [

In general, being able to read and understand a method's documentation is super important when you're writing your own code. Using the *dir* and *help* functions puts all the documentation right at your fingertips. This makes it so much easier to figure out how to use something for the first time. When you're done looking at documentation, you can just type **q** to

Python comes with a lot of classes already predefined for us, which is super useful. But the power of object-oriented programming is that we can also **define our own classes** with their own attributes and methods.

While you might not need to do this when writing a simple script, as your programs grow in complexity, object-oriented programming will help you get the most out of the language, and that includes being able to define your own classes.

Up next, we'll dive into how to write our own class definitions with their own attributes and methods. Let's get to it!

18.5. Classes and Objects in Detail

We can use the **type()** function to figure out what class a variable or value belongs to. For example, **type(" ")** tells us that this is a string class. The only attribute in this case is the string value, but there are a bunch of methods associated with the class. We've seen the **upper()** method, which returns the string in all uppercase, as well as **isnumeric()** which returns a boolean telling us whether or not the string is a number.

You can use the **dir()** function to print all the attributes and methods of an object. Each string is an instance of the string class, having the same methods of the parent class. Since the content of the string is different, the methods will return different values. You can also use the **help()** function on an object, which will return the documentation for the corresponding class. This will show all the methods for the class, along with parameters the methods receive, types of return values, and a description of the methods.

18.6. Defining New Classes

We called out earlier that the point of object-oriented programming is to help define a real-world concept in a way that the computer understands. Defining a real-world concept and code can be tricky. So let's look at how we might go about representing a concept in Python code. We'll take it step-by-step and keep it simple. Let's take our apple example from earlier.

Video 39 (3:53 *Defining New Classes*)

We could use this code to define a basic Apple class.

```
class Apple:
```

Sure, it doesn't look like much but with these two lines we've defined our first-class. Let's check out the syntax. In Python, we use the **class** reserved keyword to tell the computer that we're starting a new class. We follow this with the name of the class and a colon.

The Python style guidelines recommend that **class names should start with a capital**. So we'll be using that convention. In this case, our class is called `Apple`. Class definitions follow the same pattern of other blocks we've seen before like, functions, loops or conditional branches.

After the line with the class definition comes the body of the class, which is **indented to the** In this case, we haven't added anything to the body yet, so we use the `pass` keyword, to show that the body is empty. We can also use the same keyword as a placeholder in any empty Python block.

So how might we expand our definition of the apple class? Well, it would probably have the same attributes that represent the information we want to associate with an apple, like color and flavor. We can add that information like this:

```
class Apple:
    color = ""
    flavor = ""
```

So here we're defining two attributes: *color* and We define them as strings because that's what we expect these attributes to be. At the moment, they're empty strings, since we don't know what values these attributes will have. See how we don't need the *pass* keyword anymore now that we've got an actual body for the class.

All right. Now that we've got an Apple class and some attributes, let see our Apple in action:

```
jonagold = Apple()
```

Here, we're creating a new instance of our Apple class and assigning it to a variable called Check out the syntax. To create a

new instance of any class, we call the name of the class as if it were a function.

Now that we've got our shiny new apple object, let's set the values of the attributes:

```
jonagold.color = "red"  
jonagold.flavor = "sweet"
```

All right. We've just set the color and the flavor as string values. To check that it worked, let's try retrieving them both and printing them to the screen:

```
print(jonagold.color)  
red  
print(jonagold.flavor)  
sweet
```

The syntax used to access the attributes is called **dot notation** because of the dot used in the expression. Dot notation lets you access any of the **abilities that the object might** called **methods** or **information that it might store** called like flavor.

The attributes and methods of some objects can be other objects and can have attributes and methods of their own. For example, we could use the *upper()* method to turn the string of the color attribute to uppercase. So,

```
print (jonagold.color.upper())
```

RED

So far we've created one instance of the Apple class and set its attributes and checked that they are now correctly set. Now, we could create a new instance of the Apple class with different attributes.

```
golden = Apple()  
golden.color = "yellow"  
golden.flavor = "soft"
```

Both *golden* and *jonagold* are **instances of the Apple**. They have the same attributes, color and flavor. But those attributes have different values.

Congrats. You've learned how to create your own classes. Let's check that we've got all this down with a quick quiz. After that, we're going to learn how to define new methods for a class.

18.7. Defining Classes Recap

We can create and define our classes in Python similar to how we define functions. We start with the **class** keyword, followed by the name of our class and a colon. Python style guidelines recommend class names to start with a capital letter. After the class definition line is the class body, indented to the right. Inside the class body, we can define attributes for the class.

Let's take our Apple class example:

...

We can create a new instance of our new class by assigning it to a variable. This is done by calling the class name as if it were a function. We can set the attributes of our class instance by accessing them using dot notation. Dot notation can be used to set or retrieve object attributes, as well as call methods associated with the class.

```
>>> jonagold = Apple()
```

We created an Apple instance called jonagold, and set the color and flavor attributes for this Apple object. We can create another instance of an Apple and set different attributes to differentiate between two different varieties of apples.

```
>>> golden = Apple()
```

We now have another Apple object called golden that also has color and flavor attributes. But these attributes have different values.

18.8. Practice Quiz 13: 5 Questions

1. Let's test your knowledge of using dot notation to access methods and attributes in an object. Let's say we have a class called Birds. Birds has two attributes: color and number. Birds also has a method called count() that counts the number of birds (adds a value to number). Which of the following lines of code will correctly print the number of birds? Keep in mind, the number of birds is 0 until they are counted! – 1 point

A.

```
bluejay.number = 0  
print(bluejay.number)
```

B.

```
print(bluejay.number.count())
```

C.

```
bluejay.count()  
print(bluejay.number)
```

D.

```
print(bluejay.number)
```

2. Creating new instances of class objects can be a great way to keep track of values using attributes associated with the object. The values of these attributes can be easily changed at the object level. The following code illustrates a famous quote by George

Bernard Shaw, using objects to represent people. Fill in the blanks to make the code satisfy the behavior described in the quote. – 1 point

```
# "If you have an apple and I have an apple and we exchange these apples then  
# you and I will still each have one apple. But if you have an idea  
# and I have
```

```
# an idea and we exchange these ideas, then each of us will have two ideas."
```

```
# George Bernard Shaw
```

```
johanna = Person()
```

```
martin = Person()
```

```
#Here, despite G.B. Shaw's quote, our characters have started with  
#different amounts of apples so we can better observe the results.
```

```
#We're going to have Martin and Johanna exchange ALL their apples with  
#one another.
```

```
#Hint: how would you switch values of variables,
```

```
#so that "you" and "me" will exchange ALL their apples with one another?
```

```
#Do you need a temporary variable to store one of the values?
```

```
#You may need more than one line of code to do that, which is OK.
```

```
you.apples, me.apples = me.apples, you.apples
```

#"you" and "me" will share our ideas with one another.

#What operations need to be performed, so that each object receives

#the shared number of ideas?

#Hint: how would you assign the total number of ideas to

#each idea attribute? Do you need a temporary variable to store

#the sum of ideas, or can you find another way?

#Use as many lines of code as you need here.

```
you.ideas += me.ideas
```

```
me.ideas = you.ideas
```

```
exchange_apples(johanna, martin)
```

```
exchange_ideas(johanna, martin)
```

3. The City class has the following attributes: name, country (where the city is located), elevation (measured in meters), and population (approximate, according to recent statistics). Fill in the

blanks of the `max_elevation_city` function to return the name of the city and its country (separated by a comma), when comparing the 3 defined instances for a specified minimal population. For example, calling the function for a minimum population of 1 million: `max_elevation_city(1000000)` should return "Sofia, Bulgaria". – 1 point

```
# define a basic city class
```

```
# create a new instance of the City class and  
# define each attribute  
city1 = City()
```

```
# create a new instance of the City class and  
# define each attribute  
city2 = City()
```

```
# create a new instance of the City class and  
# define each attribute  
city3 = City()
```

```
# Initialize the variable that will hold  
# the information of the city with  
# the highest elevation  
return_city = City()
```

```
# Evaluate the 1st instance to meet the requirements:
```

```
# does city #1 have at least min_population and
```

```
# is its elevation the highest evaluated so far?
```

```
# Evaluate the 2nd instance to meet the requirements:
```

```
# does city #2 have at least min_population and
```

```
# is its elevation the highest evaluated so far?
```

```
    return_city = city2
```

```
# Evaluate the 3rd instance to meet the requirements:
```

```
# does city #3 have at least min_population and
```

```
# is its elevation the highest evaluated so far?
```

```
    return_city = city3
```

```
#Format the return string
```

4. What makes an object different from a class? – 1 point

An object represents and defines a concept

An object is a specific instance of a class

An object is a template for a class

Objects don't have accessible variables

5. We have two pieces of furniture: a brown wood table and a red leather couch. Fill in the blanks following the creation of each Furniture class instance, so that the describe_furniture function can format a sentence that describes these pieces as follows:

"This piece of furniture is made of {color} {material}" – 1 point

```
table = Furniture()
```

```
couch = Furniture()
```

```
# Should be "This piece of furniture is made of brown wood"
```

```
# Should be "This piece of furniture is made of red leather"
```

18.8.1 Answers to Practice Quiz 13

1. C. We must first call the `count()` method, which will populate the number attribute, allowing us to print number and receive a correct response.

2.

“If you have an apple and I have an apple and we exchange these apples then

you and I will still each have one apple. But if you have an idea and I have

an idea and we exchange these ideas, then each of us will have two ideas.”

George Bernard Shaw

```
johanna = Person()
```

```
martin = Person()
```

#Here, despite G.B. Shaw's quote, our characters have started with #different amounts of apples so we can better observe the results.

#We're going to have Martin and Johanna exchange ALL their apples with #one another.

#Hint: how would you switch values of variables,

#so that "you" and "me" will exchange ALL their apples with one another?

#Do you need a temporary variable to store one of the values?

#You may need more than one line of code to do that, which is OK.

```
you.apples, me.apples = me.apples, you.apples
```

#"you" and "me" will share our ideas with one another.

#What operations need to be performed, so that each object receives

#the shared number of ideas?

#Hint: how would you assign the total number of ideas to

#each idea attribute? Do you need a temporary variable to store

#the sum of ideas, or can you find another way?

#Use as many lines of code as you need here.

```
you.ideas += me.ideas
```

```
me.ideas = you.ideas
```

```
exchange_apples(johanna, martin)
```



```
exchange_ideas(johanna, martin)
```

3.

```
# define a basic city class
```

```
# create a new instance of the City class and
```

```
# define each attribute
```

```
city1 = City()
```

```
# create a new instance of the City class and
```

```
# define each attribute
```

```
city2 = City()
```

```
# create a new instance of the City class and
```

```
# define each attribute
```

```
city3 = City()
```

```
# Initialize the variable that will hold
```

```
# the information of the city with
```

```
# the highest elevation
```

```
    return_city = City()
```

```
# Evaluate the 1st instance to meet the requirements:
```

```
# does city #1 have at least min_population and
```

```
# is its elevation the highest evaluated so far?
```

```
    return_city = city1
```

```
# Evaluate the 2nd instance to meet the requirements:
```

```
# does city #2 have at least min_population and
```

```
# is its elevation the highest evaluated so far?
```

```
    return_city = city2
```

```
# Evaluate the 3rd instance to meet the requirements:
```

```
# does city #3 have at least min_population and
```

```
# is its elevation the highest evaluated so far?
```

```
    return_city = city3
```

```
#Format the return string
```

4. B. Objects are an encapsulation of variables and functions into a single entity.

5.

```
table = Furniture()
```

```
couch = Furniture()
```

```
# Should be "This piece of furniture is made of brown wood"
```

```
# Should be "This piece of furniture is made of red leather"
```

19. Classes and Methods

19.1 Instance Methods

How are you doing so far? Is everything making sense? Are all those apple examples making you hungry? Feel free to pause and grab a snack if that's what you need.

We called out earlier that we use methods to get objects to do stuff. We've seen several methods in our example so far like *lower()* for strings, *append()* for lists or *values()* for dictionaries.

The key to understanding methods is this.

Video 40 (3:41 *Instance Methods*)

Methods are functions that operate on the attributes of a specific instance of a

When we call the `append` method on a list, we're adding an element to the end of that specific list and not to any other lists. When we call the `lower` method on the string, we're making the contents of that specific string lowercase. How exactly does this happen? Let's take a closer look by defining our own methods. First, we need to define a class and create an instance of it like we've done before.

```
class Piglet:
    pass
hamlet = Piglet()
```

Nice. We've created a piglet class. While our new piglet might be cute, it can't do a whole lot now. What if we wanted to give it a voice? For objects to perform actions, they need methods and as we called out before, a method is a function that operates on a single instance of an object. Let's add a method to our class.

```
class Piglet:  
    def speak(self):  
        print("oink oink")
```

You can see here that we start defining a method with the *def* keyword just like we would for a function. See how the line with the *def* keyword is indented to the right inside the Piglet class? That's how we define a function as a **method of the**

This function is receiving a parameter called *self*. This parameter represents the instance that the method is being executed on. Let's try this out and see what happens.

```
hamlet = Piglet()  
hamlet.speak()  
oink oink
```

As you can see, the piglet goes oink, oink! But this makes the piglet say the same thing for all instances of the class. Boring? Let's make the method do something different depending on the attribute of the instance:

```
class Piglet:
    name = "piglet" # We introduce an attribute/variable called
                    # name with a default value of piglet to initialize it..
    def speak(self):
        I'm {}. Oink!".format(self.name))
```

This time we've studied the body of the class by defining an attribute called `name` with a default value of `"piglet"`. We can change that value later but it's a good idea to set it now to make sure our variable is initialized. If you look closely at how we wrote the new `speak` method in the last line, you'll see that it's using the value of **`self.name`** to know what name to print.

This means that it's accessing the attribute name from the current instance of `Piglet`. Let's try this out.

```
hamlet = Piglet()
hamlet.name = "Hamlet"
```

```
hamlet.speak()
Oink! I'm Hamlet! Oink!
```

Meet Hamlet, our python pig! You didn't know pigs could talk? Well, they can in Python!

In this example, the `speak` method printed the name `Hamlet` which was the name attribute that we set. What if we create a new instance of the same class but with a different name? It should generate a different output. Let's try this out. I think Hamlet needs a friend:

```
petunia = Piglet()
petunia.name = "Petunia"
petunia.speak()
Oink! I'm Petunia! Oink!
```

We've now created two instances of the piglet class each of them with their own name. When calling this speak method, each of them prints their name and not the other.

Variables that have different values for different instances of the same class are called instance just like the name variable in this example.

Since methods are just functions that belong to a specific class, they can work as any other function. So they can receive more parameters and return values if needed. Let's check out what a method returning a value looks like:

```
class Piglet:
    years = 0
    def pig_years(self):
        return self.years * 18
```

In this case, we've created a method that converts the age of our piglet to pig years. So the value that the method returns should change when we change the years attribute of our instance. Let's create an instance and check how this method works:

Piggy is two years old in human years, how old is he in pig years?

```
piggy = Piglet()  
print(piggy.pig_years())  
0
```

```
piggy.years = 2  
print(piggy.pig_years())  
36
```

So as the value of the years attribute changes, the return value of the pig years method changes too.

In section 19.3, we're going to learn about a few special types of methods including one in particular called

19.2. What Is a Method?

Calling methods on objects executes functions that operate on attributes of a specific instance of the class. This means that calling a method on a list, for example, only modifies that instance of a list, and not all lists globally.

We can define methods within a class by creating functions inside the class definition. These instance methods can take a parameter called **self** which represents the instance the method is being executed on. This will allow you to access attributes of the instance using dot notation, like `obj.name` which will access the name attribute of that specific instance of the class object.

When you have variables that contain different values for different instances, these are called **instance**

19.3. Constructors and Other Special Methods

Up to now, we've been creating classes with empty or default values and their attributes, and then setting the attribute values after we've created the object. This works, but it's not ideal. Working this way means we need to write a separate line for each attribute we want to set, and that makes it really easy to forget to set an important value.

Us humans are pros at forgetting to do things, even important things. So when writing code, it's a good idea to do things early to prevent from forgetting them later on. So let's set those values as we create the instance. This way, we know that our instance has all the important values in it from the moment is created and we don't have to worry about it.

To do this, we need to use a special method called `__init__`. Let's go back to our apple example to see this in action:

Video 41 (2:23 *Constructors and Other Special Methods*)

```
class Apple:
    def __init__(self,color,flavor): #Put 2 underscores after 2
underscores after
        self.color = color
        self.flavor = flavor
```

The constructor of the class is the method that's called when you call the name of the class. It's always named `__init__`. You might remember that all methods that start and end with **two underscores** are **special**

Here, we've defined a constructor, one very important special method. This method on top of the *self* variable that represents the instance receives two more parameters: *color* and

Then the method sets those values as the values of the current instance. Let's see how that works with the new instance of the apple class.

```
jonagold = Apple("red", "sweet")
```

Great. Now, let's check that all the attributes are set correctly.

```
print(jonagold.color)
red
```

Perfect! So, now by adding a constructor method that sets the attributes, we can create the class and have its values set right when it's created. Pretty handy, right?

Constructors aren't the only special methods we can write. When we use the *str* or *print* functions to convert an object to a string, we are using a super-useful special method. But before we go ahead and define one, let's see what happens when we don't define it.

```
print(jonagold)
<__main__.Apple object at 0x7fc4a58d5c18>
```

We just tried to print our apple instance, and we got a very weird message. We have the words *Apple* and *object* in there, but what's the rest of it? Well, when we don't specify a way to print an object, Python uses the **default method** that prints the position where the object is stored in the computer's memory (yours will be different). This is definitely not what we wanted.

If you ever try and print something and Python prints a random string of numbers and letters, you'll know that it's likely using the default representation, which is the position of the object in the computer's memory.

So how do we tell Python to print something that makes sense for us? We use the special *str* method which returns the string that we want to print. Let's see what this looks like:

```
class Apple:
    def __init__(self,color,flavor):
        self.color = color
        self.flavor = flavor
    def __str__(self):
        return apple is {} and its flavor is {}".format(self.color,
self.flavor)
```

By defining the special **str** we're telling Python that we want it to display when the print function is called with an instance of our class. Check it out!

```
jonagold = Apple("red", "sweet")  
print(jonagold)  
This apple is red and its flavor is sweet
```

So the str method lets us print a friendly message instead of a bunch of numbers. In general, it's a good idea to think ahead and define the str method when creating objects that you want to print.

There are a lot of other special methods. We're not going to look at the rest of them here, but you can find pointers to learn more about them in the official documentation. You'll find the link to that in the next section. These concepts are new and not too easy. So don't worry if you're still trying to figure out the difference between a method and a function.

We've all been there. The best way to feel confident is to keep practicing until it's clear. You're doing great. So keep at it!

19.4. Special Methods Recap

Instead of creating classes with empty or default values, we can set these values when we create the instance. This ensures that we don't miss an important value and avoids a lot of unnecessary lines of code. To do this, we use a special method called a `__init__`. Below is an example of an `Apple` class with a constructor method defined.

When you call the name of a class, the constructor of that class is called. This constructor method is always named `__init__`. You might remember that special methods start and end with two underscore characters. In our example above, the constructor method takes the `self` variable, which represents the instance, as well as `color` and `flavor` parameters. These parameters are then used by the constructor method to set the values for the current instance. So we can now create a new instance of the `Apple` class and set the `color` and `flavor` values all in `go`:

```
Red
```

In addition to the `__init__` constructor special method, there is also the `__str__` special method. This method allows us to define how an instance of an object will be printed when it's passed to the `print()` function. If an object doesn't have this special method defined, it will wind up using the default representation, which will

print the position of the object in memory. Not super useful.
Here is our Apple class, with the `__str__` method added:

...

Now, when we pass an Apple object to the print function, we get a nice formatted string:

This apple is red and its flavor is sweet. It's good practice to think about how your class might be used and to define a `__str__` method when creating objects that you may want to print later.

19.5. Documenting Functions, Classes and Methods

The world of classes and methods can be a little puzzling when you're still learning your way around, and that's why the `help` function can come in handy. You might remember that we can still use the Python function `help` to find documentation about classes and methods. We can also do this on our own classes, methods, and functions. Let's check this out.

Video 42 (2:07 *Documenting Functions Classes and Methods*)

```
class Apple:
    def __init__(self,color,flavor):
        self.color = color
        self.flavor = flavor
    def __str__(self):
        return "This apple is {} and its flavor is
        {}".format(self.color, self.flavor)
```

We'll start with the `Apple` class we used before, and now we'll ask for some help:

```
help(Apple)
```

```
class Apple(builtins.object)
```

```
| Apple(color, flavor)
```

|

| Methods defined here:

|

| `__init__(self, color, flavor)`

| Initialize self. See `help(type(self))` for accurate signature.

|

| -----

| Data descriptors defined here:

|

| `__dict__`

| dictionary for instance variables (if defined)

|

| `__weakref__`

| list of weak references to the object (if defined)

See how when we asked for help on our class we got a list of the methods that are defined in the class? In this example, the defined methods are the constructor and the conversion to string. But this documentation is super short and to be honest, it doesn't explain a whole lot. So let's go back to the interpreter by typing `q`.

We want our methods, classes, and functions to give us more information when we or someone else use the help function. We can do that by adding a

A docstring is a brief text that explains what something Let's see how this works with a simple function.

```
def to_seconds(hours, minutes, seconds):
```

First off, we want to define the function. We give it the parameters `hours`, `minutes`, and `seconds`. After that, we add our docstring. We do this by typing a string between triple quotes and we indent it to the right like the body of the function.

```
    """Returns the amount of seconds in a given hours, minutes and seconds"""
```

Next, we write the code for our function.

```
    return hours*3600+minutes*60+seconds
```

So there we have it, we have a function with a docstring in its body. Let's see how we can use the *help* function to see it.

```
help(to_seconds)
```

```
Help on function to_seconds in module __main__:
```

```
to_seconds(hours, minutes, seconds)
```

```
    Returns the amount of seconds in a given hours, minutes  
    and seconds
```

Success! The help function shows us the string we wrote. As we called out earlier, we can add docstrings to classes and methods too. Let's use our piglet class to see what this would look like:

```
class Piglet:  
    """Represents a piglet that can say their name"""  
    years = 0  
    name = ""  
    def speak(self):  
        """Outputs a message including the name of the piglet."""  
        print("Oink. I'm {}! Oink!".format(self.name))  
  
    def pig_years(self):  
        """Converts the current age to equivalent pig years."""  
        return self.years * 18
```

Now we've got a bunch of helpful information. We've added docstrings for our piglet class and for its methods. Remember

that the docstring always has to be indented at the same level of the block it's documenting. Docstrings are super helpful for figuring out how to use a function you've never used before.

Not only that, if you're reading a piece of code written by someone else, docstrings let us understand the code much better because the classes, methods, and functions are clearly documented. So when writing your code, add docstrings to explain your functions, classes, and methods, it will make a ton of difference to anyone who might use your code.

19.6. Documenting with Docstrings

The Python **help** function can be super helpful for easily pulling up documentation for classes and methods. We can call the **help** function on one of our classes, which will return some basic info about the methods defined in our class:

...

```
| Methods defined here:  
|  
|  
|  
| -----  
| Data descriptors defined here:  
|  
| __dict__  
|  
  
| __weakref__
```

We can add documentation to our own classes, methods, and functions using A docstring is a short text explanation of what something does. You can add a docstring to a method, function, or class by first defining it, then adding a description inside triple quotes. Let's take the example of this function:

...

We have our function called *to_seconds* on the first line, followed by the docstring which is indented to the right and wrapped in triple quotes. Last up is the function body. Now, when we call the help function on our *to_seconds* function, we get a handy description of what the function does:

```
to_seconds(hours, minutes, seconds)
```

Docstrings are super useful for documenting our custom classes, methods, and functions, but also when working with new libraries or functions. You'll be extremely grateful for docstrings when you have to work with code that someone else wrote!

19.7. Cheat Sheet 8: Classes and Methods

To locate the *Classes and Methods Cheat Sheet* pdf file, open the “Cheat Sheets” folder. It’s one of the folders inside the course resources folder you downloaded earlier.

19.8. About Jupyter Notebooks (Optional)

In all our quizzes so far, we've been working with code blocks. Code blocks are a great tool for writing small snippets of code, but now we're tackling more complex problems so we need a more powerful tool. We're going to start using a new tool called **Jupyter** kicking off with the next quiz.

A Jupyter Notebook is a special kind of document that can contain pieces of programming

We can execute these pieces of code inside the notebooks one piece at a time, and the notebooks can also contain other things like text, images, interactive widgets, and a whole lot more. These extra elements allow us to tell an interactive story with our code exercises.

Like Code Blocks, Jupyter Notebooks lets us edit and run our code in the web browser. The difference is that we can add explanations between the code, and also the pieces of code are related to each other.

Jupyter Notebooks are an open-source technology that you can use outside this course, so if you're interested, you could even run it locally on your computer. Links to Jupyter Notebooks full tutorials are given in the next section. But first, let's check out in this video how a Jupyter Notebook works and what you can do with it.

Video 43 (1:46 *About Jupyter Notebook*)

As explained in the video, once we launch it, we will wait until the notebook loads. Once it's loaded, you can see some explanatory text and a bit of code (sample code). We can execute the code by clicking the *run* button in the toolbar, or we can also run it by pressing **Shift-Enter** on our keyboard.

19.9. Help with Jupyter Notebooks

Jupyter notebooks are easy to use. But, if you get stuck, you can find more information from the links below.

If you still need help, the discussion forums are a great place to find it! Use the forums to ask questions and source answers from your fellow learners.

If you want to learn how to install, run and use Jupyter Notebooks, check out these resources:

[Jupyter Notebook](#) by datacamp.com

[How to use Jupyter](#) by codeacademy.com

[Teaching and Learning with](#) by university professors using Jupyter

19.10. Challenge Lab 1: Methods and Classes Lab

The code you are going to use in this lab is shown in the screenshot below. It defines an *Elevator* class. Look for two files (both of which contain the same *Elevator* class code) with file name in your *Challenge Labs* folder. This folder is inside the course resources folder you downloaded earlier.

Open only one of the two files:

C1M5L2_Methods_and_Classes_V3.ipynb file, which you can open in your Jupyter Notebook, or *C1M5L2_Methods_and_Classes_V3.py* which you can open with your Pycharm.

The elevator has a current floor, it also has a top and a bottom floor that are the minimum and maximum floors it can go to. Open one of the two *Elevator* class code versions you prefer to use and study the code to make sure the elevator goes through the floors requested.

```
class Elevator:
    def __init__(self, bottom, top, current):
        """Initializes the Elevator instance."""
        self.bottom = bottom
        self.top = top
        self.current = current
    def up(self):
        """Makes the elevator go up one floor."""
        if self.current < self.top:
            self.current += 1
    def down(self):
        """Makes the elevator go down one floor."""
        if self.current > self.bottom:
            self.current -= 1
    def go_to(self, floor):
        """Makes the elevator go to the specific floor."""
        self.current = floor
    def __str__(self):
        """print the current floor"""
        return "Current floor: {}".format(self.current)

elevator = Elevator(-1, 10, 0)
```

Fig. 19.10.1: The Elevator class definition code block

To test whether your *Elevator* class is working correctly, run the code snippets below:

```
elevator.up()  
elevator.current #should output 1
```

```
elevator.down()
```

```
elevator.current #should output 0
```

```
elevator.go_to(10)  
elevator.current #should output 10
```

If you get a **NameError** message, be sure to run the *Elevator* class definition code block first. If you get an **AttributeError** message, be sure to initialize *self.current* in your *Elevator* class.

Once you've made the above methods output 1, 0 and 10, it means the *Elevator* class and its methods were successfully coded.

For the up and down methods, did you take into account the top and bottom floors? Keep in mind that the elevator shouldn't go above the top floor or below the bottom floor. To check that out, try the code below and verify if it's working as expected. If it's not, then go back and modify the methods so that this code behaves correctly.

```
# Go to the top floor. Try to go up, it should stay. Then go  
down.  
elevator.go_to(10)
```

```
elevator.up()
elevator.down()
print(elevator.current) # should be 9
# Go to the bottom floor. Try to go down, it should stay. Then
go up.
elevator.go_to(-1)
elevator.down()
elevator.down()
elevator.up()
elevator.up()
print(elevator.current) # should be 1
```

Now add the **str** method to your *Elevator* class definition above so that when printing the elevator using the **print()** method, we get the current floor together with a message. For example, in the 5th floor it should say "Current floor: 5"

```
elevator.go_to(5)
print(elevator) # it should say "Current floor: 5"
```

Remember, Python uses the default method, that prints the position where the object is stored in the computer's memory. If your output is something like:

```
object at 0x7ff6a9ff3fd0>
```

Then you will need to add the special **str** method, which returns the string that you want to print. Try again until you get the desired output, "Current floor: 5".

Once you have successfully produced the desired output, you are all done with this challenge lab. Awesome!

20. Code Reuse

20.1. Inheritance

Wow, we've covered a bunch of new stuff in these last few sections. You're doing great. We've learned all about object-oriented programming, and how to define our own classes and methods, including special methods like constructors or string conversions.

We've also learned how to document them all. We're now going to talk about another aspect of object-oriented programming called Just like people have parents, grandparents, and so on, objects have an ancestry.

The principle of inheritance let's a programmer build relationships between concepts and group them together. In particular, this allows us to reduce code duplication by generalizing our code. For example, how could we develop our apple representation to include other types of fruit, too?

Well, one thing we know about an apple is that it's a fruit. So we could define a separate fruit class. We also know that all fruits have a color and taste. So what if we moved our color and flavor attributes into the fruit class?

Video 44 (3:45 *Inheritance*)

```
class Fruit:
    def __init__(self, color, flavor):
```

```
self.color = color
self.flavor = flavor
```

Here, we have a fruit class with a constructor for the color and flavor attributes. Now, we can rewrite our apple class and easily add another fruit into the mix, too.

```
class Apple(Fruit):
    pass
```

```
class Grape(Fruit):
    pass
```

In Python, we use parentheses in the class declaration to show an inheritance relationship. For our new fruit classes, we've used that syntax to tell our computer that both the apple and the grape classes inherit from the fruit class. Because of this, they automatically have the same constructor, which sets the color and flavor attributes.

You can think of the fruit class as the parent class, and the apple and grape classes as siblings. Let's see this in action. First, we create an instance of the apple class:

```
granny_smith = Apple("sweet", "tart")
carnelian = Grape("purple", "sweet")
```

Then, to check that this actually worked, let's print the attributes values.

```
print(granny_smith.flavor)
tart
print(carnelian.color)
purple
```

With the inheritance technique, we can use the fruit class to store information that applies to all kinds of fruit, and keep apple or grape specific attributes in their own classes. For example, we could have an attribute to track how much of an apple is left after it's partially eaten.

Of course, this applies to both attributes and methods. If a class has an attribute or a method defined in it, inheriting classes will have the same attributes and methods defined in them. But we can also get them to behave differently depending on what we change.

To explore this, let's go back to our piglet example and change it so that there's a base animal class.

```
class Animal:
    sound = ""
    def __init__(self,name):
        self.name = name
    def speak(self):
        print("{sound} I'm {name}!
{sound}".format(name=self.name, sound=self.sound))
```

```
class Piglet(Animal):
    sound = "Oink!"
```

In this code, we've defined a general class called `Animal` which has an attribute to store the sound that the animal makes. The constructor of the class takes the name that will be assigned to the instance when it's created. There's also a *speaking* method that prints the name of the animal together with the sound the animal makes.

Then, we have a *piglet* class that inherits from the *Animal* class. We set the value of the sound attribute to **oink** in the piglet class, and that's the only thing we've modified from the original. Everything else is inherited. Let's see this in action.

```
hamlet = Piglet("Hamlet")
print(hamlet.speak())
Oink! I'm Hamlet! Oink!
```

Let's define a new class that also inherits from *Animal*. How about a *Cow* class?

```
class Cow(Animal):
    sound = "Moouoo"
```

Cool. To finish, let's create an instance of this class to make it speak.

```
milky = Cow("Milky White")
```

```
print(milky.speak())  
Moooo I'm Milky White! Moooo
```

So you can see that we can easily define new classes that inherit from the base animal class and use both the attributes and methods that the animal class provides. Pretty cool, right? Let's think of a different example, something closer to what you might be doing at your day-to-day job.

In a system that handles the employees at your company, you may have a class called `Employee` which could have the attributes for things like full name of the person, the username used in company systems, the groups the employee belongs to, and so on.

The employee class could have methods to do a bunch of things, like check if an employee belongs to a certain group, or create an email address based on the name and username attributes. The system could also have a *manager* class. A manager is an employee, but has additional information associated with it, like the employees that report to a specific manager.

Are you starting to get an idea of the power of inheritance?

Inheritance lets you reuse code written for one class in other Next up, we're going to talk about a different way of reusing code.

20.2. Object Inheritance

In object-oriented programming, the concept of inheritance allows you to build relationships between objects, grouping together similar concepts and reducing code duplication. Let's create a custom Fruit class with color and flavor attributes:

...

We defined a Fruit class with a constructor for color and flavor attributes. Next, we'll define an Apple class along with a new Grape class, both of which we want to inherit properties and behaviors from the Fruit class:

...

...

In Python, we use parentheses in the class declaration to have the class inherit from the Fruit class. So in this example, we're instructing our computer that both the Apple class and Grape class inherit from the Fruit class. This means that they both have the same constructor method which sets the color and flavor attributes. We can now create instances of our Apple and Grape classes:

tart

purple

Inheritance allows us to define attributes or methods that are shared by all types of fruit without having to define them in each fruit class individually. We can then also define specific attributes or methods that are only relevant for a specific type of fruit. Let's look at another example, this time with animals:

...
...
...

We defined a parent class, `Animal`, with two animal types inheriting from that class: `Piglet` and `Cow`. The parent `Animal` class has an attribute to store the sound the animal makes, and the constructor class takes the name that will be assigned to the instance when it's created. There is also the `speak` method, which will print the name of the animal along with the sound it makes. We defined the `Piglet` and `Cow` classes, which inherit from the `Animal` class, and we set the sound attributes for each animal type. Now, we can create instances of our `Piglet` and `Cow` classes and have them speak:

```
>>> hamlet.speak()  
...  
...  
>>> milky.speak()
```

We create instances of both the `Piglet` and `Cow` class, and set the names for our instances. Then we call the `speak` method of each

instance, which results in the formatted string being printed; it includes the sound the animal type makes, along with the instance name we assigned.

20.3. Composition

We talked about how inheritance creates an ancestry for our objects. To check for this ancestry, we can use the “is a” rule. An apple is a fruit, a piglet is an animal. They inherit the attributes and methods of their parent class and so they allow us to reduce code duplication, but what if you have a relationship between classes, where one class isn't a child of the other? Sounds confusing? Let's check out an example to get a better idea of this.

Say we have a *package* class that represents a software package which could be installed on every machine on our network. This class has a lot of information on the software, like the name, the version, the size, and more. We also have a *repository* class that represents all the packages that we have available for installation internally.

In this class, we want to know how many packages there are and what's the total size of all the packages. In this case, the repository isn't a package and the package isn't a repository. Instead, the repository contains packages.

To model this within our code, the repository class will have an attribute that could be a list or a dictionary, which will contain instances of the package class. So for this scenario, we'll make use of the code in the other classes by calling their methods. This is what's called Let's see this in action.

We'll first create a repository class that starts with an empty dictionary of packages when it's created. The dictionary will have the names of the packages as keys and the package objects as values. Watch the illustration in this video.

Video 45 (4:55 *Composition*)

```
class Repository:
```

```
    def __init__(self):  
        self.packages = {}
```

Nice. We have our class, which starts with an empty dictionary of packages. You might be wondering why we are adding the dictionary in the constructor instead of directly to the class. The answer to this might be a bit tricky to understand. So let's go back to our juicy apple example to help make sure this is clear.

We defined earlier a class called *Apple* and set some basic attributes for it, like color and flavor. All instances of the apple class will be initialized with the values that we preset for those attributes.

If we change the color of one apple from red to golden, we substitute the old value with the new one. Super-important to remember, this action happens only for that particular instance. But what if this apple has a worm in it? What if we wanted our apple class to also have a list of worms?

If we created the list when constructing the class, then all instances of the apple class would have the same exact list. So if we added a worm to the list, it would get added to the one list that's shared by all instances.

To avoid this, we need to create the list at the time of creating the instance, instead of when creating the class. By doing this, each instance will have its own list independent of the others. This happens with all attributes that are mutable, because when we modify immutable attribute, we're not replacing a value with another, we're changing the contents of the original attribute.

In our repository case, the packages attribute is a dictionary, which is mutable. We'll be modifying its contents by adding and removing elements in it. If we created it at the class level, all instances of the repository class would use the same dictionary, and items added or removed would affect all instances at the same time.

If that's still a bit confusing, don't worry. I was also confused the first time I came across it. Just take your time. Re-watch this video if you need and remember this rule of thumb: **always initialize mutable attributes in the**

Now, we've got our dictionary, but how will we add packages to it? We'll create an *add_package* method (line 4).

```
1 class Repository:
2     def __init__(self):
```

```
3     self.packages = {}
4     def add_package(self, package):
5         self.packages[package.name] = package
```

Now, we can add packages to the dictionary. We could also write a similar method to remove the packages, but I bet you can work that out without my help. Let's do something more interesting instead.

We said that the packages had a size attribute that holds the size in bytes that the software package requires. So how could we calculate the size of the whole repository? We need to iterate over the packages contained in the dictionary, adding up all their sizes. I would go something like this:

```
6 def total_size(self):
7     result = 0
8     for package in self.packages.values():
9         result += package.size
    return result
```

I will explain the above code as follows. We're going to add up all the sizes. So the first thing we need to do, is create a *result* variable (line 7) that we'll use to sum up the values. So, awesome. We have our result initialized.

We now need to go through all the packages in the dictionary. Remember, the **keys** are the **package names** and the **values** are

the **package** For our calculation, we only care about the objects. So we'll retrieve them with the *values()* dictionary method (line 8).

Now, for each package, we want to add the size to the result variable (line 9). Nice. We're almost done. We just need to return the result now (line 10).

Take a look at the method we've just written. It's a method inside the repository class, that's making use of the values method in the dictionary class and it's accessing the size attribute in the package class. That is the power of composition. When we have other objects as attributes, we can use all their attributes and methods to get our own code to do what we want.

Wow. That was pretty complex. Chances are, you won't get it the first time around. Most of us don't. So if you're worried you might have missed something, take your time to review the contents. We want you to feel confident before moving on. In section 20.5, where we're going to talk about a different kind of code we use using Python modules.

20.4. Object Composition

You can have a situation where two different classes are related, but there is no inheritance going on. This is referred to as **composition** -- where one class makes use of code contained in another class. For example, imagine we have a **Package** class which represents a software package. It contains attributes about the software package, like name, version, and size. We also have a **Repository** class which represents all the packages available for installation.

While there's no inheritance relationship between the two classes, they are related. The Repository class will contain a dictionary or list of Packages that are contained in the repository. Let's take a look at an example Repository class definition:

```
...         result += package.size
```

In the constructor method, we initialize the packages dictionary, which will contain the package objects available in this repository instance. We initialize the dictionary in the constructor to ensure that every instance of the Repository class has its own dictionary.

We then define the *add_package* method, which takes a Package object as a parameter, and then adds it to our dictionary, using the package name attribute as the key.

Finally, we define a `total_size` method which computes the total size of all packages contained in our repository. This method iterates through the values in our repository dictionary and adds together the size attributes from each package object contained in the dictionary, returning the total at the end.

In this example, we're making use of `Package` attributes within our `Repository` class. We're also calling the `values()` method on our packages dictionary instance. Composition allows us to use objects as attributes, as well as access all their attributes and methods.

20.5. Python Modules

So far, we've been using the features that are baked into the Python language. The basic statements like `if`, `for`, `while`, or the definition of functions or classes, are part of the language and are ready for us to use whenever we need them. The same goes for integers, floats, strings, lists, and dictionaries. They're all part of the basic Python language because they're used so often. Of course, this isn't enough to get interesting things done.

We'll need a lot of additional tools like being able to send packets over the network, read files from our machine, process images, or who knows what you might want to do to make your work more effective. To organize the code we need to perform tasks like these.

Python provides an abstraction called a module. **Modules can be used to organize functions, classes, and other data together in a structured**

Video 46 (3:24 *Python Modules*)

Internally, modules are set up through separate files containing the necessary classes and functions. Python already comes with a bunch of **ready-to-use** All these modules are contained in a group called the **Python standard** Let's see how we can use some of them.

First, we'll use the **import** keyword to import the **random** This module is useful for generating random numbers or making random choices.

```
import random
```

Now that we've imported the module, let's use a function provided by this module called

```
random.randint(1, 10)
```

```
1
```

```
random.randint(1, 10)
```

```
4
```

```
random.randint(1, 10)
```

```
6
```

This function receives two parameters and generates a random number between the two parameters that we pass. In this case, we're generating a random number between 1 and 10. As you can see, this function returns *different* numbers each time it's called. Pretty fun, right?

The syntax used for calling a function provided by a module is similar to calling a method provided by a class. It **uses a dot to separate the name of the module and the function provided by that**

Let's try using a different module, the **datetime** module. We use this for handling dates and times. Now, let's get the current date.

```
import datetime
now = datetime.datetime.now()
type(now)
'datetime.datetime'>
```

If you're wondering why we have a doubled datetime, it's because the datetime module provides a datetime class, and the datetime class gives us a method called This now method generates an instance of the datetime class for the current time.

We can operate on this instance of datetime in a bunch of ways. Let's check out a couple of examples.

```
print(now)
```

```
2021-06-29 14:24:54.667220
```

When we call print with an instance of the datetime class, we see the date printed in a specific format. Behind the scenes, the print function is calling the str method of the datetime class which formats it in the way that we see here.

We can also access the instance through its attributes and methods. For example, we can look at the individual parts of the date like the year, like this:

```
now.year
2021
```

The `datetime` module provides more classes than the `datetime` class. For example, we can use the **`timedelta`** class to calculate a date in the future or in the past. Let's try this out.

```
print(now + datetime.timedelta(days=28))  
2021-07-27 14:34:52.989521
```

In this case, we're creating an instance of the `timedelta` class with a value of 28 days, then we're adding it to the instance of the `datetime` class that we already had and printing out the result.

There's a lot more things available in the `datetime` and `random` modules. If you're interested in learning more, you can read the whole reference. It's available online and I will include a link in section 20.7. This is just a sneak peek into what you could do with modules. You can also develop your own. We'll talk more about that in Part 2 of this course. For now, just focus on using existing Python modules.

20.6. Augmenting Python with Modules

Python modules are separate files that contain classes, functions, and other data that allow us to import and make use of these methods and classes in our own code. Python comes with a lot of modules out of the box. These modules are referred to as the **Python Standard**. You can make use of these modules by using the **import** keyword, followed by the module name. For example, we'll import the **random** module, and then call the **randint** function within this module:

```
8  
7  
1
```

This function takes two integer parameters and returns a random integer between the values we pass it; in this case, 1 and 10. You might notice that calling functions in a module is very similar to calling methods in a class. We use dot notation here too, with a period between the module and function names.

Let's take a look at another module: This module is super helpful when working with dates and times.

```
>>> now = datetime.datetime.now()  
2021-06-29 14:24:54.667220
```

First, we import the module. Next, we call the **now()** method which belongs to the **datetime** class contained within the **datetime** module. This method generates an instance of the datetime class for the current date and time. This instance has some methods which we can call:

```
>>> now.year  
2019  
2021-07-27 14:34:52.989521
```

When we call the print function with an instance of the datetime class, we get the date and time printed in a specific format. This is because the datetime class has a **__str__** method defined which generates the formatted string we see here. We can also directly call attributes and methods of the class, as with **now.year** which returns the year attribute of the instance.

Lastly, we can access other classes contained in the datetime module, like the **timedelta** class. In this example, we're creating an instance of the timedelta class with the parameter of 28 days. We're then adding this object to our instance of the datetime class from earlier and printing the result. This has the effect of adding 28 days to our original datetime object.

20.7. Supplemental Reading for Code Reuse

The official Python documentation lists all the modules included in the standard library. It even has a turtle in it...

[PyPI](#) is the Python repository and index of an impressive number of modules developed by Python programmers around the world.

20.8. Challenge Lab 2: Code Reuse Lab

Let's put what we learned about code reuse all together. You can do this lab right here in this book. Alternatively, you can do it in your Jupyter Notebooks or Pycharm.

If you choose to do it in your Jupyter Notebooks or Pycharm, look for two files (both of which contain the same *Code Reuse* scripts) with file name *C1M5L3_Code_Reuse_V2* in your *Challenge Labs* folder. This folder is inside the course resources folder you downloaded earlier.

Open only one of the two files: if you want to use your Jupyter Notebook, or if you want to use your Pycharm.

First, let's look back at Run the following cell that defines a generic *Animal* class.

```
class Animal:
    name = ""
    category = ""
    def __init__(self, name):
        self.name = name
    def set_category(self, category):
        self.category = category
```

What we have is not enough to do much yet. That's where you come in. For the next block, define a *Turtle* class that inherits

from the *Animal* class. Then go ahead and set its category. For instance, a turtle is generally considered a reptile. Although modern cladistics call this categorization into question, for purposes of this exercise we will say turtles are reptiles!

When you are done, your result should look like this:

```
class Turtle(Animal):  
  
    name = ""  
    category = "reptile"
```

Run the following cell to check whether you correctly defined your *Turtle* class and set its category to reptile.

```
print(Turtle.category)
```

Was the output of the above cell reptile? If not, go back and edit your *Turtle* class making sure that it inherits from the *Animal* class and its category is properly set to reptile. Be sure to re-run that cell once you've finished your edits. Did you get it? If so, great!

Next, let's practice **composition** a little bit. This one will require a second type of *Animal* that is in the same category as the first. For example, since you already created a *Turtle* class, go ahead and create a *Snake* class. Don't forget that it also inherits from the *Animal* class and that its category should be set to reptile.

```
class Snake(Animal):
```



```
name = ""  
category = "reptile"
```

Now, let's say we have a large variety of *Animals* (such as turtles and snakes) in a Below we have the *Zoo* class. We're going to use it to organize our various Remember, inheritance says a *Turtle* is an but a *Zoo* is not an *Animal* and an *Animal* is not a *Zoo* -- though they are related to one another.

In the *Zoo* class below you can use **zoo.add_animal()** to add instances of the *Animal* subclasses you created above. Once you've added them all, you should be able to use **zoo.total_of_category()** to tell you exactly how many individual *Animal* types the *Zoo* has for each category! Be sure to run the cell once you've finished your edits.

Run the following cell to check your *Zoo* class.

```
turtle = Turtle("Turtle") #create an instance of the Turtle class  
snake = Snake("Snake") #create an instance of the Snake class
```

```
zoo.add_animal(turtle)  
zoo.add_animal(snake)
```

```
print(zoo.total_of_category("reptile")) #how many zoo animal types  
in the reptile category
```

Was the output of the above cell If not, go back and edit the *Zoo* class making sure to use the appropriate attributes. Be sure to re-

run that cell once you've finished your edits.

Did you get it? If so, perfect! You have successfully defined your *Turtle* and *Snake* subclasses as well as your Zoo class. You are all done with this lab. Great work!

21. Module Review

21.1. OOP Wrap Up

Object orientation is not easy to understand. So congratulations on getting through these Concepts. Let's quickly recap the main Concepts we've just covered.

Video 47 (1:44 *OOP Wrap Up*)

We've learned that in an object-oriented language like python real-world concepts are represented by We know that **instances of classes** are usually called That **objects have attributes** which are used to store information about them and we can make objects do work by calling their

We've also learned that we can access attributes and methods using **dot** We then dove into objects can be organized by and how they can be contained inside each other using

Wow, that really is a lot of new stuff. Congratulations on sticking with it. Objects are a great way for programmers to **model real world** They let us have functions that work on specific things like reading a file, setting the subject for an email, calculating the size of a repository of packages and so on.

Isn't it cool to see how all of this is coming together? As a sysadmin, the objects ideal with the most represent individual users and their accounts. I use them to group lots of different

properties that help me turn abstract code into tangible interactions.

I also use objects in my code to group functions based on the data they act upon. For example, I recently needed to write a bunch of functions that were all operating on some specific file attributes. So I used a class to group all those functions making my code clearer and more reusable. Super helpful, right? I thought so.

21.2. Challenge Lab 3: Practice Notebook (Object Oriented Programming)

You can do this lab right here in this book. Alternatively, you can do it in your Jupyter Notebooks or Pycharm.

If you choose to do it in your Jupyter Notebooks or Pycharm, look for two files (both of which contain the same *Code Reuse* scripts) with file name `C1M5_Object_Oriented_Programming_V7` in your *Challenge Labs* folder. This folder is inside the course resources folder you downloaded earlier.

Open only one of the two files: if you want to use your Jupyter Notebook, or if you want to use your Pycharm.

In this exercise, we'll create a few classes to simulate a server that's taking connections from the outside and then a load balancer that ensures that there are enough servers to serve those connections.

To represent the servers that are taking care of the connections, we'll use a *Server* class. Each connection is represented by an id, that could, for example, be the IP address of the computer connecting to the server. For our simulation, each connection creates a random amount of load in the server, between 1 and 10.

Run the following code that defines this *Server* class.

```
#Begin Portion 1#
```

```
import random
```

```
class Server:
```

```
    def __init__(self):
```

```
        """Creates a new server instance, with no active
connections."""
```

```
        self.connections = {}
```

```
    def add_connection(self, connection_id):
```

```
        """Adds a new connection to this server."""
```

```
        connection_load = random.random()*10+1
```

```
        # Add the connection to the dictionary with the
calculated load
```

```
    def close_connection(self, connection_id):
```

```
        """Closes a connection on this server."""
```

```
        # Remove the connection from the dictionary
```

```
    def load(self):
```

```
        """Calculates the current load for all connections."""
```

```
        total = 0
```

```
        # Add up the load for each of the connections
```

```
        return total
```

```
    def __str__(self):
```

```
        """Returns a string with the current load of the server"""
```

```
        return "{:.2f}%".format(self.load())
```

```
#End Portion 1#
```

Now run the following cell to create a `Server` instance and add a connection to it, then check the load:

```
server = Server()  
server.add_connection("192.168.1.1")
```

```
print(server.load())
```

After running the above code cell, if you get a **NameError** message, be sure to run the `Server` class definition code block first.

The output should be `0`. This is because some things are missing from the `Server` class. So, you'll need to go back and fill in the blanks to make it behave properly.

Go back to the `Server` class definition and fill in the missing parts for the *add_connection* and *load* methods to make the cell above print a number different than zero. As the load is calculated randomly, this number should be different each time the code is executed.

Hint: Recall that you can iterate through the values of your connections dictionary just as you would any sequence.

Great! If your output is a random number between 1 and 10, you have successfully coded the *add_connection* and *load* methods of

the Server class. Well done!

What about closing a connection? Go back to the Server class definition and work on the *close_connection* method to make the following code work correctly:

```
server.close_connection("192.168.1.1")
print(server.load())
```

You have successfully coded the *close_connection* method if the cell above prints 0.

Hint: Remember that *del* dictionary[key] removes the item with key *key* from the dictionary.

Alright, we now have a basic implementation of the server class. Let's look at the basic **LoadBalancing** class. This class will start with only one server available. When a connection gets added, it will randomly select a server to serve that connection, and then pass on the connection to the server. The LoadBalancing class also needs to keep track of the ongoing connections to be able to close them. This is the basic structure:

```
#Begin Portion 2#
class LoadBalancing:
    def __init__(self):
        """Initialize the load balancing system with one server"""
        self.connections = {}
        self.servers = [Server()]
```

```

def add_connection(self, connection_id):
    """Randomly selects a server and adds a connection to
it."""
    server = random.choice(self.servers)
    # Add the connection to the dictionary with the selected
server
    # Add the connection to the server

def close_connection(self, connection_id):
    """Closes the connection on the the server corresponding
to connection_id."""
    # Find out the right server
    # Close the connection on the server
    # Remove the connection from the load balancer
    for connection in self.connections.keys():
        if connection == connection_id:
            server_ = self.connections[connection]
            server_.close_connection(connection_id)
            del self.connections[connection_id]

def avg_load(self):
    """Calculates the average load of all servers"""
    # Sum the load of each server and divide by the amount
of servers
    result = 0

    count = 0
    for server in self.servers:
        result += server.load()
        count += 1

```

```

        return result/count

    def ensure_availability(self):
        """If the average load is higher than 50, spin up a new
server"""
        if self.avg_load() >= 50:
            self.servers.append(Server())

    def __str__(self):
        """Returns a string with the load for each server."""
        loads = [str(server) for server in self.servers]
        return "[{}].format(", ".join(loads))
#End Portion 2#

```

As with the Server class, this class is currently incomplete. You need to fill in the gaps to make it work correctly. For example, this snippet should create a connection in the load balancer, assign it to a running server and then the load should be more than zero:

```

l = LoadBalancing()
l.add_connection("fdca:83d2::f2od")
print(l.avg_load())

```

After running the above code, the output is 0. Work on the *add_connection* and *avg_load* methods of the LoadBalancing class to make this print the right load. Be sure that the load balancer now has an average load more than 0 before proceeding.

What if we add a new server?

```
l.servers.append(Server())
```

```
print(l.avg_load())
```

The average load should now be half of what it was before. If it's not, make sure you work on the *add_connection* and *avg_load* methods so that this code works correctly.

Hint: You can iterate through the all servers in the *self.servers* list to get the total server load amount and then divide by the length of the *self.servers* list to compute the average load amount.

Fantastic! Now what about closing the connection?

```
l.close_connection("fdca:83d2::f2od")  
print(l.avg_load())
```

Fill in the code of the *LoadBalancing* class to make the load go back to zero once the connection is closed.

Great job! Before, we added a server manually. But we want this to happen automatically when the average load is more than 50%. To make this possible, work on the *ensure_availability* method and call it from the *add_connection* method after a connection has been added. You can test it with the following code:

```
for connection in range(20):  
    l.add_connection(connection)
```

```
print(l)
```

The code above adds 20 new connections and then prints the loads for each server in the load balancer. If you coded correctly, new servers should have been added automatically to ensure that the average load of all servers is not more than 50%.

Run the following code to verify that the average load of the load balancer is not more than 50%.

```
print(l.avg_load())
```

Awesome! If the average load is indeed less than 50%, you are all done with this assessment.

Module 6

“Operations keeps the lights on, strategy provides a light at the end of the tunnel, but project management is the train engine that moves the organization forward.” - Joy Gumz

22. Writing a Script from the Ground Up

22.1. Final Project Introduction

Congratulations on making it here! You're almost at the end of the course. It's been an interesting and rewarding journey don't you think? Along the way you've learned the basics of Python syntax, including functions, conditionals, and for and while loops.

You've also learned how to use the most common data types, like integers strings, lists, and dictionaries. You even learned about object-oriented programming. Now we're going to put all this knowledge together to solve more fun and exciting problems.

We're going to approach these new challenges like they're real-world problems. We need to solve with a script. By doing this we'll see yet another example of how these programming skills can make the work we do in our IT jobs faster and more efficient.

In the next sections, we'll check out how to go about solving a more complex problem by writing a script from the ground up. To do that we'll go step by step using a recommended way for dealing with more advanced challenges.

Back at the beginning of the course, I told you a little bit about the first Python script I ever wrote. It all started with a problem, my teams on call person was getting paged too much. Being on call is drawing the ultimate short straw. Whenever an issue springs up the person on call needs to be there to put out the fire.

It's an exhausting challenge. So to help alleviate some of the stress we wanted to build a better monitoring dashboard. Getting it done took a lot of refactoring, debugging, and testing. It wasn't easy, but thankfully I didn't have to start from scratch.

I had the help of my teammates and many thousands of people who posted similar struggles on the Internet. When the dashboard was finally up and running, the on-call person wasn't the only one breathing a sigh of relief.

To start solving our problem, we'll first look at the problem statement where we'll get an understanding of what we need to do and the inputs and outputs for the script we'll need to write. Then we'll do some research.

We'll think about how we can tackle the problem with the tools already baked into Python. Remember that we always want to avoid reinventing the wheel. No matter how tricky and intricate the challenge appears, chances are that others have solved a similar one before.

So it's valuable to spend some time tapping into the resources that exist to help us solve our problem. Once we know what we need to write and what we can use it to do, we'll do some planning. We'll think about what data types will be useful for our solution and how we're going to operate on them.

Finally, we'll do the actual writing of the script and then we'll check that the code does what it's supposed to do. Sound good? When we take the structured approach to tackling problems there really isn't a challenge too complex to solve. So let's get started!

22.2. Problem Statement

Video 48 (3:14 *Problem Statement*)

Imagine that you're an IT specialist working in a medium-sized company. Your manager wants to create a daily report that tracks the use of machines. Specifically, she wants to know which users are currently connected to which machines. It's your job to create the report.

In your company, there's a system that collects every event that happens on the machines on the network. Among the many events collected it records each time a user logs in or out of a computer. With this information, we want to write a script that generates a report of which users are logged in to which machines at that time.

Before we jump into solving that problem, we need to know what information we will use as input and what information we will have as output. We can work this out by looking at the rest of the system where our script will live.

In our report scenario, the **input is a list of each event is an instance** of the event class. An event class contains the date when the event happened, the name of the machine where it happened, the user involved, and the event type.

In this scenario, we care about the login and logout event type. All right, that's good to know. But we need to know exact names of the attributes, otherwise, we won't be able to access them. The **attributes** are called and

The **event types** are **strings** and the ones we care about are login and logout. With that we should have enough information about the input of our script. Our script will receive a list of event objects and we'll access the events attributes. We'll then use that information to know if a user is currently logged into a machine or not.

Let's talk about the output. We want to generate a report that lists all the machine names and for each machine, lists of the users that are currently logged in. We then want this information printed on the screen. We've been tasked with generating a report and we can decide exactly how we want that report to look.

One option would be to print the name of the machine at the beginning of the line and then list the current users on separate lines, and indent it to the right. Alternatively, we could print the machine name followed by a colon and then the usernames separated by commas all in the same line. We can probably come up with something even more fancy.

When formatting a report, it's easy to get caught up in the making-it-look-good part. I've fallen into that trap but what really matters is how well the script solves the problem. So it's better to first focus on making the program work. You can always spend time making the report look nice later.

Let's keep it simple for now. We will go with the approach of printing the machine name followed by all the current users separated by commas.

Okay, we now have a pretty good idea of what we need to do. We've identified our **problem** which is, **we need to process a list of event objects using their date, type, machine, and user attributes** to generate a report that lists all users currently logged into the machines. We're off to a great start. The next step we're going to do is some research to work out how to best actually do this.

22.3. Research

Video 49 (4:26 *Research*)

Now we have our problem statement which helps us understand the problem and focus our approach. We know we have to input a list of event objects and evaluate these objects attributes to output a report of all the users currently logged into a machine.

Now it's time for step 2, the research. We're going to consider all the tools we have available to help us solve the problem. To find out which users are currently logged into machines, we need to check when they logged in and when they logged out.

If a user logged into a machine and then logged out, they're no longer logged into it. But if they didn't logout yet, they're still logged in. I know. We're stating the obvious here, but in programming, it is super important to be clear on the parameters. Also, knowing this tells us that to solve this correctly, it's vital that we process the events in chronological order. If they're not, we can get the logout event before the corresponding login event and our code may do unpredictable things, and no one likes unpredictable code!

So how do we sort lists in Python? We'll need to do some research. Type “sort lists in Python” into your favorite search engine and you'll get a bunch of results that mentioned the list sort method and the sorted function.

The difference between these two options is that the **sort method modifies the list it's executed on**, while the **sorted function returns a new list that's been** Apart from that, they work the exact same way. Let's check out this difference in action.

First, will create a list of numbers and call the sort method to sort the list.

```
numbers = [4, 6, 2, 7, 1]
numbers.sort()
print(numbers)
[1, 2, 4, 6, 7]
```

You can see here that the elements of the list have been sorted. Let's try a different example now using the **sorted** function. We'll create a list of names.

```
names = ["Carlos", "Ray", "Alex", "Kelly"]
print(names)
['Carlos', 'Ray', 'Alex', 'Kelly']
```

Then we'll print the output of the sorted function.

```
print(sorted(names))
['Alex', 'Carlos', 'Kelly', 'Ray']
```

Let's print the original list again to check that it didn't change.

```
print(names)
['Carlos', 'Ray', 'Alex', 'Kelly']
```

So you can see that the original list was not modified. The sorted function returned a new sorted list, but the original was left untouched. Nice. We now know how to sort things in Python. For this problem, it's fine to modify the original list. So we will use the sort method.

But wait. See how both these options sorted the list alphabetically:

```
[1, 2, 4, 6, 7]
['Alex', 'Carlos', 'Kelly', 'Ray']
```

That's the default approach Python takes. But what if we wanted to organize our lists by different criteria?

Again, if we take a look at the documentation we found online, we'll see that the sort method can take a couple of parameters. One of these parameters is called `key` and it lets us use a function as the sorting key. Let's try this out on our list of names. Instead of sorting alphabetically, we could sort by the **length** of each string.

Do you remember which function we can use to do that? Yes, we can pass the `len()` function as the key.

All right. We now know how to order elements of a list based on the return value of a function. In our report scenario, we know that our elements will be instances of the event class and we want to order by date, which is an attribute of the event class.

One way we could do this is to write a function called *get_event_date* which returns the date stored in the event object. We could also create this as a method in the event class if we had access to modifying the class.

But since we're working with a bigger system that generates these events, we will assume that we can't just add a method to the class. So we'll create our own function instead. How does that sound? Is it all making sense?

Remember that there are various paths we could take to solve this problem. But some are better than others. So it's important to understand why we chose the options we did.

Feel free to take some time on your own to explore the possibilities and understand what we're doing. In the next section, we will dive into our plan to build our script.

22.4. Planning

You're doing great with this so far. We've already defined our problem statement and then we researched options to figure out what tools we have available and which are best for the job. Now it's time to plan our approach.

Video 50 (2:35 *Planning*)

We know that our input will be a list of events and we'll sort them by time. Each event in that list will include a machine name, a username, and tell us whether the event is a login or a logout. We want our script to keep track of users as they log in and out of machines. So how can we do this?

Let's think about what we'll do for each event and see if we can figure out the best strategy. When we process an event, we'll see that someone interacted with a machine.

If it's a we want to **add it to the group of users logged into that**
If it's a we want to **remove it from the group of users logged into the**

In this scenario, it makes sense to use a **set** to store the current users. Adding new users at login time and removing them at logout time. Great. But if the current users of a given machine are stored in a set, how do we know which set corresponds to the machine we're looking for?

The easiest way to know this is to store this information in a dictionary. We'll use the **name of the machine as the key** and the **current users of that machine as the** So for each event we process, we'll first check in the dictionary to see if the machine is already there.

We need to check this because it could be the first time we're processing an event for that machine. If it's not there, we'll create a new entry. If it is, we'll update the existing entry with the action corresponding to the event. This means we either add the user if the event is a login or remove the user if it's a logout.

Once we're done processing the events, we'll want to print a **report** of the information we generated. This is a completely separate task. So it should be a separate function. This function will receive the dictionary regenerated and print the report.

It's important to have **separate functions** to process the data and to print the data to the screen. This is because if we want to modify how the report is printed, we know we only need to change the function in charge of printing.

Or, if we find a bug in our processing the data, we only need to change the processing function. It would also allow us to use the same data processing function to generate a different kind of report, like generating a PDF file, for example.

Now, we know what we need to do, how we need to do it, and how we'll structure our code. Now we can get into the meaty stuff, that is, actually writing the code!

22.5. Writing the Script

We've come a long way to get here, so let's quickly rattle off what we know so far. We know that we need to process the events to generate a report. We know how to sort the list of events chronologically. We know that we'll store the data in a dictionary of sets, which we'll use to keep track of who's logged in where, and that we'll have a function that generates the dictionary and a separate one that prints the dictionary.

I think that's everything. Know what that means? We are finally ready to write our code. A copy of the completed code is inside your *Challenge Labs* folder. Its file name is `challenge_labs.py`. You can open it later in an editor such as Pycharm.

Let's start by defining the helper function that we'll use to sort the list.

Video 51 (5:44 *Writing the Script*)

```
def get_event_date(event):  
    return event.date
```

We will use this simple function as the parameter to the sort function to sort the list. Now, we're ready to start coding are processing function, which we will call `process_events`. The first step is to define the function.

```
def current_users(events):
```

Inside the function, we'll first sort our events by using the `sort` method, and passing the function we just created as the key. Also, before we start iterating through our list of events, we need to create the dictionary where we will store the names and users of a machine.

```
events.sort(key=get_event_date)
```

```
machines = {}
```

Now, we're ready to iterate through our list of events. Next, we want to check if the machine affected by this event is in the dictionary. If it's not, we'll add it with an empty set as the value.

```
for event in events:  
    if event.machine not in machines:  
        machines[event.machine] = set()
```

Now, for the login events, we want to add the user to the list, and for the logout events, we want to remove users from the list. To do this, we're going to use the *add* and *remove* methods, which add and remove elements from a set.

```
if event.type == "login":  
    machines[event.machine].add(event.user)  
elif event.type == "logout" and event.user in  
machines[event.machine]:  
    machines[event.machine].remove(event.user)
```

```
return machines
```

Once we are done iterating through the list of events, the dictionary will contain all machines we've seen as keys. With a set containing the current users of the machines as the values, this function returns the dictionary. We will handle printing in a different function.

We now have the dictionary ready, and we need to print it. For that, we'll create a new function called as shown in the first line of this block which I will explain below.

```
1 def generate_report(machines):
2     for machine, users in machines.items():
3         if len(users) > 0:
4             user_list = ", ".join(users)
5             print("{}: {}".format(machine, user_list))
```

In our report, we want to iterate over the keys and values in the dictionary. To do that, we'll use the method *items* (line 2) that returns both the key and the value for each pair in the dictionary.

Now, before we print anything, we want to ensure that we don't print any machines where nobody is currently logged in. This could happen if a user logged in and then logged out. To avoid that, we tell the computer only to print when the set of users has more than zero elements (line 3).

Now, we said earlier that we want to print the machine name, followed by the users logged into the machine, separated by commas. We generate the string of logged in users for that machine using the method *join* (line 4). Finally, we can generate the string we want using the *format* method (line 5).

Now, we've written all the functions we need to tackle our problem. Did everything makes sense? This is a great moment to pause and review the sections (or videos) for each step in our approach, from problem statement to writing the code. Make sure it's clear, not just which function we're using, but why we're using it.

If anything is a little fuzzy, remember you can contact me or use the discussion forums to ask for help. It's about to get exciting in the next section. We're going to execute this code and see if it works. I'm feeling good about it. Let's put our code to the test!

22.6. Putting It All Together

Video 52 (2:24 *Putting it All Together*)

We're almost done solving our problem. We've written the code that solves our problem statement after following our research and plan. We're now going to put all of our code in a Jupyter notebook, execute it and see what happens. Open your own copy of the code now in your Jupyter Notebook now. Alternatively, you can use your Pycharm.

To check that our code is doing everything it's supposed to do, we need an *Event* For this scenario, we'll use the very simple event class (already included in your code). Here it is:

```
class Event:
    def __init__(self, event_date, event_type, machine_name, user):
        self.date = event_date
        self.type = event_type
        self.machine = machine_name
        self.user = user
```

Now, we have an event class that has a Constructor and sets the necessary attributes. Using this Constructor, we'll create some **sample events** (already included in your code) and add them to a list by running the following cell. Click the run button on your Jupyter Notebook or Pycharm. Here are the sample events to check that the code runs correctly.

```
events = [  
    Event('2020-01-21 12:45:56', 'login', 'myworkstation.local',  
'jordan'),  
    Event('2020-01-22 15:53:42', 'logout', 'webserver.local', 'jordan'),  
  
    Event('2020-01-21 18:53:21', 'login', 'webserver.local', 'lane'),  
    Event('2020-01-22 10:25:34', 'logout', 'myworkstation.local',  
'jordan'),  
    Event('2020-01-21 08:20:01', 'login', 'webserver.local', 'jordan'),  
]
```

Now, we've got a bunch of events. They're currently unsorted, they affect a few machines and include some users. We'll feed these events into our function and see what happens. Everything is now ready to go. Now let's call the code and verify it generates the dictionary.

```
users = current_users(events)  
print(users)
```

```
{'webserver.local': {'lane'}, 'myworkstation.local': set()}
```

Now we will generate the report by running the next cell:

```
print(generate_report(users))  
webserver.local: lane
```

Great! Only one user (lane) is currently logged in.

So our code correctly created a dictionary with the machine names as keys. There's one empty set and two sets with one value. Our report correctly skipped the one machine that had an empty set. That's great.

In the world of IT, there's a bunch of other things that could happen. What happens if we come across an event for a user logging out that had never logged in? What do you think we should do then? We're going to try and figure this out in the next Challenge Lab.

22.7. Challenge Lab 4: Putting It All Together

I want you do this lab in your Jupyter Notebooks. Open your *Challenge Labs* folder and look for two files with the same title. The exact name of each file is Your Challenge Labs folder is inside the course resources folder you downloaded earlier.

You should open only one of the two files. Open *C1M6L1_Putting_It_All_Together.ipynb* if you want to use your Jupyter Notebook which I recommend, or *C1M6L1_Putting_It_All_Together.py* if you prefer to use your Pycharm or other editors.

The copy of the code is below. It is similar to what we wrote in the last section. Go ahead and run the following cell that defines our *current_users* and *generate_report* methods.

```
def get_event_date(event):
    return event.date

def current_users(events):
    events.sort(key=get_event_date)
    machines = {}
    for event in events:
        if event.machine not in machines:
            machines[event.machine] = set()
        if event.type == "login":
            machines[event.machine].add(event.user)
```

```
        elif event.type == "logout" and event.user in
machines[event.machine]:
            machines[event.machine].remove(event.user)
    return machines
```

```
def generate_report(machines):
    for machine, users in machines.items():

        if len(users) > 0:
            user_list = ", ".join(users)
            print("{}: {}".format(machine, user_list))
```

No output should be generated from running the custom function definitions above. To check that our code is doing everything it's supposed to do, we need an *Event*. The code in the next cell below initializes our Event class. Go ahead and run this cell next.

```
class Event:
    def __init__(self, event_date, event_type, machine_name, user):
        self.date = event_date
        self.type = event_type
        self.machine = machine_name
        self.user = user
```

Now, we have an Event class that has a constructor and sets the necessary attributes. Next let's create some events and add them to a list by running the following cell.

```
events = [
```

```
Event('2020-01-21 12:45:56', 'login', 'myworkstation.local',
'jordan'),
Event('2020-01-22 15:53:42', 'logout', 'webserver.local', 'jordan'),
Event('2020-01-21 18:53:21', 'login', 'webserver.local', 'lane'),
Event('2020-01-22 10:25:34', 'logout', 'myworkstation.local',
'jordan'),
Event('2020-01-21 08:20:01', 'login', 'webserver.local', 'jordan'),

Event('2020-01-23 11:24:35', 'logout', 'mailserver.local', 'chris'),
]
```

Now we've got a bunch of events. Let's feed these events into our `custom_users` function and see what happens.

```
users = current_users(events)
print(users)
{'webserver.local': {'lane'}, 'myworkstation.local': set(),
'mailserver.local': set()}
```

Uh oh! The code in the previous cell produces an error message. This is because we have a user in our events list that was **logged out of a machine he was not logged** Do you see which user this is? It's Chris!

Make edits to the first cell containing our custom function definitions to see if you can fix this error message. There may be more than one way to do so.

Remember when you have finished making your edits, rerun that cell as well as the cell that feeds the `events` list into our

`custom_users` function to see whether the error message has been fixed. Once the error message has been cleared and you have correctly outputted a dictionary with machine names as keys, your custom functions are properly finished. Great!

Now try generating the report by running the next cell:

```
print(generate_report(users))  
webserver.local: lane
```

Whoop whoop! Success! The error message has been cleared and the desired output is produced. You are all done with this practice notebook. Way to go!

23. Final Project

23.1. Final Project Overview

Video 53 (3:21 *Final Project Overview*)

Wow, first off, a huge congrats! You're about to start the final project of the course. Amazing job making it all the way here. I hope you've been having as much fun as I have on this journey.

You're going to be working on your final project using only Jupyter Notebooks. You might be starting to feel pretty confident using them. But remember if you have any issues you can always ask me or ask for help in the discussion forums.

Before we dive in, I'm going to explain a little bit about what you'll be doing for the project. It's going to be really fun. The goal of the project is to create a **word** A word cloud is an image that's made up of different sized words. An example is shown in Fig. 23.1.1.

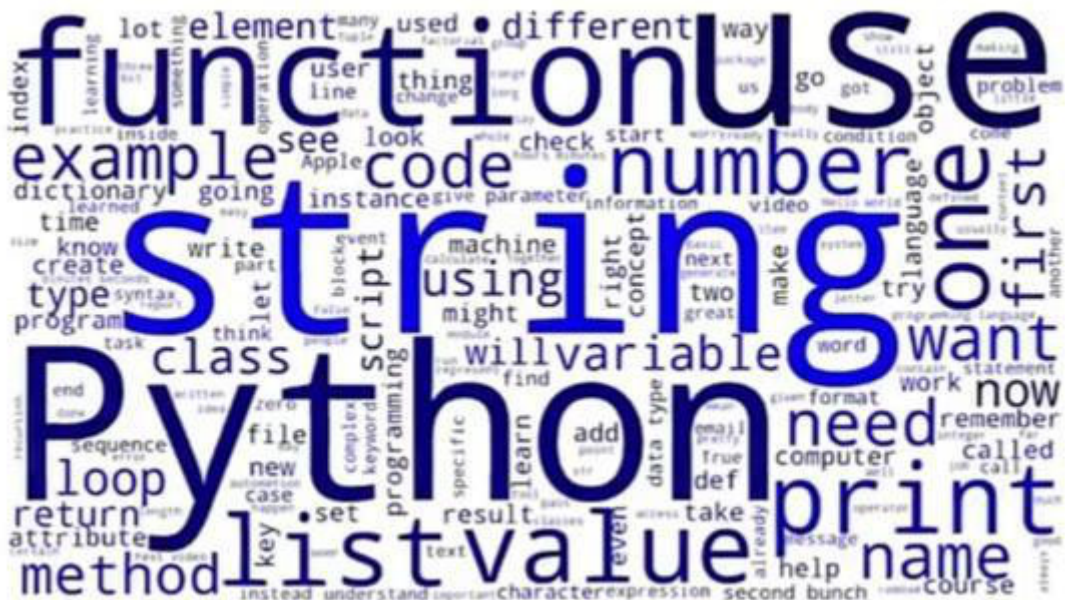


Fig. 23.1.1: A word cloud

Usually the sizes of the words are determined by how many times each word appears in a specific text. To create the image itself, we're going to use an external Python module called **creatively Word**

Your project is to create a script that would go through the text and count how many times each word appears. We've done this a few times already. Do you have any ideas how we should tackle this one?

If you're thinking of using a dictionary to count how many times each word appears, then you have a good answer! You're going to prepare a dictionary and use that as a parameter for the word cloud module. Not too tricky, right? I think you can handle a little more, so two things you have to watch out for.

First: **punctuation** Before counting the frequency of the words, you need to make sure that there are no punctuation marks in the text. If you don't, a string example with a comma at the end would be different from a string example with a dot at the end.

So before you put words into the dictionary, you have to clean up the text to remove any punctuation marks.

The second thing is we want to keep our word cloud Certain words in our language crop up a lot and if we include all of these we're going to get a pretty dull word cloud. Think about words like "a", "the", "too", "if" and so on. They usually appear a whole lot in text but aren't too relevant to the text's overall message.

We want our Cloud to show words that are relevant to the text we're using for the input. So you need to find a way to exclude irrelevant or uninteresting words when processing the text. For the input, you're going to upload a text file. You can choose any text file you like for your input.

It could be the contents of a website, a full novel or even everything that one author has ever written. You just need to make sure that it's **one text** so that it can be processed by the code. Okay, before jumping into the project, remember you can revise this section or re-watch the video if something is not clear.

I know I'm starting to sound like a broken record, but this time it's very important. This final project is the real test of how much

you've gotten your head around in this course. It can highlight areas you need to brush up on. So we want you to be super clear on what you need to do on that point.

You'll find an overview of what you have to do in the next section. Can you guess the best way of tackling this problem? Use our step-by-step approach that we outlined earlier. Understand the problem statement, research available options, plan your approach, write your code and finally execute.

Okay, feeling good? Ready to go? Remember, you know this stuff and you've totally got it!

23.2. Final Project Help

23.2.1 Project goal

Create a dictionary with words and word frequencies that can be passed to the `generate_from_frequencies` function of the `WordCloud` class. Once you have the dictionary, use this code to generate the word cloud image:

```
cloud = wordcloud.WordCloud()  
cloud.generate_from_frequencies(frequencies)  
cloud.to_file("myfile.jpg")
```

Things to remember

Before processing any text, you need to remove all the punctuation marks. To do this, you can go through each line of text, character-by-character, using the `isalpha()` method. This will check whether or not the character is a letter.

To split a line of text into words, you can use the `split()` method.

Before storing words in the frequency dictionary, check if they're part of the "uninteresting" set of words (for example: "a", "the", "to", "if"). Make this set a parameter to your function so that you can change it if necessary.

Input file

For the input file, you need to provide a file that contains text only. For the text itself, you can copy and paste the contents of a website you like. Or you can use a site like [Project Gutenberg](#) to find books that are available online. You could see what word clouds you can get from famous books, like a Shakespeare play or a novel by Jane Austen.

Jupyter Notebooks Help

Remember that if you need help with Jupyter Notebooks, you can check out [this help](#)

23.3. Final Project (Challenge Lab 5)

23.3.1. Instructions

Again, for this project, you'll create a "word cloud" from a text by writing a script. This script needs to process the text, remove punctuation, count the frequencies, and ignore uninteresting or irrelevant words. Good luck!

23.4. Final Project Grading

I (or any member of my team) is available to help you grade your final project. You can use my help link (email) at the end of chapter 25 to send your project work for grading. We will get back to you in 12 to 24 hours with your result.

However, if you cannot wait, you can open the “Challenge Labs” folder you downloaded earlier. It contains the the solution to this project. You can use it to grade your project by yourself. Just be honest as you grade. Here’s the file name to search:

Final Project Solution File Name: *C1M6L2_Final_Project_V3.ipynb*

The text used in the project is as follows:

Text Begins

I am a Control Systems engineer, Systems Integrator and a Content Creator. I have worked with over a thousand clients across business sectors, mostly the PLC automation industry. I have written numerous books, articles, and leadership classes for higher education institutions.

I have over 15 years of experience in Control Systems Engineering. I have had the opportunity to work within world class organizations such as Kraft Heinz, Procter & Gamble, and Post Holdings.

As a Control Systems Engineer, I have worked on several PLC-based systems such as the Allen-Bradley's RSLogix 5, 500, 5000, Studio 5000 and PACs. I have mastered other great technologies such as Cognex In-Sight Vision Systems & so much more.

Now I live and breathe PLCs (Programmable Logic Controllers). I've invested a lot of money and time into equipping myself with many of the latest PLC hardware in the world. This is because I truly believe that an investment in myself will pay dividends down the road and that the automation industry will only keep growing.

I believe in excellence and I'm highly driven by successful people. I am dedicated to seeing my clients succeed and achieve their goals. I love to create PLC programs and help manufacturing companies grow. I've successfully coached over a thousand business owners and leaders.

I'm proud to boast of extensive experience and a successful company which has been in business for over 15 years.

*****Text Ends*****

Feel free to use my own text if you like. Otherwise, you can copy and paste the contents of a website you like. Feel free to share your word cloud in the discussion forums!

24. Course Wrap up

24.1. Congratulations!

Congratulations! You've made it through the entire course. These were not easy concepts to learn. I want you to think all the way back to when you were just starting this journey. You remember what you were feeling when you watched those first videos. Maybe a little nervous, terrified, excited, or probably all these emotions at once.

You tuned in with me, studied the whole course, watched all my videos, and kept going when it got complex. You should be proud of yourself! Take a moment to reflect on where you are now. You've gone from having little or no knowledge of programming, to being able to write all kinds of complex functions.

You're using conditionals, loops, strings, lists, and even dictionaries. You even created your own objects. You put it all together to write your very own program, applying a process which you might use in your day-to-day IT role, and hopefully you had as much fun doing it as I did teaching you.

It's impressive that you've mastered all the stuff, and I hope it's just the beginning of your Python journey. Building a successful career in IT calls for perseverance, curiosity, and grit - three qualities you've proven to have heaps of by making it this far.

It also requires skills and knowledge, and this basic Python is definitely a powerful tool in your IT toolbox. Knowing how to write scripts will set you apart from others as you look to advance in

your career. I bet you're now seeing tasks all around you which are sparking ideas on how you could automate them with a script. The possibilities are endless, and it's just the start.

Remember what I said in one of my first videos: a journey of a thousand miles begins with a single step. There's still a lot of exciting things you can learn. I hope soon you will order my book for the next course (Part 2) where you will be learning all about **how Python interacts with the computer's operating**

But for now, I want to wish you best of luck. I look forward to seeing you and your code out there.

24.2. Discussion Forums: Share Your Learner Journey

Thinking back on the whole course: What's been the most interesting thing you learned? What's been the most difficult thing? Share your learner journey with the [learner](#)

24.3. Sneak Peek of the Next Course (Part 2)

You've made it through this Python course, which is part one. But your journey with Python is just heating up. In the next part, you will learn **how Python interacts with the operating** You'll build on all the skills you learned here and your programming is going to get a little more sophisticated.

Here is what you can expect from the next course. We're going to cover how to set up your own developer environment in Python, and in no time, you'll start feeling comfortable using codes to interact with the operating system (OS).

We'll also manipulate files and processes running on the OS, and dive into RegEx which is a super powerful tool for processing text files. You're even going to write a script that might be similar to a task you'd be assigned at your job.

But personally, my favorite part of the whole course is definitely where we talk about the Linux OS, but it's not because it's the primary OS I use in my job. Linux opens up a whole world of customization and configuration, and I find it really interesting.

I've got a lot of powerful and fun concepts coming up. So don't miss out. I'll see you over in the next course! Thank you for ordering my course. Bye for now!

25. How to Download the Course Resources

To download all the resources you need to study this course, use this short link:

If you experience any trouble downloading or accessing any of the resource, please contact me through my email address at the end of section 25.1.

25.1. How to Get Further Help

Dear reader, I hope you find this book and the accompanying videos very helpful in learning basic Python programming. You may find some of the concepts I covered in this book confusing at first. However, with time and a little more effort, you should be able to write very useful Python codes.

I am available for further help. Contact me through my support email below or visit my Author page if you need further help, or if you have questions or requests.

Regards,

mwillyd@gmail.com

A. J. Wright

(Author page:

25.2. More Helpful Resources

Below is a list of a few books from that I believe you will also find very helpful.

1. **PLC Programming from Beginner to Paid Professional - Part 1**
Learn RSLogix Software & Hardware with Demo Videos -

2. **PLC Programming from Beginner to Paid Professional - Part 2**
Learn How to Setup, Integrate & Program the Most Used Allen Bradley PowerFlex 525 Drive with Demo Videos -

3. **PLC Programming from Beginner to Paid Professional - Part 3**
Learn How to Develop & Embed Machine Vision System in PLC with Demo Videos -

4. **Simple PLC & HMI Programming** A Batching Tank Ladder Logic and HMI Tutorial for Learning PLCs -

- 5a. **Practical Electronic** A Strong Foundation for Creating Electronic Projects – **Print book** -

- 5b. **Practical Electronic** A Strong Foundation for Creating Electronic Projects – **Ebook** -

6. **Cloud Computing** Learn the Latest Cloud Technology and Architecture with Real-World Examples and Applications -

7. Artificial Intelligence The Foundations & History of Intelligent Machines -

8. Fundamentals of Satellite Navigation How to Design GPS/GNSS Receivers Book 1 - The Principles, Applications & Markets -

9. Excel Pivot Tables & A Step By Step Visual Guide -

10. How to Sell Used Books on Learn How to Generate Over \$10K Monthly Sourcing Cheap Used Books Online & Selling to Amazon without Marketing -

11. How to Study in USA on For Getting Financial Aids College Admissions & Visa Approval -

12. How to Study in Germany Tuition Get Free Education, Admission & Visa Approval. Get Prepared for Studies at German Universities -

13. Google Cloud Certified Associate Cloud Engineer Certification Guide 1: Learn with challenge labs, assessment tests and practice exams to build innovations and real-life experiences -