forzen academy

Academy of Python

From Complete Beginner to Python Expert

over 250 code examples

Nicholas Wilson

Academy of Python

by
Nicholas Wilson

Dedicated to **Jam**, the best executive personal assistant that ever was.
Without her cheering on all the crazy things I do, I’d be nothing.

Also, a very special thanks to my students for letting this crazy engineer guide them into the coding world.

Table of Contents

Preface

How do I use this book?

Most everything in the book is phrased in the form of a question so that you can easily search for things you need help with using the Table of Contents. Python is an easy to learn language, so keeping it casual as if you're just asking questions to a teacher can help it from feeling intimidating.

Will I become a Python master after reading this?

Your mileage will certainly vary! Programming is definitely a skill where you will get out of it what you put into it. I've been doing it for 25 years at this point and have certainly had my ups and downs. It's an enormous field with tons of things to learn. Even outside of languages, there's web technology, image technology, AI, app development… The list goes on and on!

The best thing you can do is just to never give up. Continual practice and experience will always be the best way to improve and succeed. Put in the time, keep chugging along, and eventually you'll realize you gained a ton of skill along the way. A black belt is just a white belt who never gave up.

How to read code segments

In the book there will be lots of included code segments. They will have a dark background and look something like this:

```
variable = "I'm a line of code!"
```

```
# I'm a comment

> I'm code output
```

Lines of code will be colored according to the different parts of the coding structure, just how most typical coding helper programs color code. Comments about the code, or lines that don’t affect how the code runs but can be used to document the code or describe what is happening, will start with a **#**. The big bolded text with the **>** in front will be the **code output**, or what your program should print out on the screen when you run it. **You shouldn’t include this in your code file if you are typing the examples into your computer to run them!**

Chapter 1
Introduction

What is the Python programming language?

Python is a general-purpose, high-level programming language that is used widely for a variety of things like website development, data analysis, scientific computing, and script writing. It is known for being simple, flexible, and for having powerful data-handling capabilities.

Python is an interpreted language, which means that it is executed directly by an **interpreter** (a program that translates the instructions), rather than being compiled into machine code. This makes it very easy to write and test code, and it also allows Python programs to run on any computer operating system that has a Python interpreter installed.

Python is named after the British comedy group Monty Python, and its design philosophy is about emphasizing code simplicity and readability. The **syntax** (the way the language is written) of Python is designed to be easy to read and understand, making it a great language for beginners to learn and get into programming with. It also has a giant and very active community of developers and users who contribute to the development of the language and its ecosystem of **frameworks** and **libraries**.

Is Python easy to learn?

Yes, Python is generally considered as an easy programming language to learn, especially for beginners and others new to coding. Its simple and readable code syntax makes it a good language for people who are new to programming, and its focus on simplicity and readability makes it a good choice for learning programming fundamentals.

As said above, Python has a huge community of developers writing and contributing code to the Python ecosystem, meaning that there are many resources available for learning Python like tutorials, books, online courses, and forums where you can ask questions and get help from other Python coders.

Overall, Python's simplicity and ease of code maintainability/readability make it a great language for beginners to learn, and its flexibility and power make it a good choice for a wide variety of purposes.

What's the difference between the Python programming language and a living Python?

Python is a programming language, which means that it is a set of instructions and language structures that a computer can execute to perform various hardware operations. It is called "Python" because it was made by a fan of the British comedy group Monty Python.

A living python, on the other hand, is a type of snake that is found in various parts of the world, including Africa, Asia, and Australia. It is called a python because it is a member of the Pythonidae family of snakes, which includes many species commonly known as pythons.

The two are completely different things. A living python is a real-world animal and Python is a computer programming language. There is no direct relationship between the two, other than the fact that they happen to share the same name! So if you are afraid of snakes, Python can still be your favorite coding language!

Is Python a popular programming language?

Python is a very popular programming language. It is widely used in many different fields like web development, data science, scientific computing, artificial intelligence, server backend development, visual applications, and much more.

According to the TIOBE Index, which ranks the popularity of programming languages, Python is the fourth most popular language in the world (at the time of writing)! It is also one of the fastest-growing languages, with a popularity rating that has increased significantly over the last few years.

Python is popular for many reasons, including its simplicity, powerful standard library, and its large and active community. Many programmers also find it easy to learn and use, and it has a wide range of applications in different areas.

Python is a very popular programming language, and it is used by many programmers and organizations around the world. Its popularity is expected to continue to grow in the coming years so learning it is always going to be a safe bet.

Do I need to be smart to learn Python?

No, you do not need to be smart to learn Python! While it is true that programming can be really challenging sometimes, and that some people may find it more difficult than others, Python is generally considered to be a language that is easy to learn, even for beginners.

Python's simplicity makes it easier to understand and work with than many other coding languages. Additionally, there are many resources available for learning Python so you’ll always have help along the way as you code.

Even if learning Python may not be the easiest thing in the world, you can do it if you put in the time and effort! It is not necessary to be smart or have any prior programming experience to learn Python.

Can I make mobile apps with Python?

Yep! You can make mobile apps with Python. While Python is not typically used for building native mobile apps that run directly on mobile platforms such as **iOS** and **Android**, it can be used to build cross-platform mobile apps that can be run on multiple mobile platforms, using awesome tools and frameworks such as BeeWare, Kivy, PyQt, and many more.

Kivy is a popular and powerful **open-source** library and framework for building mobile apps with Python that can run on multiple platforms. It provides a rich set of components and features, such as graphics, audio, and touch that let you create and develop interactive and user-friendly mobile apps. Using **Buildozer** and the **Kivy Launcher** app you can package up your Kivy app for lots of different platforms.

PyQt is another popular library for building cross-platform mobile apps with Python. PyQt is a bridge to using the **Qt** library, a powerful and comprehensive framework for building desktop and mobile apps with **C++**. PyQt allows you to leverage the capabilities of Qt from Python, and to develop cross-platform mobile apps with Python that have a native look and feel, and that can run on platforms like **iOS**, **Android**, and **Windows**.

BeeWare is a collection of tools and libraries for building mobile apps with Python. BeeWare provides tools for building, testing, and debugging Python apps for various platforms, such as **iOS**, **Android**, and **web browsers**, as well as libraries for implementing common mobile app features, such as graphics, audio, and networking, using Python. BeeWare also provides support for building and deploying mobile apps for various platforms, using tools such as the **Toga** library or the **Cordova** framework.

Can I make websites with Python?

Yes, you can make websites with Python! Python is a versatile and powerful coding language that can be great for building various types of apps, including websites. Python provides a rich set of libraries and frameworks that make it easy to make and develop web apps in Python, such as Flask, Django, and Pyramid.

Flask is a popular, lightweight web framework for Python that allows you to make **web applications** in Python quickly and easily. Flask provides a simple and flexible API that allows you to define and implement the components of a web application, such as templates, views, and routes, then connect them all together in a modular and extensible way. Flask also provides support for other things such as databases, authentication, sessions, etc, that are commonly used in web apps.

Django is another popular framework for Python that provides a **full-stack**, “batteries-included” approach to web dev with Python. Django provides a ton of integrated components and features that allow you to create and develop **web applications** with Python end-to-end, all the way from the user interface to the database. Django also provides a feature-full and secure architecture, as well as a robust and extensible set of tools and libraries that make it easy to build complex and scalable web apps, all in Python!

Pyramid is a web framework for Python that provides a modular and flexible approach to web development. Pyramid allows you to choose and combine the various components and features that you need to create and develop a **web app** using Python, like routing, templating, and authentication, and integrate them together in a scalable and maintainable way. Pyramid also provides support for various databases, web servers, and deployment scenarios that allow you to customize and optimize your web app for specific requirements and constraints.

Using Python to make websites is a popular and effective option that allows you to leverage the strengths of Python and create high quality web applications that can meet the various needs or requirements of your project.

Can I make desktop programs in Python?

Yes, you can make desktop programs in Python. Python is a versatile programming language that can be used for a wide range of purposes, including the development of **desktop applications**. There are several ways to create desktop programs in Python, depending on your specific needs and goals. One option is to use a framework such as **PyQt** or **Tkinter** which provides a set of tools that make it easy to create graphical user interfaces (**GUIs**) for your Python programs.

These frameworks allow you to create rich and interactive **desktop applications** using Python, and can be used to develop programs for a wide range of purposes, from simple utilities to complex and dynamic apps. Alternatively, you can use Python to create command-line interface programs (**CLI**), which are applications that are run from the terminal or command prompt. These do not have a graphical user interface, but can still be useful for a wide range of tasks, from automating simple things to creating powerful tools for developers.

Where can I find official documentation for Python?

The official documentation for Python is available on the Python website https://docs.python.org/3/. The documentation includes tutorials, guides, and API reference materials for all the versions of Python. It is a great resource for learning more about the tons of features in Python and how to use them.

To access the documentation just go to the Python website and click on the "Documentation" link in the main navigation menu. This will take you to the documentation homepage where you can look through the various sections of the docs and search for specific topics.

The documentation on the site is well organized and easy to navigate! It includes a table of contents on the left side of the page that allows you to quickly jump to different sections. It also includes a search bar at the top of the page that you can use to find whatever it is you’re looking for.

The official Python documentation is a valuable resource for anyone learning or writing code in Python. It is regularly updated to reflect the latest changes and improvements to the language and is an essential reference for anyone in the Python community.

What is the difference between Python 2 and 3?

Python 2 and **Python 3** are two major versions of the Python programming language. They are similar in many ways, but there are some important differences that you should be aware of if you are learning Python or working with Python code.

One of the main differences between version 2 and 3 is that Python 3 is the future of the language, and Python 2 is no longer actively developed. This means that new features and improvements are only being added to Python 3, and Python 2 will eventually reach the end of its life and be phased out.

Another important difference is that Python 3 is not backwards-compatible with Python 2. This means that code that works in Python 2 may not work in Python 3, and vice versa. So, if you are learning Python, I would recommend starting with Python 3, and avoiding Python 2 if you can.

Some other differences between Python 2 and Python 3 include changes to the syntax and standard library, improved support for Unicode, changes to the way that numbers are handled, and much much more. For a more detailed comparison of the two versions please refer to the official Python documentation.

Chapter Review

- What kind of things can you make in Python?
- Where is the official Python documentation if you get stuck?
- Can anyone learn Python?
- Is Python easy for beginners?
- Are you excited to learn Python?

Chapter 2
Getting Set-Up

Installation

Can I program Python on a smartphone?

Yes, you can program Python on a **smartphone**. While smartphones are not typically known for their powerful computing capabilities, many modern smartphones are more than capable of running simple Python programs. There are several different ways to program Python on a smartphone. One option is to use a third-party Python app, such as **Pythonista** or **PyMobile**, which provides a Python-friendly development environment and lets you write, run, and test your Python code on your phone. Another option is to use a remote development setup, where you use your **smartphone** as a **terminal** to connect to a remote computer that runs the actual Python code. This allows you to write and manage your code on a more powerful computer, while still being able to access and run your programs on your smartphone.

How do I install Python on Windows?

To install Python on a **Windows** computer, you can follow these steps:

1. Open your browser and go to the official Python website (https://www.python.org/).

2. Click on the "Downloads" button in the top menu to go to the download page.
3. Scroll down to the "Python Releases for Windows" section and click on the link for the latest Python version. This will download the Python installer to your computer.
4. Once the download is complete, open the installer and follow the instructions to install Python.
5. After the installation is complete, open the Start menu and search for "Python". You should hopefully see a Python 3.x entry in the list of results. Click this to open the Python interpreter, which you can use to run Python code.

To verify that the installation was successful, you can run the following command:

```
python --version

or

python3 –version
```

This should print the version number of Python that was installed on your computer.

How to install Python on a Mac?

To install Python on a **Mac**, you can follow these steps:

1. Open a browser and go to the official Python site (https://www.python.org/).
2. Click on the "Downloads" button in the top menu to go to the download page.
3. Scroll down to the "Python Releases for Mac OS X" section, and click on the link for the latest Python version. This will download the Python installer.

4. Once the download is finished, open the installer and follow the on-screen instructions to install Python on your Mac.
5. After the installation is complete, open the **Terminal** app and run the following command to verify that Python was installed correctly:

```
python --version

or

python3 --version
```

This should print the version number of Python that was installed on your **Mac**.

Alternatively, you can also use a package manager like **Homebrew** to install Python on your **Mac**. To do this, follow these steps:

1. Open a **Terminal** window and run the following command to install Homebrew (we are using **Ruby** here to run this as it's generally preinstalled on most Mac):

```
/usr/bin/ruby -e "$(curl -fsSL
https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

2. After Homebrew is installed, run the following command to install Python:

```
brew install python
```

This will install the latest Python version on your Mac. To verify that the installation was successful, you can run the following command:

```
python --version
```

```
or

python3 --version
```

This should print the version number of Python that was installed on your **Mac**.

How to install Python on Linux?

To install Python on a **Linux** computer, you can follow these steps:

1. Open a **terminal** and run the following command to check if Python is already installed on your Linux system:

```
python --version

or

python3 --version
```

If Python is already installed, this command will print the version number of Python that is installed on your system. If Python is not installed, you will see an error message instead.

2. If Python is not installed, you can install it using the package manager that comes with your Linux distribution. For example, on **Ubuntu** and other **Debian**-based systems, you can use the **apt** command to install Python like this:

```
sudo apt update

sudo apt install python3
```

On **Fedora** and other **Red Hat**-based systems, you can use the **dnf** command to install Python like this:

```
sudo dnf update
sudo dnf install python3
```

On other **Linux** distributions, you can use the appropriate package manager command to install Python.

After Python in installed you can verify it’s working correctly with this command:

```
python --version

or

python3 --version
```

This should print the installed version of Python or an error message if not installed properly.

Python is a popular programming language that is widely used on **Linux**, and most Linux distributions come with a version of Python pre-installed. You can also install the latest version of Python using the package manager of your Linux distribution. Once Python is installed, you can use it to write and run Python code!

What is a programming IDE?

A programming **IDE**, or **Integrated Development Environment**, is a software program that provides a range of tools and features designed to help software developers write and manage their code more efficiently. They typically include a code editor, a debugger, and a compiler or interpreter, and may also include features such as code completion, code refactoring, and version control integration. The goal of an IDE is to provide a single, integrated workspace where developers can access all the tools and resources they need to efficiently develop and maintain their code.

Do I need an IDE to be able to program?

No, you do not need an **IDE** to be able to program. While an IDE can provide a bunch of nice tools that can make the process of writing code more efficient, it is not essential for learning or practicing programming. Many programmers choose to use a simple text editor, such as **Notepad**, **Sublime Text**, **nano**, or **vim** to write and edit their code, and then use the **terminal** to compile and run their programs. In fact, learning to program without an IDE can be a valuable learning experience as it can instill concepts without you becoming over-reliant on the IDE to do many things for you.

Are there any good Python IDEs?

Yes, there are many good Python **IDE**s available! Some of the most popular Python IDEs include:

- **PyCharm**: PyCharm is an awesome Python IDE developed by JetBrains. It includes features like a code editor, a debugger, and support for version control systems. **PyCharm** is available in a paid professional edition and a free community edition.
- **Visual Studio Code**: **VSC** is a popular open-source code editor developed by Microsoft. It includes support for Python and many other languages. Visual Studio Code is highly customizable with lots of available extensions, and it is available for free.
- **Spyder**: Spyder is an open-source Python IDE that was specifically designed for scientific and data science development. It includes great features like an interactive console, a code editor, and a debugger, and is available for free.
- **IDLE**: IDLE is the default Python IDE that is included with the Python language. It is a simple and lightweight IDE that includes a code editor, a terminal for running Python code, and a debugger. IDLE is available for free.

These are just a few examples of Python **IDE**s, and there are many other good options available. The best Python IDE for you will depend on your specific needs and preferences. Some IDEs are more suited to professional development, while others are more focused on scientific or data science development. Some IDEs are open-source and free, while others are commercial products. You can try out different IDEs and see which one works best for you.

Basic Execution

What are good Python beginner projects?
There are many good Python beginner projects that you can work on to help you learn the language and improve your skills. Some ideas for beginner Python projects include:

1. Building a simple game, such as Tic-Tac-Toe or Blackjack. These can help you practice using loops and conditional statements.
2. Writing a program that simulates a simple virtual chatbot, which can respond to user input based on what you type to it.
3. Creating a program that generates strong random passwords, which can help you learn about strings and the use of random numbers in Python.
4. Building a web scraper, which can extract data from websites and save it to a file, to learn about web network calls and Strings and file input/output in Python.
5. Writing a program that performs basic image manipulation, such as resizing or cropping images, to learn about working with media in Python.

As you work through this book, try to think about how you may implement the project ideas above!

In Python, how do I print out a message?

To print out a message in Python, you can use the **print** function. The print function takes one or more **arguments**, and it prints them to the screen (technically the **standard output** but don’t worry about that for now), separated by spaces and followed by a newline character.

For example, to print a simple message like "Hello there!", you could do this:

```
print("Hello there!")
```

In this code, the **print** function is called with the string "Hello there!" as the **argument**. This code would print "Hello there!" on the screen (your terminal), followed by a line break, also known as a **newline** character.

You can also pass multiple values to the **print** function, and they will be printed to the screen separated by spaces. For example, you could do this:

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

In this code, the **print** function is called with two values or **arguments**: the string "Hello," and the string "there!". This code would print "Hello there!" to the screen, with a space between "Hello" and "there!", followed by a line break.

You can also use the **print** function to print the value of a variable by passing the variable as an argument to the **print** function. For example, you could do this:

```
message = "Hey you!"

print(message)
```

In this code, the **message** variable is defined and assigned the string "Hey you!", and then the **print** function is called with the **message** variable as the argument. This code would print "Hey you!" to the screen, followed by a line break.

Overall, the **print** function is used to print values to the screen in Python. The **print** function takes one or more values as **arguments**, and it prints them to the screen separated by spaces and followed by a **newline** character. You can use the **print** function to print simple strings, the values of variables, or the results of other code statements known as **expressions**.

How do I get input from the user in Python?

To get input from the user in Python, you can use the **input()** function. This function will print a message to the user and wait for the user to enter some text back before hitting the enter key. Here's an example:

```
name = input("What’s your name: ")

print("Hello " + name)
```

In this example, the **input()** function will print the prompt "What’s your name: " and then wait for the user to enter their name. When the user enters their name and presses enter, the **input()** function will return the user's input as a string. This string is then assigned to the name variable, and it is used in the **print()** statement to say hello to the user.

Here's how this example would work if you ran it:

```
Enter your name: Kenobi

Hello Kenobi
```

Remember that the **input()** function always returns the user's input as a string, even if the user entered a number. If you want to use the user's input as a number, like for math calculations, you will need to convert it to a number using one of Python's conversion functions, such as **int()** or **float()**. For example:

```
num = int(input("Enter a number: "))

print(f"{num} squared is {num * num}")
```

In this example, the **input()** function is used to get a number from the user, and then the **int()** function is used to convert the user's input from a string to an integer. This integer is then assigned to the num variable and used to calculate the number squared.

Also in this example, the curly brackets and letter **f** are used to format the output. Between the curly brackets you can put variables or coding statements and they’ll be executed as the message is printed to the screen.

Can you explain Python indentation?

In Python, indentation is used to indicate the coding block structure. It is a fundamental aspect of the Python language, and it is used to define the scope and hierarchy of code blocks.

In Python, indentation is done using whitespace and is typically done using four spaces for each level. Here is an example of a Python code block that uses indentation:

```
def say_hello(name):

    if name:
        print("Hello " + name)

    else:

        print("Hello there!")
```

In this example, the **say_hello()** function has two code blocks: an **if** statement and an **else** statement. Each of these code blocks is indented using four spaces, which indicates that they are both part of the **say_hello()** function. The **print()** statement inside the **if** block is also indented, which indicates that it is part of the **if** block.

Indentation is used to show the relationship between different code blocks in Python, and it is important to use it accurately. If your code is not properly indented, it will not run and you will get syntax errors.

Ultimately, indentation is a key part of the Python language, and it is essential for writing structured and readable code. Remember that it is required as part of Python syntax and must be used for code blocks.

Should I use tabs or spaces in Python?

PEP 8, the official style guide for Python, recommends using 4 spaces for indentation in Python programs. It also recommends using spaces, not tabs, for indentation.

Using 4 spaces for indentation is a common convention in Python, and it is used by many programmers and organizations. It allows for consistent indentation across different Python files, and it helps to improve how readable the code is.

Using spaces instead of tabs for indentation is also recommended by **PEP 8**, because tabs can be interpreted differently depending on the editor being used.

So just keep in mind that 4 spaces is the convention that is most commonly used in the Python community, and it is a good starting point for writing clean and readable Python code. When you use an **IDE** there is a good chance pressing tab will insert 4 spaces instead of a tab character, making it easy to get accustomed to the tradition even if you are used to pressing tab already.

Can I write Python with brackets instead?

No, you cannot use brackets instead of indentation in Python. Indentation is a fundamental aspect of the language, and it is used to define the scope and hierarchy of code blocks.

Python uses indentation to show the relationship between different code blocks, and it is not possible to use brackets instead of indentation. If you try to use brackets to define code blocks in Python, you will get a syntax error.

How do I run my Python program?

Once you have Python installed, you can run a Python program by using the **python** command in a command-line interface, such as the **terminal** on **macOS** or **Linux**, or the command prompt on **Windows**. For example, if you have a Python program saved in a file called my_program.py, you can run it by using the following command:

```
python my_program.py

or

python3 my_program.py
```

This will start the Python interpreter, which will read and execute the instructions in your program. If your program has any output, such as printed text or plotted figures, it will be displayed in the command-line interface.

In several of the examples up until now I have noted that you can use the **python** or **python3** command. Some installation methods will install Python in such a way that one or the other will be the command to use on your computer. Just keep in mind, as long as you have Python 3.x installed, they will be identical. Find the command that works for you after your installation and continue to use that command.

What is a command line interface?

A **command-line interface** (**CLI**) is a type of user interface that allows users to interact with a computer by entering commands using a keyboard. This is also commonly called a **terminal**. In contrast to a **graphical user interface** (**GUI**), which uses visual elements such as windows, buttons, and menus to provide a more user friendly way to interact with a program, a **CLI** uses text-based commands that are entered into a command-line prompt.

CLIs are often used in situations where a **GUI** is not practical or necessary, such as when working with a remote computer server over the internet or when automating tasks using scripting programs. They are also often used by developers when the flexibility and power of a **CLI** makes more sense.

In most **CLIs**, the user enters commands at a prompt. This prompt commonly ends in a symbol such as a dollar sign ($) or a percent sign (%) on **Mac** or **Linux** systems. The user can then type the command, followed by any necessary options, and then press the enter key to execute the command. The computer will then read the command options and perform the appropriate actions, printing any output back out into the **terminal** used.

How do I execute system commands in Python?

To execute system commands in Python, you can use the **subprocess** module. This module provides a number of functions that allow you to execute other programs from Python and then read their output, and much more.

Here's an example of how you can use the **subprocess** module to run a system command and capture its output:

```
import subprocess


command = "ls"

output = subprocess.check_output(command, shell=True)


print(output)
```

In **Linux** (or **Unix** systems like **Mac**), the **ls** command will print out all the files in the current directory.

In this example, the **subprocess.check_output()** function is used to run the **ls** command. The output of the **ls** command is captured and assigned to a variable, and then it is printed to the console. Here's how the output might look:

```
> file_a.txt

> file_b.txt

> file_c.txt
```

There are many other functions in the subprocess module that you can use to run system commands and interact with their input, output, or error streams. You can learn more about these in the official Python docs.

In Python, what is "*args"?

In Python, the ***args** syntax allows you to pass some arguments to a function. It is used to define a function that can take an arbitrary number of arguments, and it is often used in combination with the * operator to "unpack" a list of arguments when calling a function.

Here is an example of how to use ***args** to define a function that takes an arbitrary number of arguments:

```
def sum_numbers(*args):
    sum = 0
    for arg in args:
        sum += arg
    return sum


# Call the function with a bunch of arguments
result = sum_numbers(1, 2, 3, 4, 5)
print(result)
```

```
> 15
```

```
# Call the function with a single argument
result = sum_numbers(1)
print(result)
```

```
> 1
```

In this example, the **sum_numbers()** function has a single parameter, ***args**, which is used to collect all of the arguments passed to the function. The ***args** parameter is treated as a **tuple**, which is a sequence of values that cannot be changed.

When we call the **sum_numbers()** function, we can pass as many arguments as we want between the parenthesis. The ***args** parameter collects all of the arguments, and they are looped over in the **for** loop. The **sum** variable is initialized to 0, and then each of the arguments is added to it, and the final result is returned to be printed out.

This example shows how ***args** can be used to define a function that can take an arbitrary number of arguments. You can use this syntax to create functions that are flexible and can be used in a variety of situations.

In Python, what is "**kwargs"?

In Python, the ****kwargs** syntax allows you to pass a variable number of keyword arguments to a function. It is used to define a function that can take an arbitrary number of keyword arguments, and it is often used in combination with the ****** operator to **unpack** these into a **dictionary**.

Here is an example of how to use ****kwargs** to define a function that takes an arbitrary number of keyword arguments:

```
def print_values(**kwargs):

    for key, value in kwargs.items():

        print(key + ": " + value)
```

```
# Call the function with multiple keyword arguments

print_values(name="Luke", age=30, planet="Endor")


> name: Luke

> age: 30

> planet: Endor


# Call the function with a single keyword argument print_values(name="Han")


> name: Han
```

In this example, the **print_values()** function has a single parameter, ****kwargs**, which is used to collect all of the keyword arguments passed to the function. The ****kwargs** parameter is treated as a dictionary, which is a collection of key-value pairs.

When we call the **print_values()** function, we can pass as many keyword arguments as we want. The ****kwargs** parameter collects all of the keyword arguments, and they are looped over in the **for** loop. The **key** and **value** variables are used to access each key-value pair in the dictionary, and the values are printed to the screen.

This example shows how ****kwargs** can be used to define a function that can take an arbitrary number of keyword arguments. You can use this syntax to create functions that are flexible and can be used in a variety of situations.

Is Python as fast as other languages?

The performance of Python, compared to other programming languages, can vary depending on various factors, such as the specific implementation of Python and the code being executed. In general, Python is not considered to be as fast as some other compiled languages like **C** or **C++**, which are designed for high-performance and low-level programming tasks.

However, Python provides several tools and techniques that can be used to improve the performance of Python programs. One way is using a **just-in-time** (**JIT**) compiler, such as **Numba** or **PyPy**, which compiles Python code into **machine code** at runtime. You could also use a static **compiler**, such as **Cython** or **Nuitka**, which compiles Python code into **machine code** *before* runtime.

Additionally, Python provides a rich set of libraries and modules, such as **NumPy** and **Pandas**, that are optimized for numerical and scientific computing, and that can provide high-performance and efficient implementations of common algorithms and operations. With these libraries optimized for speed you can be sure they'll execute fast.

While Python may not be as fast as some other languages in some cases, it is well-suited for a wide range of applications, including data analysis, machine learning, and web development, where performance might not be a large concern. Additionally, there are ways to increase your Python execution speed so there will always be a way to get more speed if you need it.

What is a Python compiler?

A Python **compiler** is a program that translates Python source code written in the Python programming language into a lower form like **machine code**. Python is an interpreted language, which means that the Python interpreter, rather than a compiler, is responsible for executing Python code. Instructions are read in from your text file as text and then executed. However, Python has several **JIT** and other **compilers** made for it as discussed in the previous section.

Overall, using a Python **compiler** may provide various benefits, such as improved performance and portability.

Chapter Review

- What operating system do you have?
- What is a compiler?
- What is an interpreter?
- How do you run your Python program?
- How is indentation used in Python?
- Is indentation structure required in Python?
- How do you print out to the screen?
- Should you use tabs or spaces when writing code in Python?
- What is a terminal?
- Is a terminal the same thing as a command prompt?
- What is a CLI program?
- What is a GUI program?
- What is a system command?

Chapter 3
Variables and “If” Statements

Variables

In programming, what is a variable?

In programming, a **variable** is a named location in a computer's memory that is used to store data. A variable has a name, a type, and a value, and it is used to represent a value that can change or vary during the execution of a program.

In most programming languages, a variable is defined by specifying its name and type, and then assigning a value to it. For example, in Python, you could define a variable like this (note that an **integer** is just another word for a whole number):

```
x = 7

# x is an integer variable with the value 7
```

In this code, the **x** variable is defined as an integer with the value 7. The **x** variable has a name, a type, and a value, and you can use it in other places in your code for various operations.

For example, after you make your **x** variable, you could use it in an expression like this:

```
y = x + 13

# y is an integer variable with the value 20
```

In this code, the **x** variable is used in an expression to add 13 to its value, and the result is stored in the **y** variable. This allows the **x** variable to be used multiple times within a program, making it easier to write and maintain complex code.

Just remember that a variable is a named location in a computer's memory that is used to store data. Variables are an essential part of many programming languages.

What is a variable in Python?

Remember that in Python, you can create a variable named **z** and assign it the value **42**, like this:

```
z = 42
```

Once you have created a variable, you can access its value by using the variable's name. In this example, you could print the value of **z** like this:

```
print(z)
```

This would output the value **42** to the screen.

In Python, you can use variables to store all kinds of data like numbers, **strings** (a *string* of characters, or just text), and more complex data types such as **lists** and **dictionaries**. You can also use variables to store the results of computations and other expressions. This makes it easier to write programs that can store and manipulate data in powerful and flexible ways.

What is the difference between a mutable and immutable data type?

The main difference between a **mutable** and **immutable** data type is that a mutable data type can be modified after it is created, while an immutable data type cannot be modified. In Python, the mutable data types include **lists**, **sets**, and **dictionaries**, while the immutable data types include **strings**, numbers, and **tuples**. Mutable data types are useful when you need to change the values of an object after it is created. For example, you can add, remove, or update the items in a list, set, or dictionary, and the object will be updated to reflect these changes. Immutable data types, on the other hand, cannot be modified after they are created. This means that any operation that attempts to change the value of an immutable object will instead create a new object with the modified value. Because of this, immutable data types are often considered to be safer and more predictable than mutable data types, as they cannot be modified unexpectedly.

In programming, what is a constant variable?

In programming, a **constant variable** is a variable that has a fixed value that cannot be changed or reassigned once it has been defined. In other words, a constant is a value that remains the same throughout the execution of a program.

Constant variables are often used in programming to represent values that are fixed and known in advance, such as physics constants, mathematical constants, or other values that do not change during the execution of a program. For example, the value of the mathematical constant pi in math is a constant that is defined as 3.141592654…. This value does not change during the execution of a program, so it can be defined as a constant variable. We can simplify our representation to 3.14159 for these examples.

In many programming languages, there is built-in syntax for defining constant variables. For example, in C++, you can use the **const** keyword to define a constant variable, like this:

```
const double PI = 3.14159;
```

In this code, the **PI** variable is defined as a constant with the value 3.14159. The **const** keyword indicates that the variable is a constant, and it cannot be changed or reassigned once it has been defined.

Remember that a **constant variable** is a variable that has a fixed value that cannot be changed or reassigned once it has been defined. Constant variables are often used to represent values that are known in advance and do not change during the execution of a program.

How do I make a constant variable in Python?

A **constant** is a variable that cannot be changed or reassigned once it has been defined. In other words, a constant has a fixed value that cannot be changed.

In Python, there is no built-in syntax for defining a constant variable. However, you can achieve the same effect by using a variable name that is in all uppercase letters. For example, consider the following code:

```
MY_COOL_CONSTANT_VARIABLE = 15
```

In this code, the **MY_COOL_CONSTANT_VARIABLE** variable is defined with the value 15. By convention, the use of all uppercase letters in the variable name indicates that the variable is a constant and should not be modified.

However, it is important to note that this convention is not enforced by the Python interpreter. In other words, even though the

MY_COOL_CONSTANT_VARIABLE variable is defined with all uppercase letters, there is nothing stopping you from reassigning a different value to it, like this:

```
MY_COOL_CONSTANT_VARIABLE = 15

MY_COOL_CONSTANT_VARIABLE = 42
```

In this code, the **MY_COOL_CONSTANT_VARIABLE** variable is first defined with the value 15, but then it is reassigned to the value 42. This is allowed in Python, even though the variable is defined with all uppercase letters, because there is no built-in syntax for defining a **constant variable** in Python.

Therefore, while using all uppercase letters in a variable name is a convention that indicates that the variable should be treated as a constant, it is not a guarantee that the variable will not be changed or reassigned.

Remember that there is no built-in syntax for defining a constant variable in Python, but you can use the convention of using all uppercase letters in the variable name to indicate that the variable should be treated as a constant. Always keep in mind though that there's no guarantee this value hasn't been changed somewhere in the code.

Basic Data Types

In programming, what is a data type?

In programming, a **data type** is a classification of data that specifies the type and format of the data, and the operations that can be performed on the data. Different programming languages support different **data types**, and each data type has its own characteristics and behavior.

Data types are an essential part of many programming languages, as they allow the programmer to specify the type and format of the data that a program will work with. This helps to ensure that the data is stored and manipulated correctly, and it also allows the compiler or interpreter to check the correctness of the code and generate efficient machine code.

In most programming languages, data types are divided into two main categories: **primitive** data types and **composite** data types. Primitive data types are the basic data types that are built into the language, and they typically include types for storing numbers, characters, and **Boolean** values. Composite data types, on the other hand, are data types that are composed of multiple values or objects, and they include types such as **arrays**, **dictionaries**, and other complex **objects**.

Each data type has its own characteristics and behavior, and it is important for a programmer to choose the appropriate data type for the data that they are working with. For example, the **int** type is suitable for storing whole numbers, but it would not be suitable for storing numbers with fractions, in which case the **float** type would be a better choice.

Overall, data types are an essential part of many programming languages, and they allow the programmer to specify the type and format of the data that a program will work with.

What are data types in Python?

In Python, every value has a **data type**, which specifies the kind of value it is and determines what operations can be performed on it. Some of the most commonly used data types in Python include **integers**, **floats**, **strings**, **lists**, and **dictionaries**.

An **integer** is a whole number, and a **float** is a number with a decimal point. A **string** is a sequence of characters, and a **list** is an ordered collection of values. A **dictionary** is a collection of key-value pairs, where each key is mapped to a specific value.

Here are some examples of values with different data types in Python:

```
# Integer

a = 42


# Float

b = 3.1414159


# String

c = "Hello there!"


# List

nums = [1, 2, 3, 4, 5]


# Dictionary

jedi = { "name": "Mace Windu", "age": 42, "city": "Coruscant" }
```

The data type of a value in Python determines how that value can be used and what operations can be performed on it. For example, you can add two numbers together, but you can't add a number to a string. Understanding and working with data types is an important part of programming in Python.

What is an "int" in Python?

In Python, an **int** is a data type that represents an integer, or a whole number without a fractional part. This data type is used to store numeric values that do not have decimal places, such as quantities, ages, or sizes.

To create an **int** in Python, you can use the **int()** function, which takes a numeric value as an argument and returns the corresponding integer. For example:

```
# Create an integer from a float

int_value = int(3.14159)


# Create an integer from a string

int_value = int("42")
```

In the first example, it converts the decimal number **3.14159** to the integer **3** by removing the fractional part. In the third example, it converts the string "42" to the integer **42**.

What is a String in programming?

In programming, a **string** is a sequence of characters that represent text. A **string** is often used to store and manipulate text-based data, such as words, sentences, and paragraphs.

Strings are a common data type in many programming languages, and they are typically enclosed in quotation marks (single or double, depending on the language) to distinguish them from other types of data. Here is an example of a string:

```
"Hello there!"
```

In this example, the string is a sequence of characters that represent the text "Hello there!". The string is enclosed in double quotes (**"**) to indicate that it is a string.

Strings are an important data type in programming, and they are used to store and manipulate text-based data. They are a fundamental building block of many apps and are supported by most programming languages.

How do I make a String in Python?

To create a **string** in Python, you can use **string** literals, which are sequences of characters enclosed in quotation marks and assign it to a variable. Here is an example of how to create a string in Python:

```
# Create a string using double quotes

string_value_1 = "Hello there!"




# Create a string using single quotes

string_value_2 = 'General Kenobi...'
```

In this example, two strings are created using string literals. The first string is created using double quotes ("), and the second string is created using single quotes (').

In Python, you can use either single or double quotes to create a string, and you can use the same type of quotes to enclose the string as long as they are not used within the string. For example, if you want to create a **string** that includes a quote character, you can use the other type of quotation mark to enclose the string, as shown here:

```
# Create a string that includes a double quote

str_1 = 'The general said, "Hello there!"'
```

```
# Create a string that includes a single quote

str_2 = "The enemy replied, 'General Kenobi...'"
```

In this example, two strings are created that include quote characters. The first string uses single quotes to enclose the string, and it includes a double quote character within the string. The second string uses double quotes to enclose the string, and it includes a single quote character within the string.

You can create a string in Python by using string literals, which are sequences of characters enclosed in quotation marks. You can use either single or double quotes to create a string, and you can use the same type of quotes to enclose the string as long as they are not used within the string. Strings are an important data type in Python as they are in any language!

What are operators in Python?

In Python, **operators** are special symbols that represent computations like addition, multiplication, and division. These **operators** are used to perform operations on variables and values. For example, the + operator can be used to add two numbers together, like this:

```
king = 10

ace = 11

blackjack = x + y
```

After this code has been executed, the **blackjack** variable will hold the value 21.

There are various types of operators in Python, including:

- **Arithmetic operators**, which are used to perform mathematical operations on numbers, such as addition, subtraction, multiplication, and division.
- **Assignment operators** like “=” which are used to assign a value to a variable.
- **Identity operators**, which are used to compare the identity of two objects.
- **Comparison operators**, which are used to compare two values and return a **Boolean** value (True or False) depending on the outcome of the comparison.
- **Membership operators**, which are used to test whether a value is a member of a sequence (such as a list or a string).
- **Logical operators**, which are used to combine multiple conditions in a single expression.

Each type of operator has its own specific syntax and usage, which you can learn more about by reading the Python docs on the official site.

How do decimal numbers work in Python?

In Python, decimal numbers are represented using the **float** data type. This data type allows for the representation of fractional values, as well as very large and very small numbers.

To create a decimal number in Python, you can use the **float()** function, which takes a numeric value as an argument and returns the corresponding float number. For example:

```
# Create a decimal number from an integer

float_val = float(123)



# Create a decimal number from a string
```

```
float_val = float("3.14159")
```

In the first example, the **float()** function converts the integer **123** to the floating-point number **123.0**. In the second example, it converts the string "3.14159" to the floating-point number **3.14159**.

Once you have a decimal number, you can perform a bunch of operations with it, such as math, comparison, and type conversion. Some examples:

```
# Create a decimal number

float_val = 2.25


# Perform math operations on the decimal number

result = float_val + 3.10


# Compare the first number to another number

if float_val < 5.0:

    print("The number is less than 5.")


# Convert the decimal number to an integer

int_val = int(float_val)
```

In the first example, the code adds the decimal number **2.25** to the number **3.10** and assigns the result **5.35** to the **result** variable. In the second example, the code compares the decimal number to the number **5.0** and prints a message only if the decimal number is less than **5.0**. In the third example, the code converts the decimal number to an integer (**5**) and assigns the result to the **int_val** variable.

Note that decimal numbers are not always represented exactly in Python, due to the way they are stored in memory. This can lead to some imprecision in calculations involving decimal numbers. For example:

```
# Create a decimal number

float_val = 0.1


# Perform math operations on the decimal number

result = float_val + 0.2


# Print the result

print(result)
```

This code will print the number **0.30000000000000004**, instead of the expected result **0.3**. This is because the decimal number **0.1** cannot be represented exactly in binary floating-point format, and the imprecision accumulates when performing arithmetic operations on it.

To avoid this issue, you can use the **decimal** module, which provides the **Decimal** class for representing decimal numbers with a fixed number of digits of precision. This allows for more precise calculations with decimal numbers, but it comes at the cost of slower performance.

Here is an example of how to use the **Decimal** class to perform precise calculations with decimal numbers:

```
# Import the `decimal` module

import decimal
```

```
# Create a `Decimal` object
decimal_value = decimal.Decimal("0.1")


# Perform arithmetic operations on the `Decimal` object
result = decimal_value + decimal.Decimal("0.2")


# Print the result
print(result)
```

This code will print the number **0.3**, which is the expected result of adding the numbers.

What is a boolean in Python?

In Python, a **bool** is a data type that represents a boolean value, which is either **True** or **False**. This data type is used to represent binary choices, such as yes/no questions or an on/off switch.

To create a **bool** in Python, you can use the **bool()** function, which takes a value as an argument and returns **True** if the value is considered "**truthy**", or **False** if the value is considered "**falsy**". For example:

```
# Create a boolean from an integer
bool_val = bool(42)


# Create a boolean from a string
bool_val = bool("hello")
```

In the first example, it converts the number **42** to the boolean **True**, because non-zero integers are considered **truthy** in Python. In the third example, it converts the string "hello" to the boolean **True**, because non-empty strings are also considered truthy.

Once you have a **bool**, you can perform various operations with it, such as comparison, logical operations, and type conversion. For example:

```
# Create a boolean

bool_val = True


# Compare the boolean to another bool value

if bool_val == True:

    print("The bool is True!")


# Perform logical operations on the boolean

result = bool_val and False


# Convert the bool to an int

my_int = int(bool_val)
```

In the first example, the code compares the boolean **True** to the boolean literal **True**, and prints a message if they are equal (they are). In the second example, the code performs a logical **and** operation on the boolean **True** and the boolean **False**, and assigns the result (**False**) to the **result** variable. In the third example, the code converts the boolean **True** to the integer **1**, and assigns the result to the **bool_val** variable, since **True** can map to 1 and **False** can map to 0.

Note that in Python, **True** and **False** are not just special values, but are actually boolean constants that are instances of the **bool** class. This means that you can use the **is** keyword to check if a value is a **bool**, and you can use the **bool** class to create new **bool** objects. For example:

```
# Check if a value is a bool

if isinstance(True, bool):

    print("The value is a bool.")
```

In the example, the code checks if the boolean literal **True** is an instance of the **bool** class, and prints a message if it is.

Can you explain None in Python?

In Python, **None** is a special value that represents the absence of a value or a **null** value. It is an object of its own data type. **None** is often used to represent the default value of a function or method that does not have a return value. It is also commonly used to represent the end of a list of items.

None is a unique value in Python and it is not equivalent to any other value, not even to an empty string or 0. This means that, for example, you cannot use the **equality operator** (==) to compare a variable with **None** to see if it is equal to the null value. Instead, you have to use the **is** keyword operator to check for **None**:

```
x = None



if x is None:
    print("x is None")
```

In Python, **None** is a falsy value, which means that it is considered to be **False** in a **boolean** context. This means that, for example, you can use it as a condition in an if statement or a while loop to check if a value has been assigned to a variable or not.

```
a = None


if a:

    print("a has a value that isn’t None")

else:

    print("a is None")
```

In programming, what is casting?

That’s where you use your code wizardry to start a magic spell, of course.

More seriously though, **casting** in programming refers to the process of converting a variable from one data type to another data type. This is often necessary when working with different data types in the same program. For example, if you have a variable that holds an integer value and you want to convert it to a string so that you can **concatenate** (combine) it with another string, you would need to cast the integer to a string. This is typically done using a **typecast operator**, which specifies the data type that you want the variable to be converted to.

How do I cast variables in Python?

In Python, you can use the **str()**, **int()**, **float()**, and **bool()** functions to cast variables to the corresponding data types. For example, to cast a variable to a string, you would use the str() function, like this:

```
x = 6

str_val_of_x = str(x)
```

After this code has been executed, the **str_val_of_x** variable will hold the string value "6", which can be concatenated with other strings in your program.

Similarly, you can use the **int()** function to cast a variable to an integer, the **float()** function to cast a variable to a decimal number, and the **bool()** function to cast a variable to a bool value. For example:

```
a = "7"

int_a = int(a)



b = "3.141592654"

float_b = float(b)



c = "True"

bool_c = bool(c)
```

After this code has been executed, the **int_a** variable will hold the integer value 7, the **float_b** variable will hold the floating-point value 3.141592654, and the **bool_c** variable will hold the Boolean value True.

You can also use these functions to explicitly convert variables from one data type to another. For example, if you have a variable that holds an integer value and you want to convert it to a floating-point value, you can use the **float()** function to do this, like this:

```
my_variable = 5

float_my_variable = float(my_variable)
```

After this code has been executed, the **float_my_variable** variable will hold the floating-point value 5.0.

How do I use "is" in Python?

In Python, the **is operator** is used to test whether two variables refer to the same object. It returns True if the two variables refer to the same object, and False if they refer to different objects.

Here is an example:

```
x = [4, 5, 6]

y = [4, 5, 6]

z = x


print(x is y)
> False
print(x is z)
> True
```

In the above example, **x** and **y** are two *different* objects that have the same *value*, so the **is operator** returns False when they are compared. However, **z** is assigned the value of **x**, so **x** and **z** refer to the *same* object, and the **is operator** returns True when they are compared.

The **is operator** is different from the **equality operator** (**==**), which is used to test whether two values are equal. The **is operator** checks for object identity, while the equality operator checks for value equality. Here is an example that illustrates the difference between **is** and **==**:

```
a = [6, 7, 9]

b = [6, 7, 9]


print(a == b) > True

print(a is b) > False
```

In the above example, **a** and **b** have the same value, so the equality operator returns True when they are compared. However, **a** and **b** are still two different objects, so the is operator returns False when they are compared.

In Python is there "undefined"?

In Python, there is no special value or object that represents an undefined value like in some other programming languages. If a variable is declared but not assigned a value, it is created with the default value **None**.

Here is an example:

```
x = None

print(x)
```

```
> None
```

In the above example, **x** is declared but not assigned a value, so it is created with the default value **None**. You can use the **is operator** to check if a variable has been assigned a value or not:

It is important to note that **None** and undefined are not the same thing. **None** is a special value that represents the absence of a value, while undefined means that a value has not been assigned to a variable. In Python, you can use the is operator to check if a variable is undefined, but you cannot use it to check for undefined values directly.

If Statements

In programming, what is an "if statement"?

In programming, an **if statement** is a control flow statement that allows the programmer to specify a condition and then execute a block of code only if the condition is true. **If statements** are an essential part of many programming languages, and they are used to control the flow of execution of a program based on the values of variables or expressions.

An if statement typically takes the following form:

```
if (condition):

    # Code here executes when the condition is true

    # Take note of the indentation!
```

In this code, the **if** keyword is followed by a condition in parentheses (), and then a colon (:). The code to be executed if the condition is true is indented below the **if statement**, and it is executed only if the condition evaluates to true. The parentheses are optional as you’ll see below.

How do you use "if statements" in Python?

Here's an example of a simple **if** statement in Python:

```
if x > 10:

    print("x is greater than 10")
```

In this example, the **if statement** checks whether the value of **x** is greater than 10. If it is, then the code inside the **if** block is executed and the message "*x is greater than 10*" is printed to the screen. If **x** is not greater than 10, then the **if** block is skipped and the code continues to execute from the next line.

You can also use **else** and **elif** (short for "else if") statements to specify additional blocks of code that should be executed under different conditions. For example:

```
if x > 10:

    print("x is greater than 10")

elif x == 10:

    print("x is equal to 10")

else:

    print("x is less than 10")
```

In this example, if **x** is greater than 10, then the first **if** block is executed and the message "*x is greater than 10*" is printed. If **x** is not greater than 10 but is equal to 10, then the **elif** block is executed and the message "*x is equal to 10*" is printed. If **x** is not greater than 10 and not equal to 10, then the **else** block is executed and the message *"x is less than 10*" is printed.

if statements are a fundamental building block of Python programs, and they allow you to control the flow of your code based on the values of variables and other conditions.

How do I use "and" and "or" logic in Python if statements?

In Python, you can use the **and** and **or** keywords to combine multiple conditions in an **if** statement.

The **and** keyword allows you to specify multiple conditions that must all be true in order for the code in the **if** block to be executed. For example:

```
# Check if a number is positive and even

if num > 0 and num % 2 == 0:

    print("The number is positive and even.")
```

In this code, the **if** statement checks if the value of the **num** variable is greater than **0** and also if the value of **num modulo** 2 is equal to **0**. If both of these conditions are true, the code in the **if** block is executed, and the message "*The number is positive and even.*" is printed.
The **modulo** operator, shown here in its code form as “**%**”, returns the *remainder* of a division operation. For example, *9* ***%*** *4* would return 1, because 4 goes into 9 twice with 1 remainder. You can easily check if a number is even by using *x* ***%*** *2 == 0*, because an even number will never have a remainder when divided by 2 but an odd number will.

The **or** keyword allows you to specify multiple conditions, of which at least one must be true in order for the code in the **if** block to be executed. For example:

```
# Check if a number is zero or negative

if num == 0 or num < 0:
    print("The number is zero or negative.")
```

In this code, the **if statement** checks if the value of the **num** variable is equal to 0 or if it is less than 0. If either of these conditions is true, the code in the **if** block is executed, and the message "*The number is zero or negative.*" is printed.

Note that you can use parentheses to group multiple conditions together and specify the order in which they are evaluated. For example:

```
# Check if a number is positive and even, or zero or negative

if (num > 0 and num % 2 == 0) or (num == 0 or num < 0):

    print("The number is positive and even, or zero or negative.")
```

In this code, the **if** statement first checks if the value of the **num** variable is greater than 0 and also if the value of **num modulo 2** is equal to 0. If both of these conditions are true, the code in the **if** block is executed, and the message is printed. If either of these conditions is false, the **if** statement then checks if the value of the **number** variable is equal to 0 or if it is less than 0. If either of these conditions is true the message will be printed.

How does scope work in Python?

In Python, the **scope** of a variable is the part of the program where the variable is defined and can be accessed. There are two types of **scope** in Python: **global scope** and **local scope**.

Global scope refers to the part of the program where a variable is defined *outside* of any function or class. A variable defined in **global scope** can be accessed from anywhere in the program, including inside **functions** and **classes**. For example, consider the following code:

```
x = 7

# x is in the global scope


def cool_function():

    print(x) # x can be accessed here in a function


cool_function()
```

```
> 7
```

In this code, the variable **x** is defined in global scope, outside of the **cool_function** function. Therefore, **x** can be accessed inside the function, and when the function is called, it prints the value of **x**, which is 5.

Local scope refers to the part of the program where a variable is defined *inside* a function or class. A variable defined in **local scope** can only be accessed inside the function or class where it is defined. For example, consider the following code:

```
def cool_function():

    x = 7 # x is in local scope

    print(x) # x can be accessed in the function


cool_function()
```

```
> 7
```

```
print(x)

# x is not in the global scope so this will cause an error!
```

In this code, the variable **x** is defined inside the **cool_function** function, so its **scope** is *local* to that function. Therefore, **x** can only be accessed inside the function, and when the function is called, it prints the value of **x**, which is 5. However, if you try to access **x** outside of the function, in the **global scope**, you will get an error, because **x** is not defined in **global scope**.

Another important aspect of scope in Python is the concept of **scope nesting**. This refers to the fact that **local scope** can be nested inside other **local scope**, and the most inner scope will take precedence over outer scopes. For example:

```
x = 8

# x is declared in the global scope


def outer():

    x = 10

    # x made again in local scope (outer function)


    def inner():

        x = 15 # x is declared again in local scope (inner function)

        print(x) # this will print 15 because of the INNERmost scope with x


    inner()
    print(x) # will print 10 as the outer scope takes precedence here
```

```
outer()

print(x) # this will print 8 from the global scope
```

In this code, there are three levels of scope: **global scope**, **local scope** (outer function), and another **local scope** (inner function). When the **inner** function is called, it prints the value of **x** that is defined in its own **local scope**, which is 15. Then, when the **outer** function is called, it prints the value of **x** that is defined in its own local scope, which is 10. Finally, when the code is run, it prints the value of **x** that is defined in **global scope**, which is 8.

Scope in Python is a mechanism that determines where variables are defined and can be accessed. Variables defined in **global scope** can be accessed from anywhere in the program, while variables defined in **local scope** can only be accessed inside the function or class where they are defined.

Does Python have "switch" statements?

In Python, there is no **switch statement** like in other programming languages. Instead, Python provides a more powerful and flexible way to handle conditional branching using **if-elif-else statements**.

Here is an example of how you can use **if-elif-else statements** to implement the same functionality as a switch statement:

```
def switch_case_example(argument):
    if argument == 1:
        print("Case A")
    elif argument == 2:
        print("Case B")
    elif argument == 3:
        print("Case C")
    else:
        print("Default")

switch_case_example(1)
> Case A

switch_example(2)
> Case B
 switch_example(3)
> Case C

switch_example(4)
> Default
```

In the above example, the **switch_case_example()** function takes an argument and uses **if-elif-else statements** to evaluate it and print a different message for each possible case.

If-elif-else statements allow you to handle multiple cases more concisely and more easily than using a series of **if statements**. You can also use the **elif** keyword to combine multiple cases into one statement, which can make your code more readable and easier to maintain.

Chapter Review

- What is a variable?
- How do you declare a variable with a value?
- How is an int different than a float?
- What is a constant?
- What is a String?
- What is a boolean?
- How do you make an if statement?
- How can you use “and” and “or” to make your if statement conditions more complex?
- Does Python have a “switch” block?
- What is scope?
- What is local scope versus global scope?
- What is the modulo operator?
- What is None?
- What is the “is” keyword for?

Chapter 4
Common String Operations

How do I reverse a String in Python?

To reverse a string in Python, you can use the built-in **reversed()** function. Here is an example:

```
# Make a string

str_val = "Hello there!"


# Use the reversed() function to get a list that has  the characters of the string in reverse order

reversed_string = reversed(str_val)




# Print the reversed string

print("".join(reversed_string))
```

```
> !ereht olleH
```

This will print the original string reversed.

Alternatively, you can use **slicing** to reverse a string in Python. Here is an example of how to do that:

```
# Make a string

str_val = "Hello there!"


# Use slicing to reverse the string

reversed_string = str_val[::-1]


# Print the reversed string

print(reversed_string)
```

This method is a bit more concise and can be easier to read, especially if you are already familiar with **slicing** in Python.

How do you make a String in Python all uppercase?

To make a string in Python all uppercase, you can use the **str.upper()** method. This method returns a new string with all the characters in the original string converted to uppercase.

Here is an example of how to use the **str.upper()** method to convert a string to uppercase:

```
# Make a string

str_val = "Hello there!"
```

```
# Use the `upper()` method to convert the string to uppercase
uppercase_string = str_val.upper()


# Print the uppercase string
print(uppercase_string)
```

```
> HELLO THERE!
```

This will print the original string converted to uppercase.

If you want to modify the original string in-place, instead of creating a new string, you can use the **str.upper()** method in combination with the **assignment operator =**. This will replace the original string with a new string containing all uppercase characters.

Here is an example of how to do this:

```
str_val = "Hello there!"


# Assign the variable to itself, but upper cased
str_val = str_val.upper()


# Print the uppercase string
print(str_val)
```

This will print the same string as before.

Alternatively, you can use the **str.upper()** method in combination with the **print()** function to print the string in uppercase directly, without creating a new string. Here is an example of how to do this:

```
str_val = "Hello there!"

print(str_val.upper())
```

This will also print the string "*HELLO THERE!*".

How do I turn a String into a number in Python?

To convert a string to a number in Python, you can use the built-in **int()** and **float()** functions. These functions take a string as an argument and return the corresponding integer or float number, respectively.

Here is an example of how to use the **int()** function to convert a string to an integer:

```
# Make a numeric string

str_val = "42"


# Use the `int()` function to convert the string to an integer

int_val = int(str_val)





# Print the integer

print(int_val)
```

```
> 42
```

This will print the number *42*, which is the integer representation of the string "42".

Here is an example of how to use the **float()** function to convert a string to a floating-point number:

```
# Make a numeric string with a decimal

str_val = "3.14159"


# Use the `float()` function to convert the string to a float

float_val = float(str_val)


# Print the float

print(float_val)
```

```
> 3.14159
```

If the string cannot be converted to the specified type (for example, if it contains letters instead of numbers), the **int()** and **float()** functions will raise a **ValueError** and will crash your program unless handled correctly.

For example:

```
str_val = "hello"


# Use the `int()` function to convert the string to an integer

int_val = int(str_val)
```

This will raise a **ValueError**, because the string "hello" cannot be converted to an integer.

To handle this error, you can use a **try/except** block to catch the **ValueError** and handle it.

```
# Make a string that is NOT a number

str_val = "hello"



# Use a `try`/`except` block to handle the `ValueError` that may be raised by the `int()` function

try:

    int_val = int(str_val)

except ValueError:

    print("Trouble converting to an int!")
```

This code will print the message "Could not convert string to integer.", instead of raising an error.

Note that the **int()** function will only convert strings that contain whole numbers to integers. If the string contains a fraction (for example, "3.14159"), the **int()** function will only return the integer (whole number) part of the number (in this case, 3). To get the full floating-point representation of the number, you must use the **float()** function instead.

How do I turn a number into a String in Python?

To convert a number to a string in Python, you can use the built-in **str()** function. This function takes a number as an argument and returns the corresponding string representation of that number.

Here is an example of how to use the **str()** function to convert a number to a string:

```
int_val = 42


# Use the `str()` function to convert the number to a string

str_val = str(int_val)


# Print the string

print(my_string)
```

This will print the string "42", which is the string representation of the number 42.

Here is another example that shows how to use the **str()** function to convert a float number to a string:

```
int_val = 3.14159


# Use the `str()` function to convert the number to a string

str_val = str(int_val)


# Print the string

print(my_string)
```

```
> 3.14159 (as a String)
```

Note that the **str()** function will only convert numbers to strings. If you pass it a non-numeric value (such as a string or a list), it will raise a **TypeError**. For example:

```
# Start with a non-numeric string value

some_value = "hey"



# Use the `str()` function to convert the value to a string

string_value = str(some_value)
```

This will raise a **TypeError**, because the **str()** function cannot convert the string "hey" to a number.

To handle this error, you can use a **try/except** block to catch the **TypeError** and handle it gracefully. For example:

```
str_val = "hello there"



# Use a `try`/`except` block to handle the `TypeError` that may be raised by the
`str()` function

try:

    str_val = str(str_val)



except TypeError:

    print("Could not convert value to string!")
```

```
> Could not convert value to string!
```

This code will print the message instead of raising an error.

How do I format strings in Python?

To format strings in Python, you can use the **format()** method. This method allows you to substitute placeholders in a string with values of your choosing. Here's an example:

```
# Define a string with some placeholders in it

string = "Hey {}! Today is {}!"



# Use the format() method to substitute the placeholders with values

formatted = string.format("Luke", "Saturday")
```

```
# Print the formatted string to the console print(formatted)
```

```
> Hey Luke! Today is Saturday!
```

As you can see, the **format()** method substitutes the placeholders inside curly braces in the original string with the values that are passed to it.

You can use the **format()** method to insert any type of value into a string, including numbers, strings, and even other objects. You can also use it to control the formatting of the substituted values, such as specifying the number of decimal places for floating-point numbers or the alignment of the text.

There are a bunch more options and arguments you can use with string formatting. For more information on how to use the **format()** method you can check out the Python documentation.

How do I split a String in Python?

To split (turn a single string into a group of strings) a string in Python, you can use the **str.split()** method. This method splits a string into a list of substrings, using a specified separator string to identify where the split should occur.

Here is an example of how to use the **str.split()** method to split a string:

```
str_val = "Hey there"


# Use the `split()` method to split the string on the `,` character

split_string = str_val.split(",")


# Print the resulting list of substrings

print(split_string)
```

```
> [“Hey”, “there”]
```

If you don't specify a separator string, the **split()** method will use any whitespace characters (such as spaces, tabs, and newlines) as the separator.

For example:

```
str_val = "That’s no moon"


# Use the `split()` method without specifying a separator

split_string = str_val.split()


# Print the resulting list of substrings

print(split_string)
```

```
> [“That’s”, “no”, “moon”]
```

You can also specify multiple separator strings by passing a list of strings to the **split()** method. The string will be split on any of the specified separator strings. For example:

```
str_val = "Luke! I am your father!"


# Use the `split()` method to split the string on the `,` and `!` characters

split_string = str_val.split(",", "!")
```

```
# Print the resulting list of substrings

print(split_string)
```

```
> [“Luke”, “I”, “am”, “your”, “father”]
```

How do I find a substring in a String in Python?

To find a substring in a string in Python, you can use the **str.find()** method. This method returns the index of the first occurrence of the substring in the string, or -1 if the substring is not found.

Here is an example of how to use the **str.find()** method to find a substring in a string:

```
str_val = "Hello there!"


# Use the `find()` method to find the substring "there"

index = str_val.find("there")


 # Print the index of the substring

print(index)
```

```
> 6
```

This will print the number 6, which is the index of the first character of the substring "there" in the string.

If you want to find the last occurrence of a substring in a string, you can use the **str.rfind()** method instead. This method works the same way as **str.find()**, but it returns the index of the last occurrence of the substring in the string, instead of the first.

Here is an example of how to use the **str.rfind()** method to find the last occurrence of a substring in a string:

```
str_val = "General Kenobi! General Kenobi!"


# Use the `rfind()` method to find the last occurrence of the substring "General"

index = str_val.rfind("General")





# Print the index of the substring

print(index)
```

This will print the number 16, which is the index of the first character of the last occurrence of the substring "General" in the string.

You can also use the **in** keyword to check if a substring is contained in a string. This keyword returns **True** if the substring is found in the string, and **False** otherwise.

Here is an example of how to use the **in** keyword to check for the presence of a substring in a string:

```
str_val = "Death Star"


# Use the `in` keyword to check if the substring "Star" is in the string

if "Star" in my_string:

    print("Found it!")

else:
    print("Where is it?!")
```

```
> Found it!
```

This will print the message "*Found it!*", because the substring "Star" is indeed contained in the string.

Regex

What is a RegEx?

A **regular expression** (shortened as RegEx or RegExp) is a sequence of characters that defines a search pattern. Typically, this is used for matching text with a certain pattern, such as finding all phone numbers in a document. Regular expressions are a powerful tool for working with text, and they are supported by many programming languages.

How do I use RegEx in Python?

To use regular expressions in Python, you first need to import the **re module**. This **module** provides functions and methods for working with regular expressions.

Once you have imported the **re module**, you can use the **re.search()** method to search for a pattern in a string. This method returns a match **object** if the search pattern is found, or **None** if the pattern is not found.

Here is an example of how to use the **re.search()** function to search for a pattern in a string:

```
import re
```

```
# search for the pattern "hello" in the string "hello there"

result = re.search("hello", "hello there")



# check if a match was found

if result:

    print("Match found:", result.group())

else:

    print("Match not found!")
```

In this example, the **re.search()** method will return a match object if the pattern **"hello"** is found in the string **"hello there"**. The **result.group()** method is then used to print the part of the string that matches the pattern.

You can also use the **re.findall()** method to find all occurrences of a pattern in a string. This method returns a list of all matches found in the string.

Here is an example of how to use the **re.findall()** method to find all occurrences of a pattern in a string:

```
import re



# find all occurrences of the pattern "hello" in the string "hello world, hello friends"

results = re.findall("hello", "hello there, hello vader")



# print the list of all matches found

print("Matches found:", results)
```

In this example, the **re.findall()** method will return a list of all occurrences of the pattern **"hello"** in the string **"hello there, hello vader"**. The list of matches will be printed to the screen.

You can also use special characters in your regular expressions to match specific types of characters or character combinations. For example, the *\d* character class matches any digit (0-9), and the * character matches zero or more occurrences of the preceding character or character class.

Chapter Review

- What is a string?
- How is a string different than other variable data types like int?
- How do you convert a string to an int?
- How do you reverse a string?
- How do you find a string in a string?
- How do you format a string?
- What is a regex?

Chapter 5
Loops

In programming, what is a loop?

In programming, a **loop** is a control flow statement that allows the programmer to repeat a block of code a certain number of times, or until a certain condition is met. **Loops** are an essential part of many programming languages, and they are used to perform repetitive tasks or operations efficiently and easily.

There are several different types of loops, and the specific type of loop used in a program depends on the specific requirements and characteristics of the task at hand. Some of the most common types of loops are:

- **for loops**, which are used to iterate over a sequence of values, such as the elements of a list or the characters in a string
- **while loops**, which are used to repeat a block of code until a certain condition is met
- **do-while loops**, which are similar to **while loops**, but they guarantee that the code inside the loop will be executed at least once

What is a loop in Python?

In Python, a **loop** is a way to repeat a block of code multiple times. There are two types of loops in Python: **for** loops and **while** loops.

For loops are used to execute a block of code a specified number of times, and **while** loops are used to execute a block of code until a certain condition is met. Here are some examples of **for** and **while** loops:

```python
# For loop

for i in range(10):

    print(i)
```

```python
# While loop

i = 0

while i < 10:

    print(i)

    i += 1
```

In the first example, the **for** loop is used to print the values 0 through 9 to the screen. The **range()** function is used to generate a sequence of numbers, and the **for** loop iterates over that sequence, assigning each value to the **i** variable.

In the second example, the **while** loop is used to achieve the same result as the **for** loop. The **i** variable is initialized to 0 outside the loop, and then the **while** loop is used to print the value of **i** and increment it by 1 until **i** is no longer less than 10.

What is the range function in Python?

In Python, the **range()** function is a built-in function that generates a sequence of numbers. It is commonly used in **for** loops to iterate over a sequence of numbers and perform the same action on each number in the sequence.

The **range()** function takes three arguments: a start value, a stop value, and a step size. The start value is the first number in the sequence, and the stop value is the number that the sequence stops at (but does not include). The step size is the amount by which the sequence should increment on each iteration. Here are some examples of how to use the **range()** function in Python:

```
# Start at 0, stop at 10, increment by 1

for i in range(0, 10, 1):

    print(i)



# Start at 5, stop at 15, increment by 2

for i in range(5, 15, 2):

    print(i)



# Start at 10, stop at 0, increment by -1

for i in range(10, 0, -1):

    print(i)
```

In the first example, the **range()** function generates a sequence of numbers from 0 to 9 with a step size of 1. The **for** loop iterates over this sequence, and on each iteration the value of **i** is printed to the screen. In the second example, the **range()** function generates a sequence of numbers from 5 to 14 with a step size of 2. In the third example, the **range()** function generates a sequence of numbers from 10 to 1 with a step size of -1.

The **range()** function is a useful and versatile tool for generating sequences of numbers in Python. It is commonly used in **for** loops, but it can also be used in other contexts where a sequence of numbers is needed.

What does the slice operator do in Python?

In Python, the **slice** notation *[start:end]* is used to extract a sublist from a list or a substring from a string. It includes the elements from the start index (inclusive) to the end index (exclusive).

The **slice** notation allows you to specify either or both the start and end indices as optional arguments. If you omit the start index, it is assumed to be 0. If you omit the end index, it is assumed to be the length of the list or string.

In the case of *[-1 :]*, the start index is -1, which means the last element in the list or string, and the end index is not specified, which means all the elements from the start index to the end of the list or string.

Here are a few examples of using the slice notation with lists:

```
# create a list of numbers

numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]


# print the last element of the list

print(numbers[-1])

> 9


# print the last two elements of the list

print(numbers[-2:])

> [8, 9]
```

```
# print the last three elements of the list

print(numbers[-3:])

> [7, 8, 9]


# print a copy of the list with only the last number remaining

print(numbers[-1:])

> [9]
```

Here are a few examples of using the slice notation with strings:

```
str_val = 'Hello there!'


# print the last character of the string

print(str_val[-1])

> !


# print the last two characters of the string

print(str_val[-2:])

> e!


# print the last three characters of the string

print(str_val[-3:])

> re!
```

What is a "for loop" in Python?

In Python, a **for** loop is a way to execute a block of code multiple times. It is used to iterate over a sequence of values, such as a list or a range of numbers, and perform the same action on each value in the sequence.

To write a **for** loop in Python, you use the **for** keyword followed by a variable name, the **in** keyword, and the sequence you want to iterate over. The code that should be executed on each iteration of the loop goes inside a block indented under the **for** statement. Here's an example of a simple **for** loop in Python:

```
for i in range(10):

    print(i)
```

In this example, the **for** loop is used to print the values 0 through 9 to the screen. The **range()** function is used to generate a sequence of numbers from 0 to 9, and the **for** loop iterates over that sequence, assigning each value to the **i** variable in turn. On each iteration of the loop, the value of **i** is printed to the screen.

For loops are a powerful and important feature of Python, and they allow you to write concise, readable code to perform the same action on multiple values. They are commonly used to iterate over the elements of a list, but they can be used with many other types of sequences as well.

How do I use "while loops" in Python?

In Python, **while** loops are used to execute a block of code multiple times until a certain condition is met. To write a **while** loop, you use the **while** keyword followed by the condition you want to check, and then a colon. The code that should be repeated goes inside a block indented under the **while** statement. Here's an example of a simple **while** loop in Python:

```
while x < 10:

    print(x)

    x += 1
```

In this example, the **while** loop will execute the code inside the loop repeatedly as long as the value of **x** is less than 10. Each time the loop runs, the value of **x** is printed to the screen, and then **x** is incremented by 1. This means that the loop will print the values 0, 1, 2, 3, 4, 5, 6, 7, 8, and 9 to the screen before the condition **x < 10** is no longer true and the loop is terminated.

It's important to make sure that the condition in a **while** loop will eventually become false, otherwise the loop will run indefinitely and your program will never finish. You can use the **break** keyword to exit a **while** loop prematurely if a certain condition is met. For example:

```
while True:

    if x > 10:

        break

    print(x)

    x += 1
```

In this example, the **while** loop will run indefinitely because the condition **True** is always true. However, the **if** statement inside the loop checks whether **x** is greater than 10, and if it is, then the **break** statement is executed and the loop is terminated.

Chapter Review

- What is a loop?
- How are for loops and while loops different?
- What is the range function?
- What does the slice operator do?

Chapter 6
Functions, Comments, and Modules

Functions

In programming, what is a function?

In programming, a **function** is a self-contained block of code that performs a specific task or operation. A **function** typically takes some input, performs some operations on the input, and then returns an output. This allows a **function** to be used multiple times within a program, making it easier to write and maintain your code.

Functions are an essential part of many programming languages, and they are used to modularize and organize code into reusable, self-contained units. This makes it easier to understand and maintain large codebases, as well as allowing for code reuse and abstraction.

In most programming languages, a function is defined using a **function declaration** or definition, which specifies the name of the function, the input parameters that the function takes, and the operations that the function will execute.

How do I call a function in Python?

To **call** a **function** in Python, you use the function's name followed by a pair of parentheses (). For example, if you have a function named **cool_function** that you want to call, you would do the following:

```
cool_function()
```

If the function takes **arguments**, you can specify the **arguments** within the parentheses when you call the function. For example, if **cool_function** takes two arguments, **a1** and **a2**, you would call the function like this:

```
cool_function(a1, a2)
```

The **arguments** can be any valid Python expressions, and you can specify them in any order if the **function** is defined to accept arguments by position. If the **function** is defined to accept arguments by name, you can specify the arguments in any order, as long as you include the argument names in the **call**.

For example, if **cool_function** is defined to accept **arguments** by position, you could call it like this:

```
cool_function(3, 5)
```

Or you could call it like this:

```
cool_function(5, 3)
```

Both of these calls would be valid, but the values of **a1** and **a2** would be different depending on the order in which the **arguments** were specified.

On the other hand, if **cool_function** is defined to accept **arguments** by name, you could call it like this:

```
cool_function(a1=3, a2=5)
```

Or you could call it like this:

```
cool_function(a2=5, a1=3)
```

Either way, the values of **a1** and **a2** would be the same in the **function call**, because the **arguments** are specified by name.

Overall, calling a **function** in Python is a simple and straightforward process. You just need to use the function's name followed by a pair of parentheses, and specify any required arguments within the parentheses.

How do you write functions in Python?

To write a **function** in Python, you use the **def** keyword followed by the name of the function and any parameters it takes. The code for the function goes inside a block indented under the definition line, and you can use the **return** keyword to specify the value or values that the function should output. Here's an example of a simple function that takes two numbers as input and returns their sum:

```
def add(x, y):

    return x + y
```

To call this function and use the result in your code, you would write something like this:

```
result = add(3, 4)

print(result)
```

```
> 7
```

Functions are a powerful and important feature of Python, and they allow you to write modular, reusable code.

How do I return values from functions in Python?

To return a value from a function in Python, you use the **return** keyword followed by the value you want to return. For example, consider the following code:

```
def cool_function(x, y):

    z = x + y

    return z



result = cool_function(5, 10)
```

In this code, the **cool_function** function is defined to take two arguments, **x** and **y**, and it returns the sum of those two **arguments**. This is done by performing the addition operation on **x** and **y**, storing the result in the **z** variable, and then using the **return** keyword followed by **z** to return the value of **z** from the function.

When the **cool_function** function is called with the arguments **5** and **10**, the function performs the addition operation and returns the value of **z**, which is 15. This value is then stored in the **result** variable, so that it can be used outside the function.

You can return any valid Python expression from a function using the **return** keyword, and you can use the **return** keyword multiple times within a function to return different values depending on the conditions of the function. For example, consider the following code:

```
def my_function(x, y):

    if x > y:

        return x

    else:
```

```
    return y


result = my_function(5, 10)
```

In this code, the **my_function** function is defined to take two arguments, **x** and **y**, and it returns the larger of the two **arguments**. This is done by using an **if** statement to check if **x** is greater than **y**, and if so, returning **x** from the function. Otherwise, the **else** clause is executed and **y** is returned from the function.

When the **my_function** function is called with the arguments **5** and **10**, the **if** statement is evaluated and the **else** clause is executed, so the function returns the value of **y**, which is 10. This value is then stored in the **result** variable.

Returning values from functions in Python is a simple and powerful way to pass results from the function to the code that called the function, and it is an important part of writing modular and reusable code.

What is the difference between a function and a method?

In programming, a **function** is a block of code that performs a specific task and can be called by other parts of the code. A **method** is a type of function that is associated with an object, and it is used to perform operations on the data stored in the object.

The main difference between a **function** and a **method** is that a function is a standalone block of code, while a method is associated with an object and is used to operate on the data stored in the object. A function can be called independently of any objects, while a method must be called on an **object**.
What is a function argument in programming?

A **function argument** (or parameter) is a value that is passed to a function when it is called. The **function** can then use this value to perform its operations.

Function arguments are used to provide additional information or instructions to the function, and they can be required or optional. **Required arguments**, also known as **positional arguments**, must be provided when the function is called, in the correct order. **Optional arguments**, also known as **keyword arguments**, can be omitted when the function is called, and they can be specified in any order.

How do I use function arguments in Python?

To use **function arguments** in Python, you first need to define a function that takes one or more **arguments**. This is done by specifying the names of the arguments within the parentheses () in the function definition, separated by commas. For example, consider the following code:

```
def cool_function(a1, a2):
    # code here
```

In this code, the **cool_function** function is defined to take two arguments: **a1** and **a2**. These **arguments** are specified within the parentheses in the **function** definition, and they can be used within the function to refer to the values that are passed to the function when it is called.

To call a function and pass arguments to it, you simply specify the values of the arguments within the parentheses () in the function call, in the same order as they are defined in the function. For example, you could call the **cool_function** function defined above like this:

```
cool_function(15, 7)
```

This would pass the values **15** and **7** to the **a1** and **a2** arguments, respectively. The function could then use these values to perform its operations.

Alternatively, if the function is defined to accept arguments by name, you can specify the arguments in any order in the function call, as long as you include the argument names. For example, you could call the **cool_function** function like this:

```
cool_function(a2=7, a1=15)
```

This would have the same effect as calling the function with **cool_function(15, 7)**, but the **arguments** are specified by name instead of by position.

How do I make default arguments in Python?

To create a **default argument** in Python, you can use the equal sign to assign a **default value** to the parameter in the function definition. Here is an example:

```
def say_hi(name, message = "What's up,"):
    print(message + " " + name)


# Call the function with the default message
greet("Anakin")


# Call the function with a different message
greet("General Kenobi", "Hello there")
```

In this example, the **say_hi()** function has a default argument for the **message** parameter, which is set to "Hello". This means that if we call the **say_hi()** function without passing a value for the **message** parameter, it will use the default value.

When we call the **say_hi()** function with only one argument, the default value for **message** is used, and the output is "*What's up, Anakin*".

However, if we call the **say_hi()** function with two arguments, the second argument will be used as the value for the **message** parameter, and the output will be "*Hello there General Kenobi*".

It is important to note that **default arguments** must be placed after any non-default arguments in the function definition. In other words, in the **greet()** function above, the **name** parameter must come before the **message** parameter, because **name** does not have a default value. If you try to put a default argument before a non-default argument, you will get a syntax error.

How do I make optional arguments in Python?

To create **optional arguments** in Python, you can use **default values** for the **arguments** in the function definition. When an argument has a **default value**, it is considered to be optional, and the caller of the function can choose whether or not to provide a value for that argument.

Here is an example of how to define a function with optional arguments in Python:

```
def cool_function(a1, a2=None):

# a1 is a required argument, while a2 is optional

# If a2 is not provided, it will default to None
```

```
if a2 is not None:
    # Do something with a2
    pass


# Do something with a1
pass
# “pass” is a keyword that is a placeholder for future code
```

In this example, the **cool_function()** function is defined with two **arguments**: **a1** and **a2**. **a1** is a required argument, which means that the caller of the function must provide a value for it. **a2**, on the other hand, is an optional argument, because it has a default value of **None**. This means that the caller of the function can choose whether or not to provide a value for **a2**.

To call the function with **optional arguments**, you can simply provide values for the **required arguments**, and omit the optional arguments.

What is type hinting in Python?

Type hinting is a feature in Python that allows you to provide hints about the data types of function parameters and return values. These hints can be used by static type checkers, **IDEs**, and other tools to provide better code completion, error checking, and other functionality.

In Python, type hints are optional, and they are specified using annotations in the function signature. Here is an example of a function with type hints in Python:

```
def sum(x: int, y: int) -> int:
    return x + y
```

In this example, the **sum()** function has two parameters, **x** and **y**, and a return value. The **type hints** for these elements are specified using **annotations**, which are placed after the name of the element and a colon (:). The type hints for **x** and **y** are **int**, which indicates that these parameters should be integers, and the type hint for the return value is **int**, which indicates that the function should return an integer.

Type hints can be used to provide information about the data types of function parameters and return values, and they can help static type checkers, **IDEs**, and other tools to provide better code completion and error checking. They are optional in Python, and they do not affect the runtime behavior of the code.

How do I use with/as in Python?

In Python, the **with statement** is used to wrap the execution of a block of code with methods defined by a **context manager**. The with statement allows you to specify a block of code to be executed, and then automatically takes care of cleaning up after the block of code has been executed, even if an exception is raised.

For example, when you *open* a file stream to read in the contents of a file, you are meant to *close* it after you are done so the system can clean up resources. Using a **with statement** the stream will automatically be closed for you.

Here is an example of how to use the with statement in Python:

```
with open('file.txt', 'r') as f:



 # Read the file
```

```
    content = f.read()


# the file is automatically closed after the with block is exited
```

In this example, the **open()** function is used to open a file for reading, and the file is automatically closed after the with block is exited, even if an exception is raised within the block.

The **as** keyword is used to specify a name for the object being managed by the **context manager**. In the example above, the file object returned by the **open()** function is being managed by the context manager, and it is given the name *f* within the with block.

You can also use the as keyword with other types of **context managers**, such as those used for acquiring and releasing locks or for creating and destroying database connections.

```
import threading


lock = threading.Lock()
```

```
# Acquire a thread lock

with lock:

    # critical section of code

    # lock is automatically released when the with block is exited
```

How do I use generic typing in Python?

Python has a mechanism for specifying **generic types** using the typing module, which was introduced in Python 3.5.

The typing module provides a set of **classes** and **decorators** that can be used to **annotate** the types of variables, **function arguments**, and return values in your code. These annotations can be used by static type checkers and **IDEs** to provide type hints and improve code completion and error detection.

Here is an example of how to use generic types in Python:

```
from typing import List, Union


def concat(items: List[Union[str, int]]) -> str:

   result = ''

   for item in items:

       result += str(item)

   return result


print(concat(['a', 'b', 'c']))

> abc

print(concat([1, 2, 3]))

> 123

print(concat(['r', 2, 'd']))

> r2d
```

In this example, the **concat()** function takes a list of items and returns a string by **concatenating** the string representations of the items. The **List** and **Union** types from the typing module are used to specify that the items argument is a list of elements that can be *either strings or integers*, and the return type is a string.

Note that these type annotations are not checked at runtime and do not affect the behavior of the code. They are purely for the benefit of static type checkers and **IDEs**.

What are decorators in Python?

In Python, a **decorator** is a **function** that takes another function and extends the behavior of the latter function without explicitly modifying its code. **Decorators** are a powerful and useful feature of the Python language that allow you to modify the behavior of functions or classes in a concise and flexible way.

Here is an example of how to use a decorator in Python:

```python
def my_decorator(func):
  def wrapper(*args, **kwargs):
     print("Before call")
     result = func(*args, **kwargs)
     print("After call")
     return result
  return wrapper

@my_decorator
def say_hi(name):
  print(f"Hello there, {name}!")
```

```
say_hi("General Kenobi")
```

In this example, the **my_decorator()** function is a **decorator** that takes a **function** as an argument and returns a wrapper function that prints a message before and after calling the decorated function. The **@my_decorator** syntax above the **say_hi()** function is used to apply the decorator to the greet() function.

When you call the **say_hi()** function, it will execute the code in the **wrapper()** function, which will print the messages and then call the **say_hi()** function. The output of this code will be:

```
> Before call

> Hello there, General Kenobi

> After call
```

Decorators can be used to add functionality to existing functions or classes without modifying their code, and they can be composed together to create more complex behaviors.

Comments

What is a comment?

In programming, a **comment** is a line of text in a program that is ignored by the compiler or interpreter, and is intended to be read by humans who are reading the code. Comments are used to explain and document the code, and they are an essential part of writing readable and maintainable programs.

Comments are typically used to provide information about the purpose or behavior of a piece of code, or to explain why a particular decision was made in the code. They can also be used to disable or "*comment out*" sections of code that are no longer needed, but that the programmer wants to keep for reference or future use.

In most programming languages, **comments** are indicated by using a special symbol or sequence of characters. For example, in Python, comments are indicated by using the # character, and anything following the # character on a line is treated as a comment and ignored by the interpreter. For example, consider the following code:

```
# This is a comment

answer = 42

# This is also a comment
```

In this code, the first line is a **comment** that is ignored by the interpreter, and the second line is a statement that assigns 42 to the **answer** variable. The last part of the line, *# This is also a comment*, is also a comment, and it is ignored by the interpreter.

Comments are an essential part of writing readable and maintainable code. It is good practice to use comments liberally in your code to help everyone understand what the code is doing.

What is a docstring?

A **docstring** is a string literal that appears as the first statement in a Python function, method, or module. It is used to provide a brief description of the function, method, or module, and it is intended to make the code easier to understand and use.

Docstrings are enclosed in triple quotes ("""), and they can span multiple lines. Here is an example of a docstring in a Python function:

```
def add_numbers(x, y):

    """This function takes two numbers and returns their sum"""

    return x + y
```

In this example, the **add_numbers()** function has a **docstring** that describes what the function does. This docstring can be accessed using the **__doc__** attribute of the function, and it can be useful for understanding how the function works and what it is intended to do.

Docstrings are an important part of Python's documentation conventions, and they are often used in combination with other tools, such as the **pydoc module**, to generate more detailed documentation for Python programs.

In addition to providing useful information for other programmers, docstrings can also help you remember what your code is intended to do, and they can make it easier to maintain and update your code in the future.

What is PEP 8 style?

PEP 8 is a set of coding style guidelines for Python programs. It was written by Guido van Rossum, the creator of Python, and it was first published in 2001. **PEP** stands for *Python Enhancement Proposal*, and PEP 8 is the eighth proposal in the series.

The purpose of **PEP 8** is to improve the readability and consistency of Python code, and to make it easier for programmers to write and maintain Python programs. **PEP 8** provides guidelines for the use of whitespace, indentation, and other coding conventions in Python programs. It also covers the use of comments, docstrings, and other best practices for writing clean and efficient Python code.

PEP 8 is not a strict set of rules, and not all Python programmers follow it exactly. However, it is widely accepted as a good starting point for writing well-organized and readable Python code. Many Python **IDEs** and code editors have built-in support for PEP 8, and there are tools available that can help you check your code for compliance with PEP 8 guidelines.

Modules

What is a Python module?

A Python **module** is a file that contains Python code and defines a set of functions, classes, or variables that can be used in other Python programs. Modules provide a way to organize and reuse Python code, and they are an essential part of the Python language.

Modules are typically stored in files with the **.py** extension, and they can be imported into other Python programs using the **import** statement. For example, consider the following module named **my_module.py**, which defines a function named **my_function**:

```
def my_function(x, y):

    z = x + y

    return z
```

This module defines a single function named **my_function** that takes two arguments, **x** and **y**, and returns their sum. To use this function in another Python program, you can **import** the **module** and call the function like this:

```
import my_module

result = my_module.my_function(5, 10)
```

In this code, the **import** statement is used to import the **my_module** module, and then the **my_module.my_function** function is called with the arguments **5** and **10**. This calls the **my_function** function defined in the **my_module** module, and the result of the function is stored in the **result** variable.

Modules are an important part of the Python language, as they provide a way to organize and reuse Python code. Modules allow you to split your code into logical units, and they also provide a way to share code with others, or to use code from other libraries and frameworks. Modules are typically stored in files with the **.py** extension, and they can be imported into other Python programs using the **import** statement.

What is pip?

The program **pip** is a package management system for Python, used to install and manage software packages written in Python. It is the default package manager for the Python programming language and is included with most current Python distributions. With **pip**, you can search for, download, and install packages from the Python Package Index (**PyPI**) and other package indexes.

Using **pip**, you can install packages from the command line by running the following command:

```
pip install [package-name]
```

This will install the specified package, along with any other required dependencies. You can also use pip to uninstall packages, upgrade installed packages to the latest version, and manage package installations from multiple sources.

What is the Python standard library?

The Python **standard library** is a collection of **modules** that are included with the Python language, and that provide a wide range of functionality and capabilities. The Python standard library is an essential part of the Python language, and it is included with every installation of Python.

The Python standard library includes modules for a wide range of purposes, including:

- handling input and output
- handling strings and text
- handling numbers and math
- handling dates and time
- handling files and directories
- handling data structures, such as lists and dictionaries
- handling networks and sockets
- handling threads and concurrency
- much more

The Python **standard library** is organized into several sub-libraries, each of which provides a specific set of functionality. For example, the **string** module in the standard library provides a wide range of functions and classes for working with strings and text, while the **math** module provides functions and constants for working with numbers and math.

To use a **module** from the Python **standard library** in a Python program, you can use the **import** statement to import the module, and then use the dot **.** operator to access the functions, classes, and variables defined in the module. For example, to use the **math** module in a Python program, you could do this:

```
import math
```

```
x = math.sin(math.pi / 2)
```

In this code, the **math** module is imported, and then the **math.sin** and **math.pi** functions are used to calculate the sine of *pi/2*. The result of this calculation is stored in the **x** variable.

The Python **standard library** is an essential part of the Python language, and it provides a wide range of functionality and capabilities. The standard library includes modules for a wide range of purposes, and it is organized into several sub-libraries. To use a module from the standard library in a Python program, you can import it using the **import** statement, and then use the dot **.** operator to access the functions, classes, and variables defined in the module.

In Python, how do I import a class from another file?

To import a class from another file in Python, you can use the **import** keyword. For example, let's say you have a file called **my_module.py** that contains a class called **MyClass**. To use this class in another file, you would first need to import it like this:

```
# Import the class from the my_module.py file

from my_module import MyClass
```

Then, you can use the **MyClass** class in your code by creating an instance of it, like this:

```
# Create an instance of the MyClass class

my_object = MyClass()
```

You can then call methods on your **my_object** instance, or access its attributes, just like you would with any other object in Python.

Chapter Review

- What is a function?
- What is a method? How is it different from a function?
- What is an argument?
- What is a docstring?
- What is pip?
- What is a module?
- What is a comment?
- Can a comment ever affect your code?
- What is the standard library?
- What is PEP 8?

Chapter 7

Data Structures

In programming, what is a data structure?

In programming, a **data structure** is a way of organizing and storing data in a computer so that it can be accessed and modified efficiently. Different types of data structures are suited to different kinds of applications, and some are highly specialized to specific tasks. Some common data structures include **arrays**, **linked lists**, **stacks**, **queues**, and **trees**.

A **data structure** is typically implemented as a collection of related data values, and the operations that can be performed on the data structure are defined by its interface. For example, an **array** is a data structure that stores a sequence of values, and it provides operations for accessing, inserting, and deleting elements from the **array**. A **linked list**, on the other hand, is a **data structure** that stores a sequence of nodes, each of which contains a value and a reference to the next node in the sequence. A **linked list** provides operations for inserting and deleting nodes, as well as traversing the list to access the values it contains.

Data structures are an important part of many computer algorithms and data processing systems, as they provide efficient ways of storing and manipulating data. Different data structures have different time and space complexity characteristics, so choosing the right data structure for a given task can have a significant impact on the performance of an algorithm or system. For example, a **stack** is a data structure that provides constant-time insert and delete operations, making it well-suited to applications that require last-in, first-out (LIFO) access to data. A **queue**, on the other hand, provides constant-time insert and delete operations, making it well-suited to applications that require first-in, first-out (FIFO) access to data.

Overall, data structures are an essential part of programming and computer science, and they provide the building blocks for many algorithms and systems that manipulate and process data.

Lists

What is a list?

In programming, a **list** is a **data structure** that is used to store a collection of items. Lists are an essential part of many programming languages, and they are often used to store and manipulate collections of data.

A list is typically implemented as an **array**, which is a contiguous block of memory that is used to store the items in the list. Each item in a list has a corresponding index, which is a numerical value that indicates the position of the item in the list.

In most programming languages, **lists** are zero-indexed, which means that the first item in the list has an index of 0, the second item has an index of 1, and so on.

This allows items in a list to be accessed and manipulated using their index, and it also allows the size of a list to be determined by checking the index of the last item in the list.

For example, in Python, you could define a list like this:

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

In this code, the **numbers** list is defined with the values 1 through 5, and it can be accessed and manipulated using the indices 0 through 4. For example, you could access the third item in the list like this:

```
third_number = numbers[2]

# third_number is now set to 3
```

In this code, the **numbers[2]** expression is used to access the item at index 2 in the **numbers** list, which is the third item in the list. This value is then assigned to the **third_number** variable, which is now set to 3.
Lists are an essential part of many programming languages, and they are often used to store and manipulate collections of data. Lists are typically implemented as **arrays**, and they allow items to be accessed and manipulated using their indices.

Different programming languages have different syntax and behavior for lists, but they all provide a way to store and manipulate collections of data.

Does Python have arrays?

In Python, **arrays** are not a built-in data type like they are in some other languages. However, Python does have a **list** data type, which is similar to an array in some other languages.

How do I sort a list in Python?

In Python, you can sort a list of values using the **sorted()** function. This function takes a list as input and returns a new list containing the same values in a sorted order. By default, the **sorted()** function sorts the values in ascending order, but you can also specify the **reverse=True** keyword argument to sort the values in descending order.

Here's an example of how to use the **sorted()** function to sort a list of numbers in ascending order:

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

sorted_numbers = sorted(numbers)

print(sorted_numbers)
```

```
> [1, 2, 3, 4, 5]
```

In this example, the **sorted()** function is used to sort the **numbers** list in ascending order. The result is a new list called **sorted_numbers** that contains the same values as the original list, but in a sorted order. The **print()** statement is used to print the sorted list to the screen, and the output is **[1, 2, 3, 4, 5]**.

You can also use the **sorted()** function to sort a list of strings or other types of values in a similar way.

How do I slice a list in Python?

To **slice** a **list** in Python, you can use the **list[start:end]** syntax, where **start** is the index of the first element you want to include in the slice and **end** is the index of the first element you want to exclude from the **slice**. For example, if you have a list **my_list = [1, 2, 3, 4, 5]** and you want to create a slice that includes the elements at index 1 through 3 (that is, the elements 2, 3, and 4), you would do the following:

```
my_slice = my_list[1:4]
```

This would create a new list, **my_slice**, that contains the elements **[2, 3, 4]**.

You can also omit the **start** or **end** index if you want to include all the elements from the beginning or end of the list, respectively. For example, if you want to create a **slice** that includes all the elements from the beginning of the list up to (but not including) the element at index 3, you can do the following:

```
my_slice = my_list[:3]
```

This would create a new list, **my_slice**, that contains the elements **[1, 2, 3]**.

Alternatively, if you want to create a **slice** that includes all the elements from the element at index 2 to the end of the **list**, you can do the following:

```
my_slice = my_list[2:]
```

This would create a new **list**, **my_slice**, that contains the elements **[3, 4, 5]**.

Overall, **slicing** a **list** in Python is a simple and powerful way to extract specific elements from a **list** and create a new list from those elements.

How do I remove duplicates from a list in Python?

To remove duplicates from a **list** in Python, you can use the set data type, which is a collection of unique elements. Here is an example of how you can use a set to remove duplicates from a list:

```
my_list = [1, 2, 3, 1, 2, 3, 4]



# Convert the list to a set, which will remove the duplicates

my_set = list(my_list)
```

```
# Convert the set back to a list

my_list = list(my_set)


# Print the resulting list

print(my_list)
```

```
> [1, 2, 3, 4]
```

Alternatively, you can use the **in** keyword and a for loop to iterate over the elements of the **list** and add each element to a new **list** if it is not already present in the new list. Here is an example of how you can do this:

```
my_list = [1, 2, 3, 1, 2, 3, 4]


# Create an empty list to store the unique elements

unique_list = []


# Iterate over the elements of the list

for element in my_list:

    # Check if the element is not already in the new list

    if element not in unique_list:

        # If it is not, add it to the new list

        unique_list.append(element)


# Print the resulting list
```

```
print(unique_list)
```

```
> [1, 2, 3, 4]
```

Both of these methods will produce the same result: a new **list** with the duplicate elements removed. The method you choose will depend on your preferences and the specific requirements of your application.

What are the map, filter, and reduce functions in Python?

In Python, the **map()**, **filter()**, and **reduce()** functions are built-in functions that allow you to apply a function to a sequence of elements and produce a new sequence. These functions are often used to perform operations on **lists**, **tuples**, and other sequences of elements.

The **map()** function applies a **function** to each element of a sequence, and returns a new sequence that contains the results of the function applied to each element. For example, you can use the **map()** function to square each element of a list of numbers:

```
# Define a list of numbers

numbers = [1, 2, 3, 4, 5]


# Use the map() function to apply a lambda function to each element of the list

# The lambda function takes an element, x, and returns its square


squares = map(lambda x: x**2, numbers)
```

```
# Print the squared numbers

print(list(squares))
```

```
> [1, 4, 9, 16, 25]
```

The **filter()** function applies a **function** to each element of a sequence, and returns a new sequence that contains only the elements for which the function returns **True**. For example, you can use the **filter()** function to select only the even numbers from a **list** of numbers:

```
# Define a list of numbers

numbers = [1, 2, 3, 4, 5]


# Use the filter() function to apply a lambda function to each element of the list

# The lambda function takes an element, x, and returns True if it is even, and False otherwise

evens = filter(lambda x: x % 2 == 0, numbers)


# Print the even numbers

print(list(evens))
```

```
> [2, 4]
```

The **reduce()** function applies a function to each element of a sequence, and returns a single result. The function is applied to the first two elements of the sequence, and the result is then used as the first argument for the next iteration, along with the third element of the sequence. This process continues until all elements have been processed, and a single result is produced. For example, you can use the **reduce()** function to calculate the sum of a list of numbers:

```
# Import the reduce() function from the functools module

from functools import reduce


# Define a list of numbers

numbers = [1, 2, 3, 4, 5]


# Use the reduce() function to apply a lambda function to the elements of the list

# The lambda function takes two arguments, x and y, and returns their sum

total = reduce(lambda x, y: x + y, numbers)


# Print the total

print(total)
```

```
> 15
```

How do I make a lambda in Python?

In Python, a **lambda function** is a single-line, anonymous function that is defined using the **lambda** keyword. **Lambda functions** are used to perform simple operations, and they are often used in conjunction with other built-in functions, such as **map()**, **filter()**, and **reduce()**.

Here is an example of how you can define a **lambda function** in Python:

```
# Define a lambda function that takes two arguments, x and y, and returns their sum

my_lambda = lambda x, y: x + y



# Use the lambda function

result = my_lambda(3, 4)



print(result)
```

```
> 7
```

In this example, the **my_lambda** function is defined using the **lambda** keyword, followed by the **function arguments**, a colon, and the function body. The function body is a single expression that specifies what the function should return. In this case, the function returns the sum of its two arguments, **x** and **y**.

Lambda functions are often used in combination with other built-in functions, such as **map()** and **filter()**. Here is an example of how you can use a **lambda function** with the **map()** function:

```
# Define a list of numbers

numbers = [1, 2, 3, 4, 5]



# Use the map() function to apply a lambda function to each element of the list

# The lambda function takes an element, x, and returns its square
```

```
squares = map(lambda x: x**2, numbers)


# Print the squared numbers

print(list(squares))
```

```
> [1, 4, 9, 16, 25]
```

In this example, the **map()** function applies the **lambda function** to each element of the numbers list. The lambda function takes an element, **x**, and returns its square. This results in a list of squared numbers.

In Python what is an iterable?

In Python, an **iterable** is an object that can be used in a **for loop**, or with other **functions** and constructs that expect an iterable. An iterable is an object that defines the **__iter__** method, which returns an iterator, or an object that defines the **__getitem__** method, which allows the object to be indexed like a list.

Iterables are an essential part of Python, and they are used to represent a sequence of values that can be **iterated** over in a **for loop**, or passed to functions and constructs that expect an iterable. Examples of iterables in Python include **lists**, **tuples**, strings, and **dictionaries**.

For example, in Python, you could use a **for loop** to iterate over a **list** of numbers like this:

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

for number in numbers:

    print(number)
```

In this code, the **numbers** list is defined with the values 1 through 5, and then a **for loop** is used to iterate over the elements of the list. The **number** variable is used to refer to each element of the list in turn, and the code inside the loop is executed once for each element. This code would print the numbers 1 through 5 to the screen, one number per line.

The **numbers** list is an **iterable**, because it defines the **__getitem__** method, which allows it to be indexed like a **list**. This means that the **numbers** list can be used in a **for loop**, or with other functions and constructs that expect an **iterable**.

Sets

In programming, what is a set?

In programming, a **set** is a **data structure** that stores a collection of unique elements. **Sets** are commonly used in many programming languages, including Python, to store data that needs to be accessed quickly and efficiently.

Sets are similar to other collection data types, such as **lists** and **tuples**, but they have some key differences. Unlike lists and tuples, sets only store unique elements, so you can't have duplicates in a set. Additionally, sets are unordered, which means that the elements in a set are not stored in any particular order. This makes sets faster and more efficient for certain operations, such as checking whether an element is in a set, or computing the intersection or union of two sets.

Here's an example of how you can create and use a set in Python:

```
# Define a set of numbers

numbers = {1, 2, 3, 4, 5}


# Add a new element to the set

numbers.add(6)


# Check if an element is in the set

if 4 in numbers:

    print("4 is in the set")


# Remove an element from the set

numbers.remove(3)


# Print the set to the console

print(numbers)
```

```
> 4 is in the set

> {1, 2, 4, 5, 6}
```

This code will create a set of numbers and then add and remove elements from the **set**. When you run it, it will print 4 is in the set and then {1, 2, 4, 5, 6} to the console.

As you can see, **sets** are a useful data structure for storing and working with collections of unique elements in your programs. You can use them to efficiently store and manipulate data in many different scenarios.

Tuples

In programming, what is a tuple?

In programming, a **tuple** is a **data structure** that consists of an ordered sequence of elements. Tuples are similar to **lists**, but they are **immutable**, which means that the elements of a **tuple** cannot be changed once the tuple is created. In Python, tuples are written with round brackets, and the elements of a tuple are separated by commas. For example, the following code creates a tuple containing the three strings "apple", "banana", and "cherry":

```
my_tuple = ("apple", "banana", "cherry")
```

Tuples have several advantages over **lists**. Because they are immutable, tuples are generally considered to be safer and more predictable than lists. This can be especially useful in situations where you need to ensure that the data in your tuple remains unchanged. Additionally, tuples are often faster and more efficient than lists, because their immutability allows the interpreter to make certain optimisations when working with them. Finally, tuples are useful for creating **data structures** that have a fixed number of elements, or that consist of multiple elements of different types. For example, you could use a tuple to store the coordinates of a point in space, or to represent a record with fixed fields, such as a person's name and age.

Dictionaries

In programming, what is a dictionary?

In programming, a **dictionary** is a **data structure** that is used to store a collection of **key-value pairs**. **Dictionaries** are an essential part of many programming languages, and they are often used to store and manipulate collections of data where each item has a unique key associated with it.

A **dictionary** is typically implemented as a hash table, which is a **data structure** that uses a hash function to map keys to their corresponding values efficiently. Each **key-value pair** in a **dictionary** is called an entry, and the keys in a dictionary are typically unique and **immutable**.

In most programming languages, **dictionaries** are unordered, which means that the order in which the entries are added to the dictionary is not preserved. This means that the entries in a dictionary cannot be accessed using an index like in a **list** or **array**, but must be accessed using the keys associated with the entries.

Dictionaries are an essential part of many programming languages, and they are often used to store and manipulate collections of data where each item has a unique key associated with it. Dictionaries are typically implemented as hash tables, and they allow entries to be accessed using the keys associated with the entries. Different programming languages have

How do I use dictionaries in Python?

To use dictionaries in Python, you can create a dictionary using the **dict** class, or you can use the curly braces {} to define a **dictionary** literal. Once you have created a **dictionary**, you can add entries to it by assigning values to keys, like this:

```
ages = {}

# create an empty dictionary
```

```
ages["Alice"] = 22

# add an entry with key "Alice" and value 22


ages["Bob"] = 25

# add an entry with key "Bob" and value 25


ages["Charlie"] = 31

# add an entry with key "Charlie" and value 31
```

In this code, an empty **dictionary** is created using the **dict** class, and then three entries are added to the dictionary using assignment statements. The keys in a dictionary must be unique and **immutable**, and the values can be of any type.

Once you have created and populated a **dictionary**, you can access the entries in the dictionary using the keys associated with the entries. For example, you could access the entry with **key** "Bob" like this:

```
age = ages["Bob"]

# age is now set to 25
```

In this code, the **ages["Bob"]** expression is used to access the entry in the **ages dictionary** with the key "Bob", and the corresponding value (25) is assigned to the **age** variable.

You can also use the **in** keyword to check if a key is in a **dictionary**, like this:

```
if "Bob" in ages:
    print("Bob is in the dictionary")
```

In this code, the **"Bob" in ages** expression is used to check if the key "Bob" is in the **ages** dictionary. Since the key "Bob" was added to the **dictionary** in the previous examples, this condition evaluates to true, and the code inside the **if statement** is executed. This code would print "Bob is in the dictionary" to the screen.

Chapter Review

- What is a data structure?
- What is a list?
- How is a list different from a tuple?
- How is it different from a set?
- What is a dictionary?
- What is the slice operator?
- What do map and reduce do? Filter?

Chapter 8
Object Oriented Programming

What is object-oriented programming?

Object-oriented programming (**OOP**) is a programming paradigm that is based on the concept of "**objects**", which can contain data and code that manipulates that data. OOP is designed to make it easy to create and maintain large and complex software systems, and it is used in many different programming languages.

In **OOP**, **objects** are created from templates called **classes**, which define the properties and behavior of the objects. Objects can inherit characteristics from their parent classes, and they can be **extended** and modified to create new objects.

OOP has several key features, including **encapsulation**, **polymorphism**, and **inheritance**. **Encapsulation** refers to the idea that the internal details of an object should be hidden from the outside world, and only the object's public interface should be visible. **Polymorphism** refers to the ability of objects to take on different forms, depending on the context in which they are used. **Inheritance** refers to the ability of objects to inherit characteristics from their parent classes.

Overall, **OOP** is a powerful programming paradigm that is used to create large and complex software systems. It allows for the modularization and organization of code, and it makes it easier to manage and maintain software programs.

In programming, what is a class?

In **object-oriented programming (OOP)**, a **class** is a template or blueprint that is used to create **objects**. It defines the properties and behavior of the objects that are created from it, and it provides a common structure for the objects to follow.

A **class** typically consists of a set of data fields (or attributes) and a set of **methods** (or **functions**) that operate on those data fields. The data fields define the characteristics of the **objects** that are created from the class, and the methods define the behavior of the objects.

Classes are an important part of **object-oriented programming**, and they are used to organize and structure code. They allow for the modularization of code, and they make it easier to create and maintain large and complex software systems.

How do you use classes in Python?

To use **classes** in Python, you first need to define the class using the **class** keyword, followed by the name of the class. You can then define the data fields and methods of the class, and you can use the **self** keyword to refer to the current object within the class methods.

Here is an example of how to define and use a simple class in Python:

```
# Define the class

class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hello(self):
        print("Hello, my name is " + self.name)

# Create an object from the class
p1 = Person("John", 30)

# Access the object's data fields
print(p1.name)
```

```
> John
```

```
John print(p1.age)
```

```
> 30
```

```
# Call the object's method
p1.say_hello()
```

```
> Hello, my name is John
```

In this example, the **Person** class is defined with two data fields, **name** and **age**, and one method, **say_hello()**. An object is then created from the **Person class**, and its data fields and methods are accessed and called.

This example shows the basic steps for defining and using **classes** in Python. You can use classes to organize and structure your code, and you can create as many objects as you need from a single class.

What are public and private attributes in Python?

In Python, the terms "**public**" and "**private**" refer to the **visibility** of **class attributes**. **Public attributes** are attributes that can be freely accessed from anywhere, while **private attributes** are attributes that are only accessible from within the class itself.

In Python, there is no explicit syntax for defining public and private **attributes**. Instead, the convention is to use a leading underscore (**_**) character for private attributes and no leading underscore for public attributes. For example:

```
class MyClass:

    def __init__(self):

        self._private_attribute = 1

        self.public_attribute = 2
```

In this example, the **_private_attribute** attribute is a private attribute, because it has a leading underscore in its name. The **public_attribute** attribute is a public attribute, because it does not have a leading underscore in its name.

While this convention is not enforced by the Python interpreter, it is a widely-followed convention that helps to prevent accidental access to private attributes from outside the class.

It's important to note that, even though **private attributes** are not directly accessible from outside the class, they can still be accessed through **methods** within the class. For example:

```
class MyClass:
    def __init__(self):
        self._private_attribute = 1


    def get_private_attribute(self):
        return self._private_attribute
```

In this example, the **_private_attribute** attribute is private, but it can still be accessed through the **get_private_attribute** method. This is because the method is part of the class and has access to the **self** instance, which includes the private attribute.

In general, it is considered good practice to use **private attributes** to encapsulate implementation details of a class, and to provide public methods for accessing and modifying the state of the **object**. This helps to prevent accidental modification of the **object's** state and makes the class easier to use and maintain.

What is "self" in Python?

In Python, the keyword **self** is used to represent the instance of an **object** in a **class** method. It is automatically passed as the first argument to a class method when the method is called, but you don't have to specify it when calling the method.

Here is an example of how **self** is used in a class method:

```
class MyClass:
    def my_method(self):
```

```
    # Code for the method goes here

    # You can access instance attributes and methods

    # using the self keyword

    self.attribute = 1

    self.other_method()
```

In this example, the **my_method** method is a class method that takes the **self** argument. This **argument** is automatically passed to the method when it is called, and it refers to the instance of the class on which the method is called.

In the body of the method, the **self** keyword is used to access the instance attributes and methods of the object. For example, the **self.attribute** line sets the value of the **attribute** attribute for the instance, and the **self.other_method()** line calls the **other_method** method for the instance.

It's important to note that you don't have to specify the **self** argument when calling a class method. For example, if you have an instance of **MyClass** called **my_object**, you can call the **my_method** method like this:

```
my_object.my_method()
```

In this case, the **self** argument will be automatically passed to the **my_method** method, and it will refer to the **my_object** instance.

Does Python have interface blueprints like other languages?

No, Python does not have a traditional **interface** construct like other languages, such as Java and C#. In Python, you can use the concept of an

abstract base class to define the methods and attributes that a subclass should implement, but this is not the same as an interface.

An **abstract base class** in Python is a class that contains one or more **abstract** methods, which are methods that are declared but not implemented. When you define a **subclass** of an **abstract base class**, the subclass must implement all of the abstract methods from the base class, or it will be an abstract class itself.

Here is an example of how to define and use an abstract base class in Python:

```
from abc import ABC, abstractmethod


# Define an abstract base class

class Shape(ABC):

    @abstractmethod

    def area(self):

        pass


# Define a subclass of the abstract base class class Circle(Shape):

    def __init__(self, radius):

        self.radius = radius


    def area(self):

        return 3.14 * self.radius ** 2
```

```
# Create an instance of the Circle class

circle = Circle(10)


# Call the area() method

print(circle.area())


> 314.0
```

In this example, the **Shape** class is defined as an **abstract base class**, and it contains an abstract method called **area()**. The **Circle** class is defined as a **subclass** of **Shape**, and it implements the **area()** method to calculate the area of a circle.

When you create an instance of the **Circle** class and call the **area()** method, the **area()** method from the **Circle** class is called, rather than the **abstract** method from the **Shape** class. This allows you to enforce a common interface for subclasses of the **Shape** class, without using a traditional **interface** construct.

In Python, what is a dunder method?

In Python, a **dunder method** is a method with a double underscore (**dunder**) in its name. These methods are also known as magic methods, and they are used to define the behavior of certain operations in Python.

Dunder methods are special methods that are defined within a **class**, and they have a specific signature and behavior. They are called automatically by Python when certain operations are performed on an object, and they allow you to customize the behavior of those operations.

Here is an example of a **dunder method** in Python:

```
class Person:

    def __init__(self, name, age):

        self.name = name

        self.age = age


    def __str__(self):

        return "Person: " + self.name + ", " + str(self.age)
```

In this example, the **Person** class has a **__str__()** dunder method. The **__str__()** method is called automatically when the **str()** function is called on an object, and it defines how the object should be represented as a string. In this case, the **__str__()** method returns a string that includes the object's name and age.

Here is an example of how to use the **__str__()** dunder method using the above **class** definition:

```
# Create an object from the class

p1 = Person("John", 30)


# Call the str() function on the object

print(str(p1))

# Output: Person: John, 30
```

In this example, the **str()** function is called on the **p1** object, and the **__str__()** method is automatically called. The **__str__()** method returns a string that includes the object's name and age, and that string is printed by the **print()** function.

Dunder methods are special methods in Python that have double underscores in their names. They are called automatically by Python when certain operations are performed on an object, and they allow you to customize the behavior of those operations.

What does the __init__ method do in Python?

In Python, the **__init__()** method is a special method that is called when an object is created from a class. It is used to initialize the data fields of the object, and it is automatically called when an object is created.

The **__init__()** method is defined within the class definition, and it has a specific signature that includes the **self** keyword. Here is an example of a class that has an **__init__()** method:

```
class Person:
    def __init__(self, name, age):
        self.name = name
        self.age = age
    def say_hello(self):
        print("Hello, my name is " + self.name)
```

In this example, the **__init__()** method is defined with two parameters, **name** and **age**, which are used to initialize the data fields of the object. The **self** keyword is used to refer to the current object, and it is used to set the values of the **name** and **age** data fields.

When an object is created from the **Person** class, the **__init__()** method is automatically called, and it sets the initial values of the object's data fields. Here is an example of how to create an object from the **Person** class and call the **__init__()** method:

```
# Create an object from the class

p1 = Person("Ben Kenobi", 30)


# The __init__() method is automatically called,

# and the object's data fields are initialized

print(p1.name)


> Ben Kenobi


print(p1.age)


> 30
```

In this example, the **__init__()** method is automatically called when the **p1** object is created, and it sets the initial values of the **name** and **age** data fields.

Can I docstring Python classes?

Yes, you can use **docstrings** to document Python **classes**. Docstrings are string literals that appear as the first statement in a Python **function**, method, or class, and they are used to provide a brief description of the function, method, or class.

To create a **docstring** for a Python class, you can add a string literal as the first statement in the class definition. The string can span multiple lines, and it should provide a brief description of the class and its purpose. Here is an example of a class with a docstring:

```
class Person:
    """This class represents a person, with a name and an age"""

    def __init__(self, name, age):
        self.name = name
        self.age = age

    def say_hello(self):
        print("Hello, my name is " + self.name)
```

In this example, the **Person** class has a **docstring** that describes what the class is for and what it does. The docstring is enclosed in triple quotes (`"""`), and it appears as the first statement in the **class** definition.

You can access the docstring of a **class** using the **__doc__** attribute of the class. Here is an example of how to access the docstring of the **Person** class:

```
# Access the docstring of the class
print(Person.__doc__)

# Output: This class represents a person, with a name and an age
```

In this example, the **__doc__** attribute of the **Person class** is accessed, and the class's **docstring** is printed.

Docstrings are an important part of Python's documentation conventions.

In programming, what is class inheritance?

In **object-oriented programming (OOP)**, **class inheritance** is a mechanism that allows one **class** (the child class) to inherit the characteristics of another class (the **parent** class). The **child class** can use the **attributes** and **methods** of the **parent class**, and it can override or extend those attributes and methods to provide its own behavior.

Class inheritance is a powerful feature of **OOP** that allows for code reuse and modularization. It allows you to create a hierarchy of classes, where **child classes** inherit the attributes and methods of their parent classes. This can make it easier to manage and maintain large and complex software systems.

Here is an example of **class inheritance** in Python:

```
# Define the parent class

class Animal:

    def __init__(self, name):

        self.name = name



    def make_sound(self):

        print("Grrrr")
```

```
# Define the child class

class Dog(Animal):

    def __init__(self, name, breed):

        # Call the parent’s __init__() method

        super().__init__(name)

        self.breed = breed


def make_sound(self):

    # Override the parent’s make_sound() method

    print("Woof!")



# Create an object from the child class

dog1 = Dog("Buddy", "Labrador")


# Call the child’s make_sound() method

dog1.print()
```

```
> Woof!
```

In this example, the **Animal** class is defined with a **__init__()** method and a **make_sound()** method. The **Dog** class is defined as a child of the **Animal** class, and it inherits the **__init__()** and **make_sound()** methods from the parent class.

The **Dog** class **overrides** the **make_sound()** method of the parent class, and it defines its own behavior for this method. It also calls the **__init__()** method of the parent class using the **super()** function, and it passes the **name** parameter to that method.

Class inheritance is a powerful feature of **OOP** that allows one class to inherit the characteristics of another class. It allows for code reuse and modularization, and it makes it easier to manage and maintain large and complex software systems.

In programming, what is a static method?

In programming, a **static method** is a method that belongs to a class rather than an instance of the class. This means that the method is associated with the **class** itself, rather than with any particular instance of the class.

A **static method** can be called on the class itself, rather than on an instance of the class. This is different from a regular method, which can only be called on an **instance** of a class.

Static methods are often used as **factory methods**, which are methods that create and return new instances of a class. They are also often used to define utility functions that are related to a **class**, but don't require access to instance-specific state.

It's important to note that **static methods** do not have access to the instance of the class. This means that they can't modify object state, but they can be called on the **class** itself, rather than on an instance of the class.

How do I make a static method in Python?

To create a static method in Python, you can use the **@staticmethod** decorator. Here is an example of how to do this:

```
class MyClass:

    @staticmethod
    def my_static_method():

        # Code here!

        pass
```

To call the **static method**, you would use the following syntax:

```
MyClass.my_static_method()
```

It's important to note that **static methods** are different from regular methods in that they don't have access to the instance of the class, and therefore they don't have access to **self**. This means that they can't modify object state, but they can be called on the class itself, rather than on an **instance** of the class.

Here is an example of how you might use a **static method** to create a **factory method**, which is a common use case for **static methods** in Python:

```
class MyClass:

    @staticmethod

    def create_object(x, y):

        # Create and return an object using x and y

        return MyClass(x, y)
```

In this example, the **create_object** method is a **static method** that can be called on the **MyClass** class itself, rather than on an instance of the class. This allows you to create new instances of **MyClass** without having to first create an instance of the class.

Chapter Review

- What is a class?

- What is the difference between an object and a class?
- What are the key important parts of OOP?
- How does inheritance work?
- What’s a static method?

Chapter 9
Errors When Things Go Wrong

What is an exception in Python?

An **exception** is an error that occurs during the execution of a Python program. Exceptions can be raised when a program tries to perform an invalid operation, such as dividing a number by zero or trying to access an element in a **list** using an index that is **out of bounds**. When an exception occurs, it interrupts the normal flow of the program and causes it to terminate, unless the exception is handled by the program.

In Python, **exceptions** are represented by objects that are instances of the Exception class or one of its **subclasses**. When an exception is raised, it creates an Exception object that contains information about the error, such as the type of the exception and a detailed error message. This Exception object can be caught and handled by the program, allowing the program to continue executing even in the presence of an error.

Exceptions are an important mechanism in Python for handling errors and allowing a program to gracefully recover from unexpected situations. By using **try** and **except** statements, you can write Python programs that are able to handle exceptions and continue running even in the face of errors.

What is "try except" in Python?

"**Try catch**" is a common programming construct that is used to handle errors or **exceptions** that may occur in a program. In Python, the "**try**" and "**except**" keywords are used to implement this construct.

The "**try**" keyword is used to enclose a block of code that may throw an error or **exception**. If an error occurs within this block of code, the interpreter will immediately stop executing the code and move on to the "**except**" block, which is used to handle the error.

Here is an example of how the "**try**" and "**except**" keywords are used in Python:

```
try:
# This code may throw an error or exception
    result = some_function()
except:
# This code will be ran if there’s an error
    print("An error occurred!”)
```

In this example, the code inside the "**try**" block calls the **some_function()** function, which may throw an **exception**. If an **exception** is thrown, the interpreter will immediately stop executing the "**try**" block and move on to the "**except**" block, which is used to handle the error. In the "**except**" block, the error message is printed to the screen using the **print()** function.

The "**try catch**" construct is useful because it allows you to gracefully handle errors or **exceptions** in your code, without the need to abruptly stop the program or crash the interpreter. This can help to improve the reliability and stability of your program, and can make it easier to debug and troubleshoot any problems that may occur.

What is code debugging?

Code debugging is the process of identifying and fixing errors or bugs in a computer program. When a program contains errors, it will not run correctly, and may produce incorrect or unexpected results, or crash altogether. **Debugging** is an essential part of the software development process, and it involves finding and fixing these errors so that the program can run correctly.

There are many different techniques and tools that can be used for **debugging**, depending on the specific needs and goals of the project. Some common techniques include adding print statements or other debugging messages to the code, using a **debugger** to step through the code line by line, or using automated testing tools to find and isolate errors.

The goal of **debugging** is to identify and fix the errors in a program as quickly and efficiently as possible. This can require a combination of technical skills, problem-solving abilities, and a thorough understanding of the program's code and its intended behavior. Debugging can be a challenging and time-consuming process, but it is an essential part of creating high-quality software.

How do I debug code in Python?

To **debug** code in Python, you can use a combination of techniques, including adding print statements, using a **debugger**, and running **automated tests**.

One of the simplest and most effective ways to debug code in Python is to use the **print()** function to print out the values of variables or expressions at different points in your code. This can help you to understand the state of your program at different stages of execution, and can make it easier to identify where errors are occurring and what might be causing them.

Another useful tool for debugging Python code is the **pdb** module, which provides a command-line **debugger** that allows you to step through your code line by line, inspect the values of variables, and control the execution of the program. This can be especially helpful for understanding the flow of control in your code, and for identifying errors that are difficult to detect using print statements alone.

In addition to these techniques, you can also use automated testing tools, such as **unittest** or **pytest**, to write and run **test cases** for your code. This can help you to identify and fix errors more quickly and systematically, and can also help to ensure that your code continues to work correctly as you make changes to it.

Is there a visual debugger for Python?

Yes, there are several **visual debuggers** for Python. A visual debugger is a type of debugging tool that provides a **graphical user interface** (**GUI**) for stepping through and inspecting the code of a program. This can make it easier to understand the flow of control in your code, and to identify and fix errors.

One popular visual debugger for Python is the **pdb** module, which is included in the standard library. The **pdb** debugger provides a command-line interface (**CLI**) for stepping through and inspecting your code, and it also includes a **pdb.run()** function that allows you to run the debugger in a separate window. This can make it easier to see the code and the values of variables as you step through your program.

Another popular visual debugger for Python is **ipdb**, which is a fork of **pdb** that adds additional features and improvements. **ipdb** includes support for tab-completion, syntax highlighting, and inline debugging, which can make it even easier to use and more powerful than **pdb**.

In addition to these options, there are also several third-party visual debuggers for Python, such as **PyCharm** and **Wing IDE**, which provide more advanced features and a more user-friendly interface.

What is a programming syntax error?

A **syntax error** is a type of error that occurs when the syntax of a programming language is not followed correctly. In other words, it is an error in the structure of the program's code that makes it impossible for the interpreter to parse and execute the program.

Syntax errors are typically caused by missing or mismatched punctuation, incorrect indentation, or incorrect use of reserved words or operators. For example, a **syntax error** might occur if you forget to close a quotation mark, or if you use a variable name that is not allowed in your programming language.

Syntax errors are usually detected by the interpreter when you try to run your program. The interpreter will display an error message that indicates where the error occurred and what the problem is. For example, the error message might say something like "*SyntaxError: unexpected end of file*", or "*SyntaxError: invalid syntax*".

It is important to fix **syntax errors** in your code, as they will prevent your program from running correctly. To fix a **syntax error,** you need to carefully review your code and look for any mistakes in the syntax. Once you have identified the error, you can make the necessary corrections and run your program again to see if the error has been resolved.

What is a programming run-time error?

A **runtime error**, also known as a **runtime exception**, is a type of error that occurs when a program is running. Unlike a **syntax error**, which is detected by the interpreter when the program is first compiled, a runtime error occurs during the execution of the program, when the interpreter is trying to execute a specific line of code.

There are many different types of **runtime errors**, and they can be caused by a wide range of issues. Some common examples of **runtime errors** include trying to access an element of an **array** or **list** that does not

exist, dividing a number by zero, or calling a **function** with the wrong number of **arguments**.

Runtime errors can be difficult to **debug**, because they often do not produce an error message until the program is already running. When a **runtime error** occurs, the interpreter will typically stop executing the program and display an error message that indicates what the problem is and where it occurred.

To fix a **runtime error**, you need to carefully review the code that caused the error, and try to understand what went wrong. You may need to add additional **debugging** statements or test cases to help identify the problem, and then make the necessary changes to your code to fix the error. Once you have fixed the error, you can run your program again to see if it works correctly.

Chapter Review

- What is an exception?
- What is "try/except"? How do you use it?
- What is a debugger?
- What is a syntax error?
- What is a runtime error?

Chapter 10

Math and Charts

How do I do basic math in Python?

To do basic math in Python, you can use the built-in mathematical operators. For example, to add two numbers a and b you would use the + operator like this:

```
result = a + b
```

Other mathematical operators in Python include - for subtraction, * for multiplication, / for division, and % for modulo (remainder). For example, to compute the remainder of a divided by b you would write:

```
remainder = a % b
```

Python also has built-in functions for more complex mathematical operations, such as **pow** for exponentiation, **sqrt** for square root, and **log** for logarithms. For example, to compute the square root of a you would write:

```
sqrt_a = sqrt(a)
```

You can also use the **math module** to access a wide range of mathematical functions. To use the **math module**, you need to **import** it first, like this:

```
import math
```

Once you have imported the **math module**, you can access its functions using the dot notation, like this:

```
sqrt_a = math.sqrt(a)
```

To find the minimum and maximum of two numbers in Python, you can use the **min** and **max** functions, respectively. For example, to find the minimum of **a** and **b**, you would write:

```
minimum = min(a, b)
```

To find the maximum of **a** and **b**, you would write:

```
maximum = max(a, b)
```

The **min** and **max** functions can also be used with more than two numbers. For example, to find the minimum of three numbers **a**, **b**, and **c**, you would write:

```
minimum = min(a, b, c)
```

To find the maximum of three numbers **a**, **b**, and **c**, you would write:

```
maximum = max(a, b, c)
```

How can I get a random number in Python?

To get a **random number** in Python, you can use the **random module** and its function **randint()**. Here's an example:

```
import random
```

```
# Get a random integer between 1 and 10 (inclusive) random_number =
random.randint(1, 10)


print(random_number)
```

```
> (A random number between 1 and 10)
```

This will print a **random integer** between 1 and 10 (inclusive) to the console. You can adjust the lower and upper bounds of the range by changing the arguments to **randint()**. For example, if you want a random number between 0 and 100, you would call **randint(0, 100)**.

How do I round numbers in Python?

To round numbers in Python, you can use the **round()** function. This function takes a number as input and returns the number rounded to the nearest integer. Here's an example:

```
# Define a number to round

number = 3.14159


# Round the number to the nearest integer

rounded_number = round(number)

print(rounded_number)
```

This code will print *3* to the console. As you can see, the **round()** function rounds the input number down to the nearest integer.

You can also use the **round()** function to round a number to a specific number of decimal places. To do this, you need to pass an additional argument to the **round()** function specifying the number of decimal places you want. Here's an example:

```
# Define a number to round

number = 3.14159



# Round the number to two decimal places

rounded_number = round(number, 2)

print(rounded_number)
```

This code will print 3.14 to the console. As you can see, the **round()** function rounds the input number to two decimal places, as specified by the second argument.

How do I calculate exponents in Python?

To calculate exponents in Python, you can use the ****** operator. This operator raises the number on the left side of the operator to the power of the number on the right side. For example, to calculate 2 to the power of 3, you would write:

```
result = 2 ** 3
```

This would return 8, since 2 to the power of 3 is equal to 8.

Alternatively, you can use the **pow** function from the **math module** to calculate exponents. To use the **pow** function, you need to import the math module first, like this:

```
import math
```

Once you have imported the **math module**, you can use the **pow** function to calculate exponents like this:

```
result = math.pow(2, 3)
```

This would also return 8.

Note that the **pow** function returns a floating-point number, even if the result is an integer (whole number). So if you want to get an integer result, you need to convert the result to an integer using the **int** function, like this:

```
result = int(math.pow(2, 3))
```

This would return 8 as an integer, rather than as a floating-point number.

How do I calculate the radius and area of a circle in Python?

To calculate the radius and area of a circle in Python, you can use the **math module** to access the value of pi and the **pow()** function to calculate the radius and area.

Here is an example of how you could calculate the radius and area of a circle in Python:

```
import math
```

```
# Define the radius of the circle

r = 5


# Calculate the area of the circle using the formula

A = pi * r^2

A = math.pi * pow(r, 2)
```

```
# Print the radius and area of the circle

print("The radius of the circle is", r)

print("The area of the circle is", A)
```

In this example, the **pow()** function is used to calculate the radius raised to the second power, which is then multiplied by the value of pi from the **math module** to calculate the area of the circle.

How do I use trigonometry functions in Python?

In Python, the **math module** provides functions for working with **trigonometric functions**. For example, to use the sine function, you would first need to import the math module, and then call the **sin()** function like this:

```
import math x = math.sin(1)
```

The **sin()** function takes an angle as its argument, specified in radians, and returns the sine of that angle. You can use the other trigonometric functions in the same way, by calling the corresponding function from the math module. For example, to use the cosine function, you would call the **cos()** function like this:

```
import math x = math.cos(1)
```

Similarly, you can use the tangent function by calling the **tan()** function, and the inverse sine, cosine, and tangent functions by calling the **asin()**, **acos()**, and **atan()** functions, respectively.

Here is an example that demonstrates how to use these functions to calculate the sine, cosine, and tangent of an angle:

```
import math


# Calculate the sine of the angle

angle = 1

sin = math.sin(angle)


# Calculate the cosine of the angle

cos = math.cos(angle)


# Calculate the tangent of the angle

tan = math.tan(angle)

 # Print the results

print("The sine of the angle is", sin)
print("The cosine of the angle is", cos)

print("The tangent of the angle is", tan)
```

How do I do calculus in Python?

Python has several libraries that provide functions for working with calculus. The **sympy** library is one option that allows you to perform symbolic calculus in Python.

To use the **sympy** library, you first need to install it by running "*pip install sympy*" in your terminal. Once you have installed the library, you can use it in your Python code by importing it and using the provided functions.

Here is an example that shows how to use the **sympy** library to calculate the derivative of a function:

```
import sympy


# The function you want a derivative of

f = x**2 + 2*x + 1


# Use the diff() function to calculate the derivative

df = sympy.diff(f, x)


# Print the result

print("The derivative of f(x) is", df)
```

In this example, the **diff()** function is used to calculate the derivative of the function f(x) with respect to the variable x. The **diff()** function takes two arguments: the function that you want to take the derivative of, and the variable with respect to which you want to take the derivative.

In addition to the **diff()** function, the **sympy** library also provides functions for calculating integrals, limits, and other common operations in calculus. You can learn more about the **sympy** library and its features by reading the documentation at https://www.sympy.org/.

Charting with Matplotlib

What is Matplotlib?

Matplotlib is a popular Python library for creating visualizations of data. It provides a variety of functions and classes that make it easy to plot data in a wide range of formats, including line plots, scatter plots, bar charts, and histograms.

Matplotlib is often used in conjunction with other scientific computing libraries, such as **NumPy** and **Pandas**, to create powerful visualizations of large and complex datasets. It is also a powerful tool for creating publication-quality figures for scientific papers, reports, and other documents.

To use **Matplotlib** in a Python program, you will first need to **import** the library. You can do this using the following code:

```
import matplotlib.pyplot as plt
```

This imports the **pyplot submodule** from the **matplotlib** library and gives it the **alias** *plt*. You can then use the functions and classes in the **pyplot submodule** to create and customize your plots.

Here's an example of how you can use **Matplotlib** to create a simple line plot:

```
import matplotlib.pyplot as plt


# Define some data

x = [1, 2, 3, 4, 5]

y = [1, 4, 9, 16, 25]


# Use the plot() function to create a line plot

plt.plot(x, y)


# Show the plot

plt.show()
```

This code will create a line plot with the data **x** and **y** and display it to the screen. You can customize the plot in many different ways, such as changing the line style, adding labels and titles, and changing the axis limits.

For more information on how to use **Matplotlib** and all the different features it provides, please check out the **Matplotlib** documentation.

How do I make a scatter plot with Matplotlib?

To make a scatter plot with **Matplotlib**, you can use the **scatter()** method from the **pyplot** module. The **scatter()** method takes the x and y values of the points to plot as arguments, and it allows you to specify various attributes, such as the marker style, color, and size, to customize the appearance of the scatter plot.

Here is an example of how to make a scatter plot with **Matplotlib**:

```
import matplotlib.pyplot as plt


# Define the x and y values of the points to plot

x = [1, 2, 3, 4, 5]

y = [1, 4, 9, 16, 25]


# Create a scatter plot

plt.scatter(x, y, marker="x", color="red")

 # Add a title and labels to the axes

plt.title("Squares")

plt.xlabel("x")

plt.ylabel("y")


# Show the plot

plt.show()
```

In this example, the **scatter()** method is used to create a scatter plot of the x and y values, using a red "x" marker for the points. The **title()**, **xlabel()**, and

ylabel() methods are then used to add a title and labels to the axes of the plot. Finally, the **show()** method is used to display the plot.

How can I make a histogram with Matplotlib?

To make a histogram with **Matplotlib**, you can use the **hist()** method from the **pyplot** module. The **hist()** method takes the data to plot as an argument, and it allows you to specify various attributes, such as the number of bins and the range of values to plot, to customize the appearance of the histogram.

Here is an example of how to make a histogram with **Matplotlib**:

```
import matplotlib.pyplot as plt


# Define the data to plot

data = [1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4]


# Create a histogram

plt.hist(data, bins=4, range=(1, 4), edgecolor="black")


# Add a title and labels to the axes

plt.title("Histogram")

plt.xlabel("x")

plt.ylabel("y")


# Show the plot
```

```
plt.show()
```

In this example, the **hist()** method is used to create a histogram of the data, using 4 bins and a range of values from 1 to 4. The **title()**, **xlabel()**, and **ylabel()** methods are then used to add a title and labels to the axes of the plot. Finally, the **show()** method is used to display the plot.
Note that the **hist()** method automatically calculates the bin edges and frequencies for the data, and it plots the histogram using the specified attributes. You can also use the **hist()** method to plot multiple

How can I make a bar chart with Matplotlib?

To make a bar chart with **Matplotlib**, you can use the **bar()** method from the **pyplot** module. The **bar()** method takes the x and y values of the bars to plot as arguments, and it allows you to specify various attributes, such as the bar width and color, to customize the appearance of the bar chart.

Here is an example of how to make a bar chart with Matplotlib:

```
import matplotlib.pyplot as plt



# Define the x and y values of the bars to plot

x = [1, 2, 3, 4, 5]

y = [1, 4, 9, 16, 25]




# Create a bar chart

plt.bar(x, y, width=0.5, color="red")
```

```
# Add a title and labels to the axes

plt.title("Squares")

plt.xlabel("x")

plt.ylabel("y")



# Show the plot

plt.show()
```

In this example, the **bar()** method is used to create a bar chart of the x and y values, using a bar width of 0.5 and a red color for the bars. The **title()**, **xlabel()**, and **ylabel()** methods are then used to add a title and labels to the axes of the plot. Finally, the **show()** method is used to display the plot.

How can I make a Pie chart with Matplotlib?

To make a pie chart with **Matplotlib**, you can use the **pie()** method from the **pyplot** module. The **pie()** method takes the data to plot as an argument, and it allows you to specify various attributes, such as the labels and colors of the slices, to customize the appearance of the pie chart.

Here is an example of how to make a pie chart with **Matplotlib**:

```
import matplotlib.pyplot as plt



# Define the data to plot

data = [1, 1, 2, 2, 2, 3, 3, 3, 3, 4, 4, 4, 4, 4]



# Create a pie chart
```

```
plt.pie(data, labels=["1", "2", "3", "4"], colors=["red", "orange", "yellow", "green"])


# Add a title

plt.title("Pie Chart")


# Show the plot

plt.show()
```

What are some advanced uses of Matplotlib?

Matplotlib is a powerful library for creating visualizations of data, and there are many advanced uses for the library. Here are a few examples of advanced uses for **Matplotlib**:

- Creating custom plot types: **Matplotlib** provides a wide range of built-in plot types, such as line plots, scatter plots, and bar charts, but you can also create your own custom plot types using **Matplotlib's object-oriented API**. This allows you to create specialized plots that are tailored to your specific data and needs.
- Creating 3D plots: **Matplotlib** includes functions and classes for creating 3D plots of data. You can use these functions to create 3D scatter plots, surface plots, and contour plots, among other types of 3D plots. This can be useful for visualizing data in three dimensions, such as scientific or engineering data.
- Creating animations: **Matplotlib** includes a **submodule** called **animation** that allows you to create animations of your data. You can use this submodule to create videos or GIFs that show how your data changes over time, which can be a powerful way to convey information and highlight trends in your data.
- Creating interactive plots: **Matplotlib** can be used to create interactive plots that allow the user to explore and manipulate the

data. For example, you can use **Matplotlib's** event handling functions to create plots that respond to mouse clicks or keyboard input, or you can use the widgets submodule to create plots with interactive controls, such as sliders and buttons. This can be a powerful way to create interactive visualizations of your data that allow users to explore and understand it in more depth.

Pyplot

What is Pyplot?

Pyplot is a submodule of the **Matplotlib** library in Python, which provides tools for creating visualizations of data in the form of plots and charts. The **pyplot** module is commonly used to create line plots, scatter plots, histograms, bar charts, and other types of plots, and it provides functions for customizing the appearance of the plots and for adding annotations, such as titles and labels.

The **pyplot** module is commonly imported with the alias *plt*, which is used as a namespace for the functions and methods provided by the **pyplot** module. Here is an example of how to import the **pyplot** module and use it to create a simple line plot:

```
import matplotlib.pyplot as plt


# Define the x and y values of the points to plot

x = [1, 2, 3, 4, 5]

y = [1, 4, 9, 16, 25]


# Create a line plot
```

```
plt.plot(x, y)


# Add a title and labels to the axes
plt.title("Squares")
plt.xlabel("x")
plt.ylabel("y")


# Show the plot
plt.show()
```

In this example, the **plot()** method from the **pyplot** module is used to create a line plot of the x and y values. The **title()**, **xlabel()**, and **ylabel()** methods are then used to add a title and labels to the axes of the plot. Finally, the **show()** method is used to display the plot.

Chapter Review

- How do you do basic math in Python?
- What are some of the advanced math operators?
- How do you do trig functions?
- How do you do calculus functions?
- What is Matplotlib?
- What type of plots can you make with Pyplot?

Chapter 11
Dates and Times

How do I work with Dates in Python?

In Python, the **datetime module** provides classes for working with dates and times. To use the **datetime module,** you first need to import it, like this:

```
import datetime
```

Once the **datetime module** is imported, you can use its classes to create date and time objects, which you can then manipulate and format in various ways.

Here are some examples of how you can work with dates and times in Python:

- To create a date object representing the current date, you can use the **date.today()** method, like this:

```
import datetime

today = datetime.date.today()
```

- To create a date object representing a specific date, you can use the **datetime.date()** constructor, which takes the year, month, and day as arguments, like this:

```
import datetime

my_birthday = datetime.date(1997, 2, 3)
```

- To create a time object representing the current time, you can use the **datetime.time()** constructor with no arguments, like this:

```
import datetime

now = datetime.time()
```

- To create a time object representing a specific time, you can use the **datetime.time()** constructor with the hour, minute, second, and microsecond as arguments, like this:

```
import datetime

lunch_time = datetime.time(12, 30, 0)
```

- To create a datetime object representing the current date and time, you can use the **datetime.datetime.now()** method, like this:

```
import datetime

now = datetime.datetime.now()
```

- To create a datetime object representing a specific date and time, you can use the **datetime.datetime()** constructor, which takes the year, month, day, hour, minute, second, and microsecond as arguments, like this:

```
import datetime

my_birthday = datetime.datetime(1997, 2, 3, 12, 30, 0)
```

Once you have created date and time objects, you can use various methods to manipulate and format them in different ways. For example, you can use the **strftime()** method to convert a date or **datetime** object to a string, using a specific format string, like this:

```
import datetime


today = datetime.date.today()

now = datetime.datetime.now()


# format the date as a short date (e.g. "02/03/97")

date_str = today.strftime("%m/%d/%y")


# format the datetime as a long date and time (e.g. "February 3, 1997 12:30:00 PM")

datetime_str = now.strftime("%B %d, %Y %I:%M:%S %p")
```

For a full list of format codes that you can use with the **strftime()** method, see the Python documentation.

How do I work with timezones in Python?

To work with timezones in Python, you can use the **pytz** library. The **pytz** library provides functions and classes for working with timezones and handling timezone-aware datetimes.

To use the **pytz** library, you first need to install it using the **pip** package manager. To do this, you can use the following command:

```
pip install pytz
```

Once **pytz** is installed, you can import it into your Python scripts and use its functions and classes. To do this, you can use the import statement to import the **pytz** library, and then use the **pytz** namespace to access its functions and classes.

Here is an example of how to use the **pytz** library to work with timezones in Python:

```
# import the pytz library and the datetime class

import pytz from datetime

import datetime


# create a timezone object for the Pacific time zone

pacific = pytz.timezone("US/Pacific")


# get the current time in the Pacific time zone

now = datetime.now(pacific)


# print the current time in the Pacific time zone

print(now)
```

How do I get the current time in Python?

To get the current time in Python, you can use the **datetime module** and the **datetime.now()** method. The **datetime module** provides classes and functions for working with dates and times, and the **datetime.now()** method returns the current date and time as a datetime object.

Here is an example of how to use the **datetime.now()** method to get the current time in Python:

```
from datetime import datetime

# get the current time

now = datetime.now()



# print the current time

print(now)
```

In this example, the **datetime.now()** method is used to get the current time, and the datetime object is printed to the console.

By default, the **datetime.now()** method returns the current time in the local timezone. If you want to get the current time in a specific timezone, you can use the **pytz** library and the **datetime.now()** method's **tzinfo** argument.

How do I get the current date in Python?

To get the current date in Python, you can use the **datetime module** and the **datetime.today()** method.

Here is an example of how to use the **datetime.today()** method to get the current date in Python:

```
from datetime import datetime


# get the current date

today = datetime.today()



# print the current date

print(today)
```

In Python what is the strftime method?

The **strftime** method is a method in the datetime module in Python that allows you to specify a format for a date/time and return it as a string. For example, you could use it to format a date/time in a specific way, such as *"%Y-%m-%d %H:%M:%S"* to get a string like *"2022-12-10 12:30:45"*. Here, the %Y indicates the year as a four-digit number, the %m indicates the month as a two-digit number, and so on.

How do I use the timedelta class in Python?

The **timedelta** class is part of the **datetime module** in Python, and it represents a duration of time rather than a specific date or time. You can use timedelta objects to perform basic arithmetic on date and time objects, such as adding or subtracting time intervals.

To use the **timedelta** class, you first need to import the datetime module, like this:

```
import datetime
```

Once the **datetime module** is imported, you can create a timedelta object by calling the **timedelta()** constructor and passing in the number of days, seconds, and microseconds that the duration should represent, like this:

```
import datetime


# create a timedelta that represents one day

one_day = datetime.timedelta(days=1)


# create a timedelta that represents two hours

two_hours = datetime.timedelta(hours=2)


# create a timedelta that represents thirty minutes thirty_minutes =
datetime.timedelta(minutes=30)
```

You can then use these timedelta objects to perform arithmetic on date and datetime objects. For example:

```
import datetime


today = datetime.date.today()


# add one day to the current date

tomorrow = today + datetime.timedelta(days=1)
```

```
# subtract thirty minutes from the current time

time_thirty_minutes_ago = datetime.datetime.now() -
datetime.timedelta(minutes=30)
```

You can also use the **timedelta** class to find the difference between two dates or times by subtracting one from the other. For example:

```
import datetime


# find the difference between two dates

date1 = datetime.date(1997, 2, 3)

date2 = datetime.date(2020, 5, 6)

date_difference = date2 - date1


# find the difference between two times

time1 = datetime.time(12, 30, 0)

time2 = datetime.time(14, 45, 0)

time_difference = time2 - time1
```

After this code has been executed, the **date_difference** variable will hold a **timedelta** object representing the difference between the two dates (in this case, 8557 days), and the **time_difference** variable will hold a **timedelta object** representing the difference between the two times (in this case, 2 hours and 15 minutes).

You can then use the various attributes of the **timedelta** object, such as days, seconds, and microseconds, to retrieve specific parts of the duration. For example:

```
import datetime


date1 = datetime.date(1997, 2, 3)

date2 = datetime.date(2020, 5, 6)

date_difference = date2 - date1


# print the total number of days between the two dates

print(date_difference.days)


# print the total number of seconds between the two dates

print(date_difference.total_seconds())
```

This code will print the following output:

```
> 8557 736834400.0
```

Chapter Review

- How do you get today’s date with Python?
- How do you get the current time?
- How do you get the current time zone?
- How do you find the difference between two times?

Chapter 12
Networking and JSON

Networking

What is HTTP?

HTTP (**Hypertext Transfer Protocol**) is a protocol for sending and receiving data on the internet. It is the foundation of how data is exchanged on the web, and is used by **web browsers** and **web servers** to communicate with each other. In general, when you enter a URL into your web browser, your browser sends an **HTTP** request to the appropriate **web server**, and the server responds by sending back the requested data. This data is then displayed in your browser. **HTTP** is a key component of the net, and allows for the retrieval of data from the web, such as web pages, images, and videos.

How do I make an HTTP request in Python?

To make an **HTTP** request in Python, you can use the request module from the **urllib package**. Here is an example of how to use this module to make a **GET** request to a **web server**:

```
import urllib.request



# Set the URL you want to send the request to
```

```
url = "http://www.example.com"


# Create a request object

request = urllib.request.Request(url)


# Send the request and retrieve the response

response = urllib.request.urlopen(request)


# Print the response

print(response.read())
```

This example will send a **GET** request to the specified URL and print the response that the server sends back. You can also use the **urllib.request.urlretrieve()** function to download the data from the response and save it to a file on your local machine.

How do I make a POST request in Python?

To make a **POST** request in Python, you can use the request module from the **urllib** package. Here is an example of how to use this module to make a **POST** request:

```
import urllib.request


# Set the URL you want to send the request to

url = "http://www.example.com"
```

```
# Set the data you want to send in the request

data = { "key1": "value1", "key2": "value2", }


# Encode the data as a URL query string

data = urllib.parse.urlencode(data)
```

```
# Create a request object

request = urllib.request.Request(url, data=data.encode())


# Send the request and retrieve the response

response = urllib.request.urlopen(request)


# Print the response

print(response.read())
```

This example will send a **POST** request to the specified URL with the specified data, and print the response that the server sends back. Like above, you can also use the **urllib.request.urlretrieve()** function to save it.

How can I host a webserver in Python?

To host a **webserver** in Python, you can use the **http.server module**. This module provides classes for implementing **HTTP** servers, also informally known as **web servers**. With the help of these classes, you can easily create a basic **web server** to serve static files and handle **HTTP** requests.

Here is an example of how you can use the **http.server** module to create a simple **webserver**:

```
# Import the necessary modules

import http.server

import socketserver


# Define the server address and port

SERVER_ADDRESS = ("localhost", 8000)


# Create a request handler class

class RequestHandler(http.server.BaseHTTPRequestHandler):

    # Handle GET requests

    def do_GET(self):

        self.send_response(200)

        self.send_header(

            "Content-type",

            "text/plain"

        )

        self.end_headers()

        self.wfile.write("Hello there!".encode())


# Create a server object
```

```
server = http.server.HTTPServer(SERVER_ADDRESS, RequestHandler)
```

```
# Start the server

server.serve_forever()
```

This example creates a server that listens for incoming connections on **localhost port** 8000. When the server receives a **GET** request, it sends a "Hello there!" response to the client.

You can run this script using the following command:

```
python server.py
```

This will start the server, which will listen for incoming connections on the specified address and port. You can then access the server using a web browser or a tool like curl:

```
curl http://localhost:8000/
```

What is a server socket?

A **server socket** is a type of network socket that is used by a server to communicate with clients. In a client-server architecture, the server listens for incoming connections from clients, and creates a new **socket** for each client that connects. The server can then use the **socket** to send and receive data with the client.

Server sockets are typically created using the **socket module** in Python, and they use the **bind()**, **listen()**, and **accept()** methods to start listening for incoming connections. Once a connection is established, the server can use the socket to communicate with the client using the **send()** and **recv()** methods.

In general, **server sockets** are useful for creating network applications that need to listen for incoming connections from clients, such as **web servers**, chat servers, and file servers.

Can a server socket be used to make a simple chat app?

Yes, a **server socket** can be used to create a simple chat app. To do this, you would create a **server socket** that listens for incoming connections from clients, and then creates a new socket for each client that connects.

Once a connection is established, the server and client can use their respective sockets to send and receive messages with each other. For example, the server could send a message to the client using the **send()** method, and the client could receive the message using the **recv()** method.

To make the chat app more interactive, you could use a separate **thread** to handle each client connection, which would allow the server to handle multiple clients simultaneously. You could also implement additional features, such as allowing clients to send private messages to each other, or displaying a list of all connected clients.

JSON

What is JSON?

JSON (**JavaScript Object Notation**) is a lightweight, text-based, language-independent data interchange format that is used for exchanging data between servers and web applications. **JSON** is a very common format for transmitting data over the web, and it is natively supported by many programming languages, including **Python**, **JavaScript**, and **Ruby**.

JSON is based on a subset of the **JavaScript** programming language, and it has a simple, human-readable syntax for representing data as collections of key-value pairs. Here's an example of a simple **JSON** object:

```
{ "name": "Alice",

 "age": 25,

 "languages": ["English", "French", "Spanish"]

}
```

In this example, the **JSON** object has three keys: "name", "age", and "languages". The "name" and "age" keys have string and numeric values, respectively, while the "languages" key has an array value that contains a list of strings. **JSON** supports many other data types, including **objects**, **arrays**, and **booleans**.

JSON is commonly used for transmitting data over the web because it is easy to read and write, and it is easy to parse and generate using programming languages. Many web **APIs** use **JSON** as their primary data format for transmitting data between servers and web applications.

How do I convert something to JSON in Python?

To convert something to **JSON** in Python, you can use the **json.dumps()** function, which is part of the built-in json module in Python. This function takes an object (such as a Python dictionary) and returns a string that represents the object as a **JSON** object. Here's an example:

```
import json


data = {

"name": "Alice",

"age": 25,

"languages": ["English", "French", "Spanish"]

}


json_str = json.dumps(data)

print(json_str)
```

In this example, the data dictionary is converted to a **JSON** string using the **json.dumps()** function. The resulting **JSON** string is then printed to the console. Here's how the output would look:

```
{

"name": "Alice",

"age": 25,

"languages": ["English", "French", "Spanish"]

}
```

As you can see, the **JSON** string is a valid representation of the original data, and it can be easily parsed and processed by other programming languages or tools.

You can also specify additional options when using the **json.dumps()** function, such as the indentation level or the encoding of the resulting **JSON** string. You can learn more about these options in the Python documentation.

How do I convert JSON to a Python object?

To convert **JSON** to a Python object, you can use the **json.loads()** function, which is part of the built-in **json module** in Python. This function takes a string that represents a **JSON** object and converts it into a Python object (such as a **dictionary** or **list**) that can be used in your Python code. Here's an example:

```
import json


json_str = '{"name": "Alice", "age": 25, "languages": ["English", "French",
"Spanish"]}'




data = json.loads(json_str)

print(data)
```

In this example, the **json_str** variable contains a **JSON** string that represents a **dictionary** with three keys: "name", "age", and "languages". This **JSON** string is then converted to a Python dictionary using the **json.loads()** function. The resulting **dictionary** is then printed to the console. Here's how the output would look:

```
> {'name': 'Alice', 'age': 25, 'languages': ['English', 'French',
'Spanish']}
```

As you can see, the **JSON** string has been successfully converted to a Python **dictionary**, and you can now use this dictionary in your Python code just like any other **dictionary**.

You can also use the **json.loads()** function to convert **JSON** arrays to Python lists, and **JSON** booleans and numbers to Python booleans and numbers, respectively.

Django

What is Django?

Django is a popular web framework written in Python. It is designed to help developers build secure, scalable, and maintainable web applications quickly and easily. **Django** provides a really awesome declarative interface for creating database-driven web applications. It is built on top of the Python language, and it includes many features such as authentication, URL routing, templating, and object-relational mapping to make it easier to build sites.

Django was initially released in 2005, and since then, it has become one of the most popular **web frameworks** for Python. It is used by many well-known companies, including Instagram, Pinterest, and The Washington Times.

One of the key important features of **Django** is its "*batteries-included*" approach. This means that **Django** comes with a lot of built-in libraries so you don't have to spend time and effort on building common components from scratch. For example, **Django** includes a built-in authentication system, a powerful ORM for working with databases, and a suite of tools for creating and sending emails.

Django is also highly customizable. It uses a pluggable architecture, so you can easily swap out any of the built-in components with your own custom implementations. This allows you to tailor **Django** to your specific needs and preferences.

How do I install Django?

To install **Django**, you will need to have Python installed on your system. **Django** is a Python web framework, so you will need to have Python installed to use it.

Afterward you can use the **pip** command to install **Django**. **pip** is a package manager for Python as you should know by now!

To install **Django** using **pip**, open a **terminal** or **command prompt**, and run the following command:

```
pip install Django
```

This will install the latest version of **Django** on your system. Once the installation is complete, you can verify that Django was installed successfully by running the following command:

```
python -m django --version
```

This will print the version of **Django** that was installed on your system. If the installation was successful, you should see the version number printed to the terminal.

How do I make a web app with Django?

To create a web app with **Django**, you will need to follow these steps:

1. *Install **Django***: as mentioned above, you will need to have **Django** installed on your system to create a web app with it.
2. *Create a new **Django** project*: the next step is to create a new **Django** project. To do this, open a terminal or command prompt and run the following command: *django-admin startproject myproject*.
3. *Create a new app*: in **Django**, an app is a self-contained module that contains the code for a specific feature or functionality. To create a new app, run the following command: *python manage.py startapp myapp*.
4. *Define your models*: in **Django**, models are Python classes that represent the data in your database. To create a model, open the *models.py* file within your app directory, and define a new class that **extends** the *django.db.models.Model* **class**.
5. *Create and run migrations*: After you have defined your models, you will need to create the database tables that will store your data. In **Django**, this is done using migrations.
6. *Register your app*: After you have created your app, you need to register it with your **Django** project.
7. *Create views*: Views in **Django** are Python functions that handle web requests and return **HTTP** responses. To create a view, open the *views.py* file within your app directory, and define a new function that takes a request parameter.
8. Define URL patterns: In **Django**, URL patterns are used to map URLs to views. To define URL patterns, open the *urls.py* file within your app directory, and add a new *urlpatterns* list.
9. Test your app! After you have defined your views and URL patterns, you can start the Django development server to test your app using the following command: *python manage.py runserver*. This will start the development server, and you can access your app at http://127.0.0.1:8000/.

These are the basic steps for creating a web app with **Django**. Of course, your app will likely have more features and functionality than the simple example above, but this should give you a good starting point. Please check the **Django** documentation for the full steps to find what will best suit your specific needs!

Chapter Review

- What is a web request?
- What is a web server?
- What is JSON?
- What is a web socket?
- What is Django for?

Chapter 13
Files, Video, Audio, and More

Files

How do I read from a file in Python?

To read from a file in Python, you first need to open the file in **reading mode**. This is done using the **open()** function, which takes the file name and the mode ("r" for reading) as arguments.

Here is an example of how to open a file in **reading mode**:

```
# open the file "myfile.txt" in reading mode

file = open("myfile.txt", "r")
```

Once the file is open, you can use the **read()** method to read the entire contents of the file as a string. You can also use the **readline()** method to read a single line from the file, or the **readlines()** method to read all lines of the file as a list of strings.

Here is an example of how to use the **read()** method to read the entire contents of a file:

```
# open the file "myfile.txt" in reading mode

file = open("myfile.txt", "r")
```

```
# read the entire contents of the file as a string

contents = file.read()


# print the contents of the file

print(contents)
```

In this example, the **read()** method is used to read the entire contents of the file "*myfile.txt*" as a string. The contents of the file are then printed to the screen.

After you have finished reading from a file, you should always remember to close the file using the **close()** method. This is important because it frees up system resources that are associated with the file.

Here is an example of how to close a file after reading from it:

```
# open the file "myfile.txt" in reading mode

file = open("myfile.txt", "r")


# read the entire contents of the file as a string

contents = file.read()


# print the contents of the file

print(contents)


# close the file
```

```
file.close()
```

In this example, the file is first opened in reading mode, the contents are read and printed to the screen, and then the file is closed.

How do I write to a file in Python?

To write to a file in Python, you first need to open the file in **writing mode**. This is done using the **open()** function, which takes the file name and the mode ("w" for writing) as arguments.

Here is an example of how to open a file in writing mode:

```
# open the file "myfile.txt" in writing mode

file = open("myfile.txt", "w")
```

Once the file is open, you can use the **write()** method to write a string to the file. This method takes the string to be written as an argument and writes it to the file.

Here is an example of how to use the **write()** method to write a string to a file:

```
# open the file "myfile.txt" in writing mode

file = open("myfile.txt", "w")



# write the string "Hello there!" to the file

file.write("Hello there!")
```

In this example, the **write()** method is used to write the string "*Hello there*!" to the file "*myfile.txt*".

After you have finished writing to a file, you should always remember to close the file using the **close()** method. This is important because it ensures that any remaining data is flushed to the file and frees up system resources that are associated with the file.

Here is an example of how to close a file after writing to it:

```
file = open("myfile.txt", "w")

file.write("Winter is coming!")



# close the file

file.close()
```

In this example, the file is first opened in **writing mode**, the string is written to the file, and then the file is closed.

How do I delete files in Python?

To delete a file in Python, you can use the **os module** and the **os.remove()** method. The **os module** provides functions for interacting with the **operating system**, and the **os.remove()** method can be used to delete a file.

Here is an example of how to use the **os.remove()** method to delete a file:

```
import os



# delete the file "myfile.txt"

os.remove("myfile.txt")
```

In this example, the **os.remove()** method is used to delete the file "*myfile.txt*". If the file does not exist, an error will be raised.

It is important to note that the **os.remove()** method only deletes the file, and it does not remove any directories. If you want to delete a directory and all of its contents, you can use the **shutil module** and the **shutil.rmtree()** method.

Here is an example of how to use the **shutil.rmtree()** method to delete a directory and all of its contents:

```
import shutil


# delete the directory "mydir" and all of its contents

shutil.rmtree("mydir")
```

In this example, the **shutil.rmtree()** method is used to delete the directory "*mydir*" and all of its contents, including any files or subdirectories.

Video

Can Python be used to edit videos?

Yes, Python can be used to edit videos. Python provides several libraries and modules that allow you to manipulate and edit video files, such as the **OpenCV** library, which provides tools for video capture, analysis, and processing, and the **moviepy** library, which provides tools for video editing, cutting, concatenation, and compositing.

To edit a video with Python, you can use the **OpenCV** library to read and decode the video frames, and to apply various image processing operations, such as cropping, resizing, and filtering, to the frames. You can then use the

moviepy library to encode the edited frames into a new video file, using a specified **codec** and frame rate.

How do I edit a video with OpenCV using Python?

To edit a video with **OpenCV** using Python, you can use the **VideoCapture** class from the **cv2 module** to read and decode the video frames, and the **VideoWriter** class to encode and write the edited frames to a new video file. The **VideoCapture** and **VideoWriter** classes allow you to specify various options, such as the codec and frame rate, to customize the input and output video files.

Here is an example of how to edit a video with **OpenCV** using Python:

```
import cv2


video = cv2.VideoCapture("in.mp4")


# Create a new video file using OpenCV

output = cv2.VideoWriter("out.mp4", cv2.VideoWriter_fourcc(*"MP4V"), 30,
(1920, 1080))


# Iterate over the frames in the video

while video.isOpled():

    # Read a frame from the video

    success, frame = video.read()


    # Check if there are no more frames
```

```
if not success:

    break


# Apply image processing operations to the frame

frame = cv2.cvtColor(frame, cv2.COLOR_BGR2GRAY)

frame = cv2.resize(frame, (1920, 1080))


# Write the edited frame to the output video

output.write(frame)


# Close the video files

video.release()

output.release()
```

In this example, the **VideoCapture** class is used to read the input video file, and the **VideoWriter** class is used to create the output video file. The **VideoCapture** class returns a **VideoCapture** object, which provides methods for reading and decoding the frames of the video, and the **VideoWriter** class returns a **VideoWriter** object, which provides methods for writing and encoding the frames of the video.

The example iterates over the frames in the input video, using the **read()** method of the **VideoCapture** object to read and decode the frames. The **read()** method returns a tuple of two values, the first of which is a Boolean value that indicates whether the frame was successfully read, and the second of which is the frame itself, as a **numpy array**.

The example applies some image processing operations to the frame, such as converting the frame to grayscale and resizing the frame, using the appropriate methods from the **cv2 module**. The edited frame is then written to the output video using the **write()** method of the **VideoWriter** object.

Finally, the example calls the release() method on the **VideoCapture** and **VideoWriter** objects to close the input and output video files. This is important to ensure that the video files are properly saved and closed, and that the associated resources are released.

Can I use ffmpeg with Python to edit videos?

Yes, you can use **ffmpeg** with Python to edit videos. **ffmpeg** is a popular and powerful open-source command-line tool for transcoding and manipulating multimedia files, such as audio and video files. **ffmpeg** provides a rich set of command-line options and arguments that allow you to perform various operations on multimedia files, such as resizing, cutting, concatenating, and transcoding.

To use **ffmpeg** with Python, you can use the **subprocess module**, which provides tools for running and interacting with external command-line programs from Python. The **subprocess module** allows you to run ffmpeg as a separate process, and to pass the appropriate command-line options and arguments to ffmpeg to perform the desired operations on the video file.

Here is an example of how to use **ffmpeg** with Python to edit a video:

```
# Import the subprocess module

import subprocess



# Define the command to run ffmpeg
```

```
command = [ "ffmpeg", "-i", "input.mp4", "-vf", "scale=640:-1", "-c:v", "libx264",
"-crf", "18", "-preset", "veryfast", "-c:a", "copy", "output.mp4", ]



# Run the ffmpeg command using the subprocess module subprocess.run(command)
```

In this example, the **subprocess.run()** method is used to run the **ffmpeg** command, which is defined as a list of strings that specify the command-line options and arguments to pass to ffmpeg. The **subprocess.run()** method runs ffmpeg as a separate process, and it waits for ffmpeg to finish executing before returning the result of the **ffmpeg** command.

The **ffmpeg** command in this example specifies the input and output video files, as well as various options and arguments to apply to the input video, such as the video filter to use to resize the video, the **codec** to use to encode the output video, and the quality and speed settings to use for the encoding process.

Audio

Can I edit audio with Python?

Yes, you can edit audio with Python. Python provides several libraries and modules that allow you to manipulate and edit audio files, such as the **librosa** library, which provides tools for audio analysis, processing, and synthesis, and the **pydub** library, which provides tools for audio editing, cutting, and concatenation.

To edit an audio file with Python, you can use the **librosa** library to load and decode the audio file, and to apply various signal processing operations, such as filtering, resampling, and normalization, to the audio signal. You can then use the **pydub** library to encode the edited signal into a new audio file, using a specified codec and sample rate.

Here is an example of how to edit an audio file with Python and the **librosa** and **pydub** libraries:

```
# Import the librosa and pydub libraries

import librosa from pydub

import AudioSegment


# Load the audio file using librosa

audio, sr = librosa.load("input.mp3", sr=None)


# Apply signal processing operations to the audio signal

audio = librosa.resample(audio, sr, 16000)

audio = librosa.util.normalize(audio)
```

```
# Convert the edited signal to a pydub audio segment

segment = AudioSegment( audio.tobytes(), frame_rate=sr, sample_width=2,
channels=1, codec="pcm_s16le" )


# Save the edited audio segment to a new file segment.export("output.mp3",
format="mp3")
```

In this example, the **librosa.load()** function is used to load the input audio file, and the **librosa.resample()** and **librosa.util.normalize()** functions are used to apply various signal processing operations to the audio signal. The edited audio signal is then converted to an mp3 file and saved.

Can I make music with Python?

Yes, you can make music with Python. Python provides several libraries and modules that allow you to create and generate music, such as the **music21 library**, which provides tools for musical notation, analysis, and composition, and the **pretty_midi library**, which provides tools for generating and manipulating **MIDI** data.

To make music with Python, you can use the **music21** library to create and manipulate musical scores, using the appropriate classes and methods provided by the library. The **music21** library allows you to represent and manipulate musical elements, such as notes, chords, melodies, and rhythms, and to combine them into complex musical structures, such as phrases, measures, and movements.

You can then use the **pretty_midi library** to convert the musical score created with **music21** into a MIDI file, which can be played and edited using a MIDI editor or synthesizer. The **pretty_midi library** allows you to specify various options, such as the tempo, instrument, and dynamics, to customize the **MIDI** file generated from the musical score.

Here is an example of how to make music with Python and the **music21** and **pretty_midi libraries**:

```
# Import the libraries

from music21 import *



import pretty_midi
```

```
# Create a new musical score using music21

score = stream.Score()

part = stream.Part()



# Add some notes

part.append(note.Note("B4", quarterLength=1.0)) part.append(note.Note("A4",
quarterLength=1.0)) part.append(note.Note("F4", quarterLength=1.0))



# Add to the score

score.insert(0, part)



 # Convert the score to a MIDI file using pretty_midi

midi_file = pretty_midi.PrettyMIDI()
midi_file.instruments.append(pretty_midi.Instrument(0, program=0))

midi_file.timeSignature = meter.TimeSignature("3/4") midi_file.tempo = 110



# Add the notes to the MIDI file

for note in part.notes:

    midi_note = pretty_midi.Note( start=note.offset, end=note.offset +
note.duration.quarterLength, pitch=note.pitch.midi, velocity=90 )

    midi_file.instruments[0].notes.append(midi_note)
```

```
# Save your new MIDI file

midi_file.write("output.mid")
```

In this example, the **music21 library** is used to create a new musical score, and to add some notes to the score. The **music21 library** provides a rich and flexible API that allows you to represent and manipulate various aspects of music, such as notes, pitches, rhythms, and chords, using the appropriate classes and methods.

Desktop Apps

How do I make a desktop app in Python?

To create a desktop application with Python, you will need to use a **GUI** library to handle the graphical user interface. Some popular choices for building Python desktop applications include **Tkinter**, **PyQt**, and **wxPython**. These libraries provide a simple way to create a graphical interface for your application, allowing you to add widgets such as buttons, labels, and text input fields.

To get started, you will need to install the chosen GUI library using **pip**. For example, to install **Tkinter**, you would run the following command:

```
pip install tkinter
```

Once you have installed the library, you can use it to create a new window and add widgets to it. For example, the following code creates a simple window with a text label and a button using Tkinter:

```
import tkinter as tk
```

```
# Create the main window

root = tk.Tk()


# Add a label with some text

label = tk.Label(root, text="Hello there!")

label.pack()


# Add a button with a callback function

def on_click():

    print("Button clicked!")
```

```
button = tk.Button(

    root,

    text="Click me!",

    command=on_click

)

button.pack()


# Start the main event loop

root.mainloop()
```

This code creates a window with a label and a button. When the button is clicked, it will print a message to the console. You can modify this code to add more widgets and functionality to your application.

What kind of widgets are there in tkinter?

Tkinter is a built-in GUI library for the Python programming language. It provides a variety of standard GUI widgets, including buttons, labels, and text input fields. Some of the common **Tkinter** widgets include:

- *Button* is a simple button widget that can display text or an image
- *Label* is a widget that displays text or an image
- *Entry* is a single-line text input field
- *Radiobutton*s are buttons that can be part of a group of buttons, where only one button in the group can be selected at a time
- *Text* is a multi-line text input widget
- *Checkbutton* is a button that can be in one of two states, selected or unselected
- *Listbox* is a list of selectable items
- *ComboBox* is a combination of a drop-down list and an editable text field
- *Menubutton* for displaying menus on a button

Tkinter also provides a variety of **layout managers** that can be used to arrange widgets in the main window, such as **pack**, **grid**, and **place**. These **layout managers** allow you to specify the position and size of each widget in the window.

How do I use layout managers in tkinter?

Tkinter provides several layout managers to arrange the widgets in the main window of your application. The three most commonly used **layout**

managers in **Tkinter** are **pack**, **grid**, and **place**.

The **pack layout manager** organizes widgets in horizontal and vertical boxes. You can use the **pack()** method to add a widget to a container, and specify options such as the side of the container to which the widget will be packed (e.g. **tk.TOP**, **tk.BOTTOM**, **tk.LEFT**, **tk.RIGHT**), and whether the widget should expand to fill the available space.

```
import tkinter as tk


# Create the main window

root = tk.Tk()


# Create a frame to hold the widgets

frame = tk.Frame(root)

frame.pack()




# Create a button and pack it to the top of the frame

button1 = tk.Button(frame, text="Button 1") button1.pack(side=tk.TOP)


# Create another button and pack it to the bottom of the frame

button2 = tk.Button(frame, text="Button 2") button2.pack(side=tk.BOTTOM)


# Start the main event loop

root.mainloop()
```

The **grid layout manager** arranges widgets in a grid, allowing you to specify the row and column numbers for each widget. You can use the **grid()** method to add a widget to a container, and specify options such as the row and column numbers, and whether the widget should expand to fill the available space.

```
import tkinter as tk


# Create the main window

root = tk.Tk()




# Create a frame to hold the widgets

frame = tk.Frame(root)

frame.pack()


# Create a button and place it in the first row and first column

button1 = tk.Button(frame, text="Button 1")

button1.grid(row=0, column=0)


# Create another button and place it in the first row and second column

button2 = tk.Button(frame, text="Button 2")

button2.grid(row=0, column=1)


# Start the main event loop
```

```
root.mainloop()
```

The **place layout manager** allows you to specify the exact position and size of each widget in the main window, using absolute coordinates. You can use the **place()** method to add a widget to a container, and specify options such as the coordinates and size of the widget.

```
import tkinter as tk


# Create the main window

root = tk.Tk()


# Create a frame to hold the widgets

frame = tk.Frame(root)

frame.pack()


# Create a button and place it at the top left corner of the frame

button1 = tk.Button(frame, text="Button 1")

button1.place(x=0, y=0, width=100, height=50)


# Create another button and place it at the bottom right corner of the frame

button2 = tk.Button(frame, text="Button 2")

button2.place(x=200, y=200, width=100, height=50)
```

```
# Start the main event loop

root.mainloop()
```

You can choose the **layout manager** that best suits your needs.

Gaming

Can I make games in Python?

Yes, you can make games in Python. Python is a popular and versatile programming language that provides a rich set of libraries and modules for developing various types of applications, including games. Python is well-suited for game development, thanks to its simplicity, expressiveness, and flexibility, as well as the availability of powerful libraries and frameworks for game development.

To make games in Python, you can use a **game development framework** or library, such as **Pygame**, **Pyglet**, or **Kivy**, which provides the necessary tools and components for building and running games in Python. These frameworks and libraries provide various features and capabilities, such as graphics, audio, input, and networking, that allow you to create and develop games in Python.

Chapter Review

- How do you read/write files in Python?
- What are two ways to edit videos with Python?
- How easy is it to edit and make music/audio with Python?
- What are some of the widget types in Tkinter?

- What are some of the layout managers in Tkinter and what are their differences?

Chapter 14
Images and Threads

Images

Can Python do image manipulation?

Yes, Python can be used for image manipulation. Python has several built-in modules and third-party libraries that provide tools for working with images, such as the **Pillow** library.

With these libraries, you can perform a variety of image manipulation tasks in Python, such as resizing, cropping, and rotating images, as well as applying filters and other transformations. You can also use Python to create and manipulate complex images, such as digital art, using techniques such as drawing shapes and text, and blending multiple images together.

Overall, Python provides a powerful and flexible platform for working with images, and it is widely used in fields such as computer vision and graphic design.

How do I resize an image with Python?

To resize an image with Python, you can use the **Pillow** library, which is a fork of the **Python Imaging Library** (**PIL**). To resize an image with **Pillow**, you can use the **resize()** method, which takes the desired size as arguments, and returns a new Image object with the resized image.

Here is an example of how to resize an image with Python using **Pillow**:

```
from PIL import Image


# Open the image

image = Image.open("my_image.jpg")


# Resize the image

resized_image = image.resize((200, 200))


# Save the resized image

resized_image.save("resized_image.jpg")
```

In this example, the **resize()** method is used to resize the image to a width of 200 pixels and a height of 200 pixels. The resized image is then saved to a new file called *resized_image.jpg*.

Note that the **resize()** method takes the new size as a tuple of width and height values in pixels, and it preserves the aspect ratio of the original image by default. You can also use the **resize()** method to scale the image by a specific factor, or to crop the image to a specific size.

How do I flip an image with Python?

To flip an image with Python, you can use the **transpose()** method from the **Pillow** library. The **transpose()** method allows you to flip, rotate, and transform images in various ways, and it takes a constant value as an argument to specify the type of transformation to apply.

To flip an image with Python using **Pillow**, you can use the **Image.FLIP_LEFT_RIGHT** or **Image.FLIP_TOP_BOTTOM** constant to flip the image horizontally or vertically, respectively. Here is an example of how to flip an image with Python using **Pillow**:

```
from PIL import Image


# Open the image

image = Image.open("my_image.jpg")


# Flip the image horizontally

flipped_image = image.transpose(Image.FLIP_LEFT_RIGHT)


# Save the flipped image

flipped_image.save("flipped_image.jpg")
```

In this example, the **transpose()** method is used to flip the image horizontally, using the **Image.FLIP_LEFT_RIGHT** constant. The flipped image is then saved to a new file called *flipped_image.jpg*.

Note that the **transpose()** method creates a new Image object with the transformed image, so you need to save the transformed image to a new file or overwrite the original image if you want to keep the changes. You can also use the **transpose()** method to rotate the image, or to apply other transformations, by using different constant values.

How do I rotate images with Python?

To rotate an image with Python, you can use the **transpose()** method from the Pillow library.

To rotate an image with Python using Pillow, you can use the **Image.ROTATE_90**, **Image.ROTATE_180**, or **Image.ROTATE_270** constant to rotate the image by 90, 180, or 270 degrees, respectively. Here is an example of how to rotate an image with Python using Pillow:

```
from PIL import Image


# Open the image

image = Image.open("my_image.jpg")


# Rotate the image by 90 degrees

rotated_image = image.transpose(Image.ROTATE_90)


# Save the rotated image

rotated_image.save("rotated_image.jpg")
```

In this example, the **transpose()** method is used to rotate the image by 90 degrees, using the **Image.ROTATE_90** constant. The rotated image is then saved to a new file called *rotated_image.jpg*.

How can I draw on an image in Python?

To draw on an image with Python, you can use the **ImageDraw** module from the **Pillow** library. Here is an example of how to draw on an image using the **ImageDraw** module:

```
from PIL import Image, ImageDraw


# Open the image

image = Image.open("my_image.jpg")


# Create an ImageDraw object

draw = ImageDraw.Draw(image)


# Draw a rectangle on the image

draw.rectangle((100, 100, 200, 200), fill="red")


# Save the modified image

image.save("modified_image.jpg")
```

In this example, the **ImageDraw.Draw()** method is used to create an **ImageDraw** object, which is used to draw shapes on the image. The **rectangle()** method is then used to draw a rectangle on the image, using the specified coordinates and fill color. Finally, the modified image is saved to a new file called *modified_image.jpg*.

How can I write text on an image using Python?

Here is an example of how to write text on an image with Python using the **ImageDraw** module:

```
from PIL import Image, ImageDraw
```

```
# Open the image

image = Image.open("my_image.jpg")


# Create an ImageDraw object

draw = ImageDraw.Draw(image)


# Write text on the image

draw.text((100, 100), "Hey there!", fill="red")


# Save the modified image

image.save("modified_image.jpg")
```

Note that the **text()** method takes the position of the text as a tuple of x and y coordinates, and it allows you to specify the font, font size, and other text attributes as optional arguments. You can also use the **ImageDraw** module to draw other shapes and objects on the image, and to apply transformations and other effects to the image.

How can I draw an image on another image in Python?

Here is an example of how to draw one image on top of another image with Python using the **Pillow** library:

```
from PIL import Image
```

```
# Open the two images

image1 = Image.open("image1.jpg")

image2 = Image.open("image2.jpg")


# Draw image2 on top of image1

image1 = Image.alpha_composite(image1, image2)


# Save the resulting image

image1.save("result.jpg")
```

Note that the **Image.alpha_composite()** method allows you to control the transparency of the images by specifying their alpha values, and it allows you to specify the position of the second image relative to the first image. You can also use the **Image.alpha_composite()** method to blend multiple images together, by calling it multiple times with different images.

Threads

In programming, what are threads?

In programming, a **thread** is a separate flow of execution within a program. This means that a program can have multiple threads running concurrently, each of which can execute a different task or set of tasks. **Threads** are useful for running multiple tasks simultaneously, such as downloading a file from the internet while also rendering a user interface, or performing multiple calculations at the same time.

Using **threads** can improve the performance and responsiveness of a program, since it allows the program to perform multiple tasks at the same time instead of having to execute them one after the other. However, managing threads can also be complex, and if not done correctly, can lead to errors or unexpected behavior in a program.

How do I make multiple threads in Python?

To create multiple **threads** in Python, you can use the **threading module**. Here is an example of how to create multiple threads and start them simultaneously:

```
import threading


def task1():

    # Do something here

    print("Hello from thread 1!")


def task2():

    # Do something here

    print("Hello from thread 2!")


# Create the first thread

thread1 = threading.Thread(target=task1)


# Create the second thread

thread2 = threading.Thread(target=task2)
```

```
# Start the first thread

thread1.start()

# Start the second thread

thread2.start()
```

In this example, we define two functions, **task1** and **task2**, which represent the tasks we want the threads to perform. We then create two **Thread** objects, passing each function as the target argument. Finally, we call the **start()** method on each **Thread** object to start the threads. Once the **start()** method is called on a **Thread** object, the function passed to the target argument will be executed in a separate thread.

Note that the order in which the threads are started does not necessarily reflect the order in which they will be executed. In this example, the two threads may be executed simultaneously, or one may be executed before the other depending on various factors such as the system's resources and the time it takes for each thread to complete its task.

How do I wait for threads to finish in Python?

To wait for threads to finish in Python, you can use the **join()** method on the **Thread** objects. The **join()** method blocks the execution of the calling thread until the thread that it is called on finishes its execution. Here is an example of how to use the **join()** method to wait for threads to finish:

```
import threading



def task1():

    print("Hello from thread 1!")
```

```
def task2():
    print("Hello from thread 2!")

# Create the first thread
thread1 = threading.Thread(target=task1)
# Create the second thread
thread2 = threading.Thread(target=task2)

# Start the first thread
thread1.start()
# Start the second thread
thread2.start()

# Wait for the first thread to finish
thread1.join()
# Wait for the second thread to finish
thread2.join()

# Do something here after the threads have finished
print("Both threads have finished!")
```

In this example, we create two **Thread** objects and start them using the **start()** method. We then call the join() method on each Thread object to wait for the threads to finish. Once the **join()** method returns, we know that the thread has finished its execution and we can continue with the rest of the program.

Note that calling the **join()** method on a **thread** will block the execution of the calling thread until the thread that it is called on finishes its execution. This means that if you call the **join()** method on a thread from the main thread of your program, your program will not be able to do anything else until the thread finishes its execution. For this reason, it is usually a good idea to use threads only for tasks that take a long time to complete, so that the main thread of your program is not blocked for an extended period of time.

Chapter Review

- What’s the best library for editing images in Python?
- How do you rotate or resize an image?
- How do you composite images?
- What is a thread?
- How do you wait for your threads to all finish?

Chapter 15
Databases

MySQL

What is MySQL?

MySQL is a popular open-source **relational database** management system. It is typically used for web application development and is known for its speed, reliability, and ease of use. **MySQL** is used by many different types of organizations, from small businesses to large enterprises, to store and manage their data.

What is SQL?

SQL is a programming language that is used to manage and manipulate data stored in **relational databases**. It stands for *Structured Query Language* and is used to communicate with databases to perform various operations, such as inserting, updating, and deleting data. **SQL** is the standard language for interacting with relational databases, and is used by a wide range of systems, including **MySQL**.

What is a relational database?

A **relational database** is a type of database that stores data in a structured format, using tables and relationships between the data. The data in a **relational database** is organized into separate **tables**, with each table containing a specific set of data. **Tables** can be related to each other through a common field, allowing data from multiple tables to be combined and accessed in a single **query**. This makes it easy to manage and access large amounts of data, and to perform complex queries on the data to extract useful information. **Relational databases** are widely used in a variety of applications, including business and financial systems, e-commerce websites, and online social networks.

How do I connect to MySQL with Python?

To connect to a **MySQL** database from Python, you need to first install the appropriate Python package for **MySQL**. The recommended package for this is **mysql-connector-python**, which is a well done Python implementation of the **MySQL** client-server protocol. Once you have installed this you can use the **mysql.connector.connect()** function to connect to a **MySQL** server. This function takes a number of parameters, including the hostname of the server, the user name and password to use for authentication, and the name of the database to connect to. For example:

```python
import mysql.connector


mydb = mysql.connector.connect(
   host="localhost",
   user="yourusername",
   password="yourpassword",
   database="yourdatabase"
)
```

Once you have established a connection to the **MySQL** server, you can use the cursor() method of the **mydb** object to create a cursor that can be used to execute **SQL** queries and retrieve the results. For example:

```
mycursor = mydb.cursor()


mycursor.execute("SELECT * FROM yourtable")

myresult = mycursor.fetchall()


for x in myresult:

   print(x)
```

This code will execute a **SELECT** query on the specified table, retrieve the results, and print them to the console. You can then use the cursor object to execute other **SQL** queries as needed.

How do I make tables in MySQL using Python?

To create tables in a **MySQL** database from Python, you can use the **CREATE TABLE** statement in an **execute()** method of a cursor object. This statement takes the name of the table and a list of column definitions, as well as any other options such as primary keys and constraints. For example:

```
mycursor = mydb.cursor()


mycursor.execute("CREATE TABLE customers (name VARCHAR(255), address
VARCHAR(255))")
```

This code will create a table named *customers* with two columns: *name* and *address*. The data type for each column is specified using the **SQL** syntax for defining column types. You can add additional columns to the table by including them in the list of column definitions, separated by commas. You can also specify primary keys and constraints, as well as other options, in the **CREATE TABLE** statement.

Once you have created the table, you can use other **SQL** statements such as **INSERT**, **UPDATE**, and **DELETE** to add, modify, and remove data from the table. You can also use the **SELECT** statement to retrieve data from the table and perform various operations on it.

How do I make conditional queries in MySQL using Python?

To make conditional **queries** in **MySQL** from Python, you can use the **WHERE** clause in the **SELECT** statement. The **WHERE** clause allows you to specify conditions that must be met for the **query** to return the specified rows. For example:

```
mycursor = mydb.cursor()


mycursor.execute("SELECT * FROM customers WHERE name='John'")

myresult = mycursor.fetchall()


for x in myresult:

    print(x)
```

This code will execute a **SELECT query** that retrieves all rows from the customers table where the name column is equal to 'John'. The **query** will only return rows that meet this condition, and the results will be printed to the console.

You can use other operators in the **WHERE** clause to specify different conditions, such as <, >, <=, and >= for comparing values, and **AND** and **OR** for combining multiple conditions. You can also use wildcard characters such as % and _ to match patterns in the data. For more information on the syntax and usage of the **WHERE** clause, you can refer to the **MySQL** documentation.

How do I join tables in MySQL using Python?

To join tables in **MySQL** from Python, you can use the **JOIN** clause in the **SELECT** statement. The **JOIN** clause allows you to combine rows from two or more tables based on a common field between them. For example:

```
mycursor = mydb.cursor()


mycursor.execute("SELECT * FROM customers INNER JOIN orders ON
customers.id=orders.customer_id")

myresult = mycursor.fetchall()


for x in myresult:

  print(x)
```

This code will execute a **SELECT** query that retrieves all rows from the customers and orders **tables**, and combines the **rows** based on the id field in the customers table and the *customer_id* field in the orders table. This will create a result set that contains all the data from both tables, with the rows being combined based on the matching values in the common fields.

There are several different types of **JOIN** clauses that you can use, depending on the specific requirements of your **query**. For example, you can use an **INNER JOIN** to only return **rows** that have matching values in the common field, or a **LEFT JOIN** to return all **rows** from the left table (customers in this case) and only the matching **rows** from the right table (orders). For more information on the different types of **JOIN** clauses and their usage, you can refer to the **MySQL** documentation.

How do I update a row in MySQL from Python?

To update a row in a **MySQL** table from Python, you can use the **UPDATE** statement in an **execute()** method of a cursor object. The **UPDATE** statement allows you to modify the values of one or more columns in a specified row. For example:

```
mycursor = mydb.cursor()


mycursor.execute("UPDATE customers SET name='Storm Trooper' WHERE id=1")

mydb.commit()
```

This code will execute an **UPDATE** statement that modifies the value of the name column in the row with id equal to 1, setting it to 'Storm Trooper'. The **WHERE** clause is used to specify the row that should be updated, and the **SET** clause is used to specify the new values for the columns.

Once the **UPDATE** statement has been executed, you can use the **commit()** method of the **mydb** object to save the changes to the database. *It is important to remember to call this method*, as any changes made to the database through **SQL** statements are not permanent until they are committed.

You can also use the **UPDATE** statement to modify multiple rows at once by omitting the **WHERE** clause. This will cause the statement to update all rows in the table, so it should be used with caution. For more information on the **UPDATE** statement and its usage, you can refer to the **MySQL** documentation.

How do I insert a row in MySQL using Python?

To insert a row into a **MySQL** table from Python, you can use the **INSERT INTO** statement in an **execute()** method of a cursor object. The **INSERT INTO** statement takes the name of the table and a list of values to be inserted, and inserts a new row into the table with those values. For example:

```
mycursor = mydb.cursor()


sql = "INSERT INTO customers (name, address) VALUES (%s, %s)"

val = ("Luke", "Jedi Temple")

mycursor.execute(sql, val)


mydb.commit()
```

This code will execute an **INSERT INTO** statement that inserts a new row into the customers table, with the values 'Luke' and 'Jedi Temple' for the name and address columns, respectively. The values to be inserted are specified as a tuple in the **VALUES** clause, and are passed as a second argument to the **execute()** method.

Once the **INSERT INTO** statement has been executed, you can use the **commit()** method of the **mydb** object to save the changes to the database. Remember that any changes made to the database through **SQL** statements are not permanent until committed.

You can also use the **INSERT INTO** statement to insert multiple rows at once by specifying a list of **tuples** containing the values to be inserted.

How do I delete a row in MySQL using Python?

To delete a row from a **MySQL** table from Python, you can use the **DELETE FROM** statement in an **execute()** method of a cursor object. The **DELETE FROM** statement takes the name of the table and a **WHERE** clause to specify the rows that should be deleted. For example:

```
mycursor = mydb.cursor()


mycursor.execute("DELETE FROM customers WHERE id=1")

mydb.commit()
```

This code will execute a **DELETE FROM** statement that deletes the row with id equal to 1 from the customers table. The **WHERE** clause is used to specify the row that should be deleted.

You can also use the **DELETE FROM** statement to delete multiple rows at once by omitting the **WHERE** clause. This will cause the statement to delete all rows in the table, so it should be used with caution.

How do I sort results from a MySQL query?

To sort the results of a **MySQL** query from Python, you can use the **ORDER BY** clause in the **SELECT** statement. The **ORDER BY** clause allows you to specify the order in which the rows should be returned, based on the values in one or more columns. For example:

```
mycursor = mydb.cursor()
```

```
mycursor.execute("SELECT * FROM customers ORDER BY name")

myresult = mycursor.fetchall()



for x in myresult:

    print(x)
```

This code will execute a **SELECT** query that retrieves all rows from the customers table and sorts the rows by the values in the name column. The rows will be returned in ascending alphabetical order based on the values in the name column.

You can specify multiple columns in the **ORDER BY** clause to sort the rows based on multiple criteria. You can also use the **ASC** and **DESC** keywords to specify whether the rows should be sorted in ascending or descending order. For example:

```
mycursor.execute("SELECT * FROM customers ORDER BY name ASC, address DESC")
```

This code will sort the rows first by the name column in ascending order, and then by the address column in descending order.

MongoDB

What is MongoDB?

MongoDB is a popular and powerful open-source **database management system** (**DBMS**) that is based on the **NoSQL** (not only **SQL**) data model. **NoSQL databases** are designed to store and manage data in a non-relational, distributed, and scalable way, and they are often used to support modern web and mobile applications that require high performance and flexible data storage.

MongoDB is known for its high scalability and performance, as well as its rich and expressive query language, which allows developers to easily and efficiently access and manipulate the data stored in the database. **MongoDB** also provides support for various data types and data structures, such as arrays and objects, and it offers built-in mechanisms for replication, sharding, and indexing, which help to improve the availability and performance of the database.

MongoDB is often used in combination with other technologies, such as the **Node.js** runtime and the **Express web framework**, to create modern web and mobile applications that require fast, flexible, and scalable data storage. Many companies and organizations, including Fortune 500 companies and government agencies, use **MongoDB** to power their critical business applications and services.

Overall, **MongoDB** is a powerful and popular database management system that is well-suited for modern web and mobile applications, and it offers many benefits and features that make it a valuable tool for developers and data professionals.

Can I connect to MongoDB using Python?

Yes, you can connect to **MongoDB** using Python. **MongoDB** provides drivers and libraries for various programming languages, including Python, that allow you to connect to the database and perform operations on the data stored in the database.

To connect to **MongoDB** from Python, you can use the **pymongo** library, which is the official Python driver for **MongoDB**. The **pymongo** library provides classes and methods for connecting to the **MongoDB** server, and for accessing and manipulating the data stored in the database.

Here is an example of how to connect to **MongoDB** using Python and the **pymongo** library:

```
# Import the pymongo library

import pymongo


# Connect to the MongoDB server

client = pymongo.MongoClient("mongodb://localhost:27017/")


# Select the database to use

db = client["mydatabase"]


# Select the collection to use

collection = db["mycollection"]
```

In this example, the **MongoClient** class from the **pymongo** library is used to connect to the **MongoDB** server, which is running on the **localhost** at **port** 27017. The **MongoClient** class returns a client object, which is used to select the database and collection to use for the subsequent operations.

Note that the **MongoClient** class allows you to specify various options, such as the authentication credentials and the server connection timeout, as optional arguments. You can also use the **MongoClient** class to connect to a remote **MongoDB** server, or to a cluster of **MongoDB** servers, by specifying the appropriate connection string.

How do I update my MongoDB database using Python?

To update a **MongoDB** database using Python, you can use the **update_one()** or **update_many()** method from the **pymongo** library, which is the official Python driver for **MongoDB**. These methods allow you to update one or more documents in the database that match a specified filter, using the **MongoDB** update operators to specify the update operations to apply to the matching documents.

Here is an example of how to update a **MongoDB** database using Python and the **pymongo** library:

```
# Import the pymongo library

import pymongo


# Connect to the MongoDB server

client = pymongo.MongoClient("mongodb://localhost:27017/")


# Select the database and collection to use

db = client["mydatabase"]

collection = db["mycollection"]


# Define the filter to match the documents to update

filter = {"name": "John Doe"}


# Define the update operations to apply to the matching documents

update = {"$set": {"age": 42}}


# Update the documents in the database
```

```
result = collection.update_one(filter, update)


# Print the number of updated documents print(result.modified_count)
```

In this example, the u**pdate_one()** method is used to update the documents in the collection that match the specified filter, using the *$set* operator to update the age field of the matching documents. The **update_one()** method returns a **UpdateResult** object, which includes the modified_count property that indicates the number of updated documents.

Note that the **update_one()** method updates only the first document that matches the filter, while the **update_many()** method updates all the documents that match the filter. You can also use the **update_one()** and **update_many()** methods to specify additional options, such as the upsert option, which determines whether to insert a new document if no documents match the filter.

Are relational databases better than NoSQL databases?

It is not accurate to say that **relational databases** are inherently better than **NoSQL** databases, or vice versa. Both types of databases have their own strengths and weaknesses, and the appropriate choice of a **database management system** depends on the specific requirements and goals of a project or application.

Relational databases, such as **MySQL**, **Oracle**, and **SQL Server**, are based on the relational data model, which organizes data into **tables**, **rows**, and **columns**, and uses **structured query language** (**SQL**) to access and manipulate the data. Relational databases are well-suited for applications that require high data integrity and consistency, and that involve complex queries and transactions.

NoSQL databases, on the other hand, are based on the non-relational (**NoSQL**) data model, which allows for the storage and management of data in a flexible and distributed way. **NoSQL** databases are often used for applications that require high scalability and performance, and that involve large and complex data sets, such as web and mobile applications, social media platforms, and IoT systems.

In the end, the choice between a **relational database** and a **NoSQL** database depends on the specific requirements and goals of a project or application, and it should be based on a thorough analysis of the strengths and weaknesses of each type of database. In some cases, it may be appropriate to use a combination of both types of databases, depending on the needs of the application.

Chapter Review

- What is a relational database?
- What is a NoSQL database?
- Which type is MySQL?
- Which type is MongoDB?
- What are the pros and cons of each type?

Chapter 16

Advanced Math and Science

Can I do physics calculations in Python?

Yes, you can do physics calculations in Python using a variety of different tools and libraries. Python has a rich ecosystem of scientific computing libraries that make it a powerful platform for physics calculations and simulations.

One popular library for physics calculations in Python is **NumPy**, which provides a set of functions and data structures for working with large arrays and matrices of data. **NumPy** makes it easy to perform mathematical operations on large datasets, such as computing the dot product of two vectors or solving systems of linear equations.

Another popular library for physics calculations in Python is **SciPy**, which provides a wide range of numerical algorithms and functions for scientific computing. **SciPy** includes many functions that are useful for physics, such as functions for computing distances between points in space, calculating forces and accelerations, and solving differential equations.

There are also many other libraries and frameworks available for physics calculations in Python, such as **SymPy** for symbolic mathematics, **PyParticles** for particle simulations, and **VPython** for creating 3D visualizations. With these tools, you can perform a wide range of physics calculations and simulations in Python.

NumPy

What is NumPy?

NumPy is a library for the Python programming language, adding support for large, multi-dimensional arrays and matrices, along with a large collection of high-level mathematical functions to operate on these arrays. It is a fundamental package for scientific computing with Python, and many other packages in the Python ecosystem make use of **NumPy's** functionality.

What can I do with NumPy?

NumPy is a Python library that is widely used for scientific computing and data analysis. It provides tools and functions for working with large and complex arrays of numerical data. Some of the things that you can do with NumPy include the following:

- Perform mathematical operations on arrays, such as element-wise addition, subtraction, multiplication, and division.
- Use its functions like **mean**, **median**, **min**, and **max** to compute statistical properties of arrays.
- Use vectorized operations to speed up computations on large arrays.
- Generate random numbers and arrays using functions like **rand**, **randn**, and **randint**.
- Use functions like **sort**, **unique**, and **intersect1d** to manipulate and analyze arrays.
- Perform linear algebra operations on arrays, such as matrix multiplication, inverse, and determinant.
- Use functions like **load** and **save** to read and write arrays to and from files.

- Use functions like **fft** and **ifft** to perform fast Fourier transforms on arrays.

Overall, **NumPy** is a powerful tool for working with numerical data in Python. It provides a wide range of functions and capabilities that make it easy to perform complex computations on arrays of data.

How can I get started with NumPy?

To get started with **NumPy**, you first need to install the **NumPy** library. This is typically done using the **pip** package manager. You can use the following command:

```
pip install numpy
```

Once **NumPy** is installed, you can import it into your Python scripts and use its functions and methods.

Here is an example of how to import and use NumPy in a Python script:

```
import numpy as np


# create a NumPy array with random values

arr = np.random.rand(5, 5)


# print the array

print(arr)
```

```
# compute the mean of the array
mean = np.mean(arr)


# print the mean
print(mean)
```

In this example, the **NumPy** library is imported using the **import** statement, and the **numpy** namespace is used to access its functions and methods. The **np.random.rand()** function is used to create a NumPy array with random values, and the **np.mean()** function is used to compute the mean of the array.

What are good places to learn more about NumPy?

There are many good places to learn more about **NumPy**, including the following:

1. The official **NumPy** documentation (https://numpy.org/doc/stable/). This is an excellent resource for learning about the various features of **NumPy** and how to use them. The documentation includes tutorials, guides, and API reference materials for all versions of **NumPy**.
2. **NumPy** tutorials on websites like *DataCamp* (https://www.datacamp.com/courses/intro-to-numpy). These type of websites offer interactive tutorials that can help you learn **NumPy** step by step. The tutorials include exercises and quizzes to help you practice and test your knowledge.
3. There are many books available that can help you learn **NumPy**, both for beginners and experienced data scientists. Some of the best

books for learning **NumPy** include "*Python for Data Analysis*" by Wes McKinney or "*NumPy Beginner's Guide*" by Ivan Idris.

4. There are many online forums and communities where you can ask questions and get help with your **NumPy** code. Some of the best forums and communities for **NumPy** include reddit and Stack Overflow

By using these resources, you can learn a lot about **NumPy** and how to use it effectively for data analysis and scientific computing. Remember to always ask for help if you are stuck on a problem, and don't be afraid to try new things and experiment with different approaches.

What math functions can NumPy do?

In terms of math functions, **NumPy** provides many functions that can be used to perform mathematical operations on arrays. These include functions for basic arithmetic operations like addition, subtraction, multiplication, and division, as well as functions for more advanced operations like trigonometric functions, logarithms, and exponentials.

Here is a list of some of the math functions that **NumPy** provides:

- **abs**, **fabs**, and **sqrt**: Functions for computing the absolute value, floating-point absolute value, and square root of an array.
- **sin**, **cos**, **tan**, **arcsin**, **arccos**, and **arctan**: Trigonometric functions for computing the sine, cosine, tangent, arcsine, arccosine, and arctangent of an array.
- **sinh**, **cosh**, **tanh**, **arcsinh**, **arccosh**, and **arctanh**: Hyperbolic trigonometric functions for computing the hyperbolic sine, hyperbolic cosine, hyperbolic tangent, hyperbolic arcsine, hyperbolic arccosine, and hyperbolic arctangent of an array.
- **exp** and **log**: Functions for computing the exponential and natural logarithm of an array.
- **log10** and **log2**: Functions for computing the base-10 and base-2 logarithm of an array.
- **ceil**, **floor**, and **trunc**: Functions for computing the ceiling, floor, and truncated value of an array.

- **degrees** and **radians**: Functions for converting angles from degrees to radians and vice versa.

In addition to these functions, NumPy also provides many other functions for working with arrays and much more.

What is NumPy ufunc?

NumPy ufuncs are designed to be fast and efficient, and they can perform operations on arrays of data much more quickly than if you were to perform the same operations using loops in pure Python. For example, you can use a **NumPy ufunc** to add together all the elements in a large array, or to compute the square root of every element in an array, or to apply any other mathematical function to every element in an array.

Here's an example of using a **NumPy ufunc** to add two arrays together element-wise:

```
import numpy as np


# Define two arrays

a = np.array([1, 2, 3, 4, 5])

b = np.array([6, 7, 8, 9, 10])


# Use a NumPy ufunc to add the arrays element-wise

c = np.add(a, b)

print(c)



> [7, 9, 11, 13, 15]
```

As you can see, the **add()** function takes two arrays as input and returns a new array where each element is the sum of the corresponding elements in the input arrays.

There are many different **ufuncs** available in **NumPy**, and you can use them to perform a wide variety of mathematical operations on arrays of data. For more information, you can check out the **NumPy** documentation.

What are some advanced uses of NumPy?

NumPy is a powerful library for working with large arrays and matrices of data, and there are many advanced uses for the library. Here are a few examples of advanced uses for **NumPy**:

- **NumPy** includes a set of functions and classes for performing linear algebra operations, such as matrix multiplication, solving systems of linear equations, and computing the determinant of a matrix. This can be useful for many applications, such as machine learning, data analysis, and scientific computing.
- **NumPy** includes functions for computing discrete Fourier transforms and inverse Fourier transforms, which are important tools for signal processing and other applications. With these functions, you can perform operations such as filtering, convolution, and correlation on signals and other data.
- **NumPy** includes functions for generating random numbers and arrays of random data. This can be useful for many applications, such as simulating physical systems, testing algorithms, and creating random data for machine learning models.
- **NumPy** provides a wide range of functions and methods for manipulating arrays of data, such as reshaping, transposing, and concatenating arrays. This can be useful for many different tasks, such as preprocessing data for machine learning models or performing mathematical operations on large datasets.

SciPy

What is SciPy?

SciPy is a popular Python library for scientific computing. It provides a wide range of numerical algorithms and functions for scientific and engineering applications. **SciPy** builds on the **NumPy** library, which provides data structures and functions for working with large arrays and matrices of data, and adds a wide range of algorithms and functions for scientific computing tasks.

SciPy includes many different submodules that cover a wide range of scientific computing tasks. For example, the **scipy.optimize submodule** includes functions for optimizing parameters in a model or function, the **scipy.interpolate submodule** includes functions for interpolating data, and the **scipy.integrate submodule** includes functions for numerically integrating functions.

In addition to these submodules, **SciPy** also provides a number of high-level functions and classes that make it easy to perform common scientific computing tasks in Python. For example, the **scipy.signal module** includes functions for working with signals, such as filtering and convolution, and the scipy.sparse module includes functions for working with sparse matrices.

Overall, **SciPy** is a powerful and versatile library for scientific computing in Python, and it is widely used in many different fields, such as physics, engineering, and data science. For more information, you can check out the **SciPy** documentation.

How can I get started with SciPy?

To get started with **SciPy**, you will first need to install the library on your system. You can install **SciPy** using **pip**, which is the Python package manager. Run the following command:

```
pip install scipy
```

This will install **SciPy** and all its dependencies on your system. Once **SciPy** is installed, you can start using it in your Python programs.

To use **SciPy** in a Python program, you will first need to import the library. You can import the entire **SciPy** library using the following code:

```
import scipy
```

Alternatively, you can import only the specific submodules or functions you need, which can make your code more efficient and easier to read. For example, if you only need the optimize submodule, you can import it like this:

```
from scipy import optimize
```

Once you have imported the **SciPy** library or submodule you need, you can start using the functions and classes it provides in your code. For example, if you have imported the optimize submodule, you can use the **minimize()** function to find the minimum of a function. Here's an example:

```
# Import the optimize submodule
```

```
from scipy import optimize


# Define a function to minimize

def my_function(x):

    return x**2


# Use the minimize() function to find the minimum of the function

result = optimize.minimize(my_function)

print(result)
```

This code will find the minimum of the function **my_function()** and print the result to the console.

What are some good resources for learning how to use SciPy?

There are many good resources available for learning how to use SciPy, including online tutorials, books, and the official SciPy documentation.

One good place to start learning about **SciPy** is the official **SciPy** tutorials, which provide a series of **Jupyter notebooks** that cover the basics of using **SciPy** for scientific computing. The tutorials cover a wide range of topics, including optimization, interpolation, signal processing, and solving differential equations.

Also, the official **SciPy** documentation is an excellent resource for learning about the different functions and classes that **SciPy** provides. The documentation includes detailed descriptions of all the functions and classes in **SciPy**, as well as examples of how to use them in your own code.

What are some advanced uses of SciPy?

SciPy is a popular Python library for scientific computing, and there are many advanced uses for the library. Here are a few examples of advanced uses for SciPy:

- Optimization, functions for optimizing parameters in a model or function. This can be useful for many applications, such as fitting a curve to data, minimizing the error of a machine learning model, or finding the maximum of a function.
- Interpolation, functions for interpolating data. This can be useful for many tasks, such as filling in missing data points, smoothing noisy data, or estimating values at intermediate points.
- Signal processing, functions for working with signals, such as filtering, convolution, and correlation. This can be useful for many applications, such as filtering noise from a signal, detecting patterns in data, or estimating the frequency of a signal.
- Differential equations, functions for numerically solving differential equations. This can be useful for many applications, such as modeling physical systems, simulating systems over time, or analyzing the behavior of systems that change over time.

Pandas

What is Pandas?

Pandas is a popular Python library for working with data. It provides a set of data structures and functions that make it easy to manipulate, analyze, and visualize large and complex datasets.

Pandas builds on the **NumPy** library, which provides data structures and functions for working with large arrays and matrices of data. **Pandas** adds a higher-level set of abstractions and functions that make it easier to work with tabular, structured data, such as data that is stored in tables or spreadsheets.

Pandas includes two main data structures for working with data: the **Series** and the **DataFrame**. A **Series** is a one-dimensional array of data with an associated index, and a **DataFrame** is a two-dimensional array of data with row and column labels. These data structures make it easy to store and manipulate large datasets in a consistent and organized way.

In addition to these data structures, **Pandas** also includes a wide range of functions and methods for performing common data manipulation tasks, such as selecting and filtering data, merging and joining datasets, and grouping and summarizing data. Pandas also integrates seamlessly with other scientific computing libraries, such as **Matplotlib** and **SciPy**, which makes it easy to create powerful visualizations and perform complex analyses of your data.

Pandas is a powerful and versatile library for working with data in Python, and it is widely used in many different fields, such as data science, finance, and statistics. For more information, you can check out the Pandas documentation.

How do I get started with Pandas?

To get started with Pandas, you will first need to install the library on your system.

```
pip install pandas
```

This will install **Pandas** and all its dependencies on your system. Once **Pandas** is installed, you can start using it in your Python programs. You know the drill!

To use Pandas in a Python program, you will first need to import the library. You can do this using the following code:

```
import pandas as pd
```

This imports the **pandas** library and gives it the alias pd. You can then use the functions and classes in the **pandas** library to work with data in your programs.

How would I use Pandas to work with a CSV?

To use **Pandas** to work with a **CSV** (**Comma-separated values**) file, you can use the **pandas.read_csv()** function to load the data from the file into a **DataFrame**. Here's an example of how you can use this function to load a **CSV** file and access the data it contains:

```
import pandas as pd

# Load the CSV file into a DataFrame

df = pd.read_csv("my_data.csv")


# Print the first five rows of the DataFrame

print(df.head())


# Access the values in a specific column

values = df["my_column"]


# Access the values in a specific row

values = df.loc[0]
```

```
# Access a specific value in the DataFrame

value = df.loc[0, "my_column"]
```

This code loads the **CSV** file into a **DataFrame**, prints the first five rows of the **DataFrame**, and then shows how to access specific values in the **DataFrame**. Once you have the data in a **DataFrame**, you can use the other functions and methods in the pandas library to perform operations on the data, such as selecting and filtering data, grouping and aggregating data, and visualizing the data.

What is a CSV?

A **CSV** (**Comma-separated values**) file is a type of text file that stores tabular data in a simple, human-readable format. **CSV** files are commonly used for storing data in a tabular format, such as data from a database or a spreadsheet program.

A **CSV** file consists of a series of lines, each of which contains a row of data. The data in each row is separated by a comma, which is why the format is called "comma-separated values". For example, a **CSV** file might look like this:

```
name,age,height

Ben Kenobi,38,173

Anakin Skywalker,27,182
```

In this example, the **CSV** file contains three rows of data, each of which has three values. The first row contains the names of the columns, and the subsequent rows contain the data for each row.

CSV files are simple and easy to read and write, which makes them a popular format for storing and exchanging data. Many different programs and languages support reading and writing **CSV** files, including Python.

Chapter Review

- Is Python good for complex math operations?
- What is NumPy?
- What is SciPy?
- What is Pandas?
- What is a CSV file?

Chapter 17
Machine Learning

What is machine learning?

Machine learning is a type of artificial intelligence that allows software applications to become more accurate in predicting outcomes without being explicitly programmed. It is a method of teaching computers to learn and make decisions based on data, without being explicitly programmed to perform a specific task. **Machine learning** algorithms use statistical methods to enable computers to improve their performance on a specific task over time. This allows the computer to automatically improve its performance on that task by learning from the data.

Is Python good for machine learning / AI?

Yes, Python is a good language for **machine learning** or **AI**. It is a high-level, interpreted language that is easy to read and write, and it has a large and active community of users who contribute a variety of useful libraries and frameworks for machine learning. Python also has a number of powerful libraries for **numerical computing**, such as **NumPy** and **Pandas**, which are useful for working with data in machine learning applications. Additionally, many popular machine learning frameworks, such as **TensorFlow** and **PyTorch**, are written in Python, which makes it a convenient language to use for developing machine learning models. Overall, Python's simplicity, flexibility, and strong support for data science and machine learning make it a good choice for many applications in this field.

What is Tensorflow?

TensorFlow is an open-source software library for machine learning and artificial intelligence. It was developed by Google and is used by many companies, organizations, and researchers worldwide for a wide range of tasks, such as training and deploying **machine learning** models, natural language processing, and image and video analysis.

TensorFlow is based on the concept of tensors, which are multi-dimensional arrays of data that represent the input, output, and intermediate computations of a machine learning model. **TensorFlow** provides a set of APIs and libraries that allow you to define and manipulate tensors, and to apply various mathematical operations and transformations to them, such as linear algebra, calculus, and probability.

TensorFlow also provides a runtime environment that allows you to execute tensor computations on various hardware platforms, such as CPUs, GPUs, and TPUs, and to optimize and parallelize them for maximum performance and efficiency. **TensorFlow** also provides tools and libraries for training, evaluating, and deploying **machine learning** models, and for exporting them to various formats and platforms, such as **TensorFlow Lite** for mobile and embedded devices, and **TensorFlow.js** for web browsers.

TensorFlow is a powerful and widely-used library for machine learning and artificial intelligence that provides a comprehensive and flexible set of tools and features for building and deploying **machine learning** models with Python and other programming languages.

How do I get started with Tensorflow using Python?

To get started with **TensorFlow** using Python, you can follow these steps:

1. Make sure you have Python installed on your computer.
2. Install **TensorFlow** by following the instructions on the TensorFlow website. TensorFlow can be installed using **pip**, which is the Python package manager, or using **conda**, which is the package manager for the **Anaconda** Python distribution.
3. Once **TensorFlow** is installed, you can start using it by importing it in your Python code. You can do this by adding the following line at the top of your Python file:

```
import tensorflow as tf
```

4. To verify that **TensorFlow** is working correctly, you can try running a simple example, such as creating a constant tensor and printing its value:

```
x = tf.constant([[1, 2], [3, 4]])

print(x)
```

5. If the code runs without any errors and prints the expected output, you have successfully installed and started using **TensorFlow**.

To learn more about **TensorFlow** and how to use it for building and training **machine learning** models, you can refer to the official **TensorFlow** documentation, which includes tutorials and other useful resources. Additionally, there are many online tutorials and courses that can help you learn TensorFlow and apply it to solve real-world problems.

Where can I find the Tensorflow documentation?

You can find the official **TensorFlow** documentation at https://www.tensorflow.org/docs. The documentation includes detailed information about the **TensorFlow API**, tutorials, and examples to help you get started with **TensorFlow** and understand how to use it for building and training **machine learning** models. It also includes information about the different features and capabilities of **TensorFlow**, such as support for distributed training and integration with other popular Python libraries for data science and machine learning. The documentation includes resources for more advanced users.

What is PyTorch?

PyTorch is an open-source machine learning library for Python. It is developed by Facebook's **artificial intelligence** research group, and is used for applications such as natural language processing and computer vision. **PyTorch** provides a flexible, high-performance environment for building and training deep learning models, and it is widely used by researchers and practitioners in the field of **machine learning**. **PyTorch** is known for its simplicity and flexibility, as well as its support for dynamic computation graphs, which allow for more efficient and flexible modeling of complex **neural networks**. It also integrates well with other popular Python libraries for data science and machine learning, such as **NumPy** and **Pandas**.

How do I get started with PyTorch?

To get started with **PyTorch**, you can follow these steps:

1. Make sure you have Python installed on your computer. If you don't have Python, you can download it from the official Python website.
2. Install **PyTorch** on your computer by following the instructions on the PyTorch website. You can choose to install PyTorch using **pip** or you can install it using **conda**

3. Once **PyTorch** is installed, you can start using it by importing it in your Python code

```
import torch
```

4. To verify that **PyTorch** is working correctly, you can try running a simple example, such as creating a tensor (a multidimensional array) and performing some operations on it:

```
x = torch.ones(3, 3)

y = x + 2

z = y * y * 3

print(z)
```

5. If the code runs with no errors and prints the expected output, you have successfully installed and started using **PyTorch**.

To learn more about **PyTorch** and how to use it for building and training **machine learning** models, you can refer to the official **PyTorch** documentation which includes tutorials and other helpful resources. There are many online tutorials and courses that can help you learn **PyTorch** and apply it to real-world problems.

Where can I read the official PyTorch documentation?

You can read the official **PyTorch** documentation at https://pytorch.org/docs/. The documentation includes tons of information on the **PyTorch** API including tutorials and examples to help you get started. It will also show you how to use it for building and training **machine learning** models.

Is Tensorflow better than PyTorch?

It is difficult to say definitively whether **TensorFlow** or **PyTorch** is better, as both are powerful and widely-used libraries for deep learning. **TensorFlow** and **PyTorch** have many similarities, such as their support for dynamic computation graphs, which allow for efficient and flexible modeling of complex neural networks. However, they also have some important differences, such as the way they handle memory and the level of flexibility and control they offer to users. Ultimately, the choice between **TensorFlow** and **PyTorch** will depend on the specific needs and preferences of the user. Some users may prefer TensorFlow for its strong support for production and deployment, while others may prefer **PyTorch** for its simplicity and flexibility. It is also worth noting that both **TensorFlow** and **PyTorch** are constantly evolving, with new features and improvements being added regularly, so the relative strengths and weaknesses of the two libraries may change over time.

What is Stable Diffusion?

Stable Diffusion is a text-to-image **AI** model that can create many different image types and perform lots of types of image processing based on given text prompts. **Stable Diffusion** can be run locally with Python or run in one of many online interfaces. Python can be used to script and manipulate the image creation as not only can it call the **Stable Diffusion** Python methods used to generate images, but it can also script the generation using traditional control structures like loops and conditional statements.

An image generated entirely using Stable Diffusion AI

Python and **Stable Diffusion** are a match made in heaven. Learn more at https://stability.ai/

Chapter Review

- What is machine learning?
- Is machine learning the same as AI?
- What is TensorFlow?
- What is PyTorch?
- What is Stable Diffusion?

Glossary

__doc__
A method that provides documentation for an object

__getitem__
A method to get an item from an invoked instance

__init__
A method to initialize an object

__iter__
A method that returns an iterator object that goes through all the elements in the object

__str__
A method that returns a string representing the object

.py
The file extension for Python code files

@staticmethod
A decorator marking a method as static

__**kwargs__
A dictionary of key-value arguments passed to a Python program

__*args__
A list of arguments passed to a Python program

__#__
The symbol used to start a comment line

__%__
The modulo operator which returns the remainder of a division operation

__==__
The equality operator which checks if two things are equal

__abstract__
A blueprint for a method or class that doesn’t have a function otherwise

__abstract base class__
A blueprint for a class

__AI__
Artificial intelligence

alias
Using a name in place of another name, for example giving an imported module a shorter name

Anaconda
A package manager for Python

and
A keyword used to compare if two things are both true

Android
A mobile phone operating system by Google

annotations
Decorators added to code that describe more about its properties

API
Application Programmer Interface. A contract for how computer programs can communicate

apt
A package management system on some versions of the Linux operating system

argument
A value passed into a function

Arithmetic operators
Operators that perform math, like + for addition

array
A data structure that holds a group of items in slots, like an egg carton

artificial intelligence
A term for a computer program that seems intelligent

as
A keyword to create an alias

ASC
A SQL keyword for ascended sorting

assignment operator
The operator to give a variable its value, =

attribute
A property or variable within an object

automated tests
Tests that run automatically to test your program

bar()
A Matplotlib function to make a bar chart

BeeWare
A multiplatform library for exporting Python apps to different operating systems

bool
Short for Boolean

bool()
A conversion function to turn something into a Boolean

Boolean
A true or false value

break
A keyword to exit a loop early

Buildozer
A tool for creating application packages

C
A programming language made in the 1970s

C#
A high-level language designed by Microsoft

C++
A “sequel” language to C first appearing in 1985

call
Making a function execute

casting
Turning a variable into a different data type

child classes
An object that has inherited traits and methods from a parent class

class
A definition for an object with defined properties and methods

class attributes
The properties inside a class definition

class inheritance
Where a child class can take on the properties and methods from a parent class

CLI
A Command Line Interface such as a terminal or command prompt

close()
A function for closing the reading or writing of a file

Code debugging
Working through problems in your code to find how to best fix it

codec
An algorithm that decodes a video or audio stream

columns
Fields in a SQL row

Comma-separated values
A type of file format where values are separated between comma characters

command prompt
The terminal command line interface program on Windows

command-line interface
A text-based interface for interacting with a computer

comment
A line in a code file describing what the code does without changing its functionality

commit()
A function to commit database changes to memory

Comparison operators
Operators to compare two things, like less than (<)

compiler
A program that turns code into another form, usually to make it faster to execute

composite
Overlaying two images on top of each other

concat()
A function for concatenating Pandas objects

concatenate
To combine together, like “a” + “bc” = “abc”

conda
The executable name of the Anaconda package manager

constant variable
An immutable variable that does not change

Cordova
A cross platform framework for making apps

cos()
Performs the trigonometric cosine operation

CREATE TABLE
A SQL command to create a Table

CSV
A Comma Separated Value file

cv2 module
The import module for the OpenCV library

Cython
A superset of Python that supports C functions

data structure
A way to store and structure data that can help coders organize their data

data type
The type of a variable such as integer or string

database
A system of storing structured data

datetime
An object for storing dates and times

datetime module
The module for importing datetime objects

DBMS
Short for DataBase Management System

Debian
A flavor of the Linux operating system

debug
Going through code to find and fix problems

debugger

A tool that can help you debug code

decorator
An annotation on a method or class to describe properties about how it should function

def
The Python keyword for defining a function

default argument
A value supplied to be the value an argument will have if its function isn’t called with an overriding value

DELETE
A SQL command to delete a row or set of rows

DESC
A SQL keyword for descended sorting

desktop applications
Applications that typically run on a desktop computer or desktop operating system

dictionary
A data structure that holds keys that map to values

diff()
A Pandas function that can return the difference between a set of values

Django
A framework for making web apps in Python

do-while loops
A loop that executes once before deciding if it should repeat

docstring
A string added to a class or method to help other coders know how to use it or what its function is

dunder method
A special method type in Python for providing object properties

elif
"Else if", a way in an if statement to add another conditional branch

else
A keyword in an if statement for code to execute if the first condition isn't true

encapsulation
A term for how variables can be wrapped together

in one unit

equality operator
The operator for checking if two things are equal,
==

except
The latter half of a try/except structure, code that
will execute if the try block had an error

exception
An error that occurs while running code

Express
A framework for writing web apps

expressions
A coding statement that can be evaluated to get
a value

extended
When an object has extended another object, as in
became a child of a parent class

extends
A term related to denoting a parent class of a child

class

f
A keyword used when formatting strings

factory method
A function pattern that creates new objects from inputs

falsy
Objects that will default to false if evaluated as booleans

Fedora
A flavor of the Linux operating system

ffmpeg
A library for editing video and audio

files
Computer documents that can be of various types

float
A number data type with decimals

for
The keyword to start a for loop

for loop
A loop that will generally run a certain number of times before stopping

frameworks
Code libraries that help a developer make a certain type of app

full-stack
A term referring to the whole stack of computer software, from back-end servers to front end apps

function
A code block that takes in arguments and can return a value

function arguments
The values passed into a function which it will perform operations with

function call
The act of calling a function to get a value

function declaration
The code that defines a function, what arguments

it takes in, and what it does

game development framework
A framework for helping a developer make a game

generic types
The ability to make a function that doesn’t know what type it will take in

GET
A HTTP method for getting data from a web server endpoint using a url and query strings

global scope
Variables that can be accessed anywhere in a program

graphical user interface
A visual interface to an app typically with widgets like buttons and labels

grid layout manager
A Tkinter layout manager type

GUI
A graphical user interface

hist()
A Matplotlib function to make a histogram

Homebrew
A package manager to install new packages from a CLI

HTTP
Hypertext Transfer Protocol

Hypertext Transfer Protocol
A method of getting data over the internet in request and response calls

IDE
A program that assists a developer in writing code

Identity operators
Keywords like “is” that help determine the identity or type of a variable

IDLE
An IDE for Python

if
The keyword used to start an if statement

if statement
A block of code that will only execute if the given condition is true

if-elif-else statements
A coding structure with branches of code that will execute under given conditions

immutable
Unable to be changed directly

import
Bringing in external code to be used and referenced in a file

in
A keyword that checks inclusion of one variable in a larger structure, like a list

inheritance
Receiving methods and properties from a parent class

INNER
A join type in SQL

INSERT

A SQL keyword to add a row into the database

instance
A specific instantiation of a variable

int
A variable type of just whole numbers, short for integer

integer
A whole number, represented by the variable type int

Integrated Development Environment
An IDE, a program that assists developers in writing code

interface
A blueprint that a class must follow, as in certain methods and variables it must contain

interpreter
A program that executes code of an interpreted language like Python

iOS
A mobile phone operating system by Apple

is operator
An operator that checks if a variable is of a certain type

iterable
Something that can be iterated over to get successive elements, like each element in a list

Jam
The best executive personal assistant who ever did live

JavaScript
A flexible language used mostly for logic on web pages

JavaScript Object Notation
A data format similar to the object declaration format used in JavaScript

JIT
Just In Time, a strategy for compiling repeated interpreted code into machine code

JOIN
A SQL keyword for combining two tables in a query

JSON
See JavaScript Object Notation

json module
A Python module that helps you work with JSON

json.dumps()
A function that converts an object to a JSON string

json.loads()
A function that converts a JSON string to an object

Jupyter notebooks
A web platform for interacting with code and other visualizations

just-in-time
A compilation strategy that compiles repeated interpreted code into machine code

key
The key from a dictionary that defines where the value will live

key-value pair
A key that maps to a value in a dictionary, like a word to its definition

lambda function
A lightweight function defined in place without a more formal function definition, that can be passed as an argument and used by other functions

layout manager
A class in Tkinter that defines how widgets will be laid out

LEFT
A type of JOIN in SQL

library
A set of code that has a purpose, that helps developers write their code

librosa
A Python package for music and audio

linked list
A list where each item has a reference to the next item in the list

Linux
A popular operating system used in phones and desktops, and much more

list
A sequential ordering of objects

local scope
Items that only can be accessed inside their current code block

localhost
An alias given to the web address of the user’s machine

log
A function to get the logarithm of a number

Logical operators
Keywords like “and” and “or” that help write logic statements

loop
A code structure that can execute multiple times

ls
A Unix or Linux command for listing the files and folders in a folder

Mac
A computer by Apple

machine code
Bytes that can be ran by a processor that represent instructions

machine learning
Algorithms for helping computer programs become specialized at a given task

macOS
The operating system running on Mac computers

map()
A function that maps a list of one type to a list of another

math module
A Python module with helpful math functions

math.pi
A variable in Python representing pi

Matplotlib
A powerful Python library for charting data

max
A Python function for getting the higher of two numbers

method
A function of a class that can be executed

MIDI
A type of audio format

min
A Python function to get the lower of two numbers

module
A package that can be imported to provide more functionality in your code

modulo
An operator for returning the remainder of a division operation

MongoDB
A popular NoSQL database

moviepy
A Python library for editing videos

music21 library
A Python library for creating music

mutable

Something that can be changed

MySQL
A popular relational database framework

nano
A Linux and Unix utility for editing text documents in a terminal

networking
Sharing data between computers

neural networks
A form of machine learning that seeks to structure data similar to a brain

newline
A character that moves input or output to the next line

Node.js
A framework for running JavaScript for a wide variety of applications, like a web server

None
A keyword in Python representing that a variable

doesn’t have a usable value

NoSQL
A type of database named such because it doesn’t use SQL like many databases do

null
A keyword in many programming languages for denoting that a variable has no value

Numba
A high-performance Python compiler

NumPy
A Python library for a wide range of numerical operations

object
A data type with properties and methods

Object Oriented Programming
A programming paradigm revolving around the creation, inheritance, and use of objects

OOP
Acronym for Object Oriented Programming

open source
A library is open source if it is free to use and modify by anyone

OpenCV
A popular library for image processing

operating system
Software that lets a user operate a computer

operator
A character that represents a mathematical or logical action

optional arguments
Function arguments that don’t have to be provided for the function to run

or
A logical operator that is true if at least one of two values is true

Oracle
A software company that owns Java as well as many other software products

ORDER BY
A SQL keyword that tells a query what field is should

sort the results by

os module
A Python module for interfacing with operating system functions

os.remove()
A function in Python for deleting a file

out of bounds
Trying to access an array or list in an index that is outside how many items the array has available

OUTER
A type of JOIN in SQL

overrides
A term for when a child class makes new functionality for a method or property

pack layout manager
A layout manager type in Tkinter

package
A collection of code that can be imported to add functionality to your code

Pandas

A Python library for dana analysis

parent class
A class that a child class has inherited from

PEP 8
The style guide standards for writing Python code

pie()
A function for making a pie chart in Matplotlib

PIL
The Python Imaging Library

Pillow
A fork of the PIL library with more features

pip
A popular package manager and installer for Python

place layout manager
A Tkinter layout manager type

polymorphism
The concept of using a class as one of its super
(parent) types

port
A channel for networking traffic

positional arguments
Arguments in a function that must be placed in a specific order

POST
An HTTP request method where data is held in the body of the request and not in the url

pow()
A function for calculating exponents

pretty_midi library
A library for making MIDI music

primitive
The lowest level data types of a language, like int or boolean

print
Placing a message into a terminal or some other location that can receive text

private attributes
Class attributes that can only be read or accessed by

the class that contains them

public attributes
Class attributes that can be read and accessed by anyone

PyCharm
A popular Python IDE

pymongo
A Python library for interfacing with MongoDB

pytest
A Python library that helps with testing

python
The command used to run a Python code file with the currently installed Python version

Python 2
The previous major version of Python. It is no longer supported

Python 3
The current major version of Python!

Python Imaging Library
PIL, a Python library for processing images

python3
The command used to run a Python code file with Python 3

Pythonista
An environment for writing Python code on iOS

PyTorch
A machine learning framework for Python based on Torch

Qt
A framework for making GUI applications

query
A SQL command for performing operations on a database

queue
A data structure where the earliest items added will be the first to come out

randint()
A function to get a random integer

random module
A Python module for making random numbers

range()
A function for getting a range of numbers depending on the arguments provided

re module
The Python module for regular expressions

read()
The function to open a file for reading

reading mode
A file opening mode that denotes the file is able to be read from

Red Hat
A flavor of the Linux operating system

Regex
A ReGular EXpression, a way to match patterns on a string that will return substrings

regular expression
See Regex above

relational database
A database type where Tables hold Rows of data, whose fields are separated into columns

required arguments
Arguments of a function that must be provided for the function to run

resize()
A method to resize an image using Pillow

return
A keyword that tells a function to stop executing and give a value back to the caller

reversed()
A function that reverses a string

round()
A function to round a float to a specified amount of decimal places or to a whole number

rows
Entries inside of a relational database

Ruby
A scripting language released in 1995

runtime error
An error that occurs while a program is running

runtime exception
An exception that is thrown while a program is running

scatter()
A Matplotlib function for making a scatter plot

SciPy
A popular Python framework for scientific calculations

scope
Areas where variables can live in a program and where those variables can be accessed from

scope nesting
Making a variable in an inner scope with the same name as one in the outer scope

SELECT
A SQL keyword to get data from the database

self
A keyword referring to the current Python instance that the code is being written in

server socket

A networking pipeline for client and server interaction

set
Similar to a list, but every element is unique

shutil module
A Python package for working with files

sin()
The trigonometric sine function

slice
An operator that can take a list or a string and split it into smaller pieces

slicing
The act of using the slice operator to slice a list or string or other sliceable object

sorted()
A function to sort a list

Spyder
A Python IDE focused on data science and engineering

SQL
Structured Query Language, the language used most often to interface with a relational database

SQL Server
A server that performs operations on a SQL database

sqrt
A math function to get the square root of a number

Stable Diffusion
A machine learning platform that generates images from text prompts using machine learning

stack
A data structure that returns the most recently added item as the first one to leave

standard library
The functions and objects built into a programming language

standard output
The output to the terminal the CLI application is running in

statement
A sentence of programming code

static method
A function that is referenced from a class rather than an instance of that class

str.find()
A function that finds a substring in a string

str.rfind()
A function that finds the last instance of a substring in a string

str.split()
A function that splits a string into a list by a specified separator

str.upper()
A function that converts a string to all uppercase characters

str()
A function that converts a variable into a string

string
A sequence of characters

structured query language
SQL, a language used to interface with SQL relational databases

subclass
A child class that has inherited from a parent class

Sublime Text
A popular text editing application

submodule
A module that exists inside another module

subprocess module
A Python module for creating and running new processes

sum
A function to get the sum of a group of numbers

super()
A function call to the constructor of a class’ parent class

switch statement
A structure for providing varied code branches for

a group of listed conditions

SymPy
A Python library for various math operations

syntax
The required structure that a language must be typed in to be valid

syntax error
When the code entered does not match the required syntax for the language

tan()
The mathematical tangent function

TensorFlow
An open source machine learning library from Google

TensorFlow Lite
A more lightweight version of TensorFlow with less features

TensorFlow.js
A JavaScript port of TensorFlow

terminal
A command line, text-based window used to interface with a computer via CLI applications

test cases
Conditions that must be satisfied to ensure a piece of code is running correctly

thread
A single sequential flow of code inside a program

threading module
A Python module to create and work with threads

Tkinter
A Python framework for making GUI applications

Toga
A cross platform GUI framework for Python

trees
A data structure where values are organized on leaves and branches

trigonometric functions
Math functions used to calculate properties about triangles

True
A Python keyword denoting a logical statement to be true

truthy
A variable is truthy if when converted to a boolean, it is converted to a true value

try
The keyword to being a try/except block to capture errors that can occur during code execution

Try catch
The concept of running code and capturing errors that occur. In Python this is called “try except”

tuple
Similar to a list, but its contents may not be changed once created

type hints
Parts of code that give hints as to what data type a function will return, or what data type function arguments should accept

typecast
Casting a variable from one data type to another

Ubuntu
A popular and user-friendly Linux distribution

ufunc
A NumPy function that operates over a set of NumPy arrays

Union
A way to work with generic typing in Python

unit test
A small unit of work encapsulated into a test

Unix
An operating system. MacOS is based on this and Linux was created to be an open-source analog of this

UPDATE
A SQL keyword to change the values in a set of rows

variable
A spot in memory that holds a value. It can be assigned to, read from, and used in other expressions

vim
An old command line text editing program known widely for the difficulty exiting it

visibility
This refers to the scope of variables and where they can be accessed and by whom

visual debuggers
A debugging program that displays debugging info graphically to the developer

Visual Studio Code
A popular text editing program that can function as a lightweight IDE

VSC
Acronym for Visual Studio Code

web applications
Applications that run over the internet with calls made to them generating responses with data

web browsers
An application that can browse websites from servers

web frameworks
A library that can be used to network with other servers

web server
A computer that will give data back to the caller when a request is made to it

WHERE
A SQL statement denoting that there is a condition to the query being made

while
The Python keyword to make a while loop

while loops
A loop that will run until a specified condition is met

Windows
A popular operating system by Microsoft

with statement
A code block that uses an object inside of it and then automatically closes any resources that object may be using after the block is done

write()
A function for writing text into a file

writing mode

A mode of opening a file denoting it’s to be written to

About the Author

Nicholas Wilson has been programming for a very long time. He began his grand coding adventure at the age of 8 and has been enjoying and crying over it ever since.

After doing it all through school, he has worked at some of the largest companies in the world as a programmer before deciding that torturing students would be more entertaining. Now he still works in the industry and shares his knowledge with anyone willing to learn.

The picture above is of his beloved cat Swiffer, who may or may not be the true coding mastermind in the family. She can often be found on Nic’s keyboard, “helping”.