
HOW TO CODE IN
PYTHON
LISA TAGLIAFERRI

How To Code in Python 3

Lisa Tagliaferri

DigitalOcean, New York City, New York, USA

ISBN 978-0-9997730-1-7

About DigitalOcean

DigitalOcean is a cloud services platform delivering the simplicity developers love and businesses trust to run production applications at scale. It provides highly available, secure and scalable compute, storage and networking solutions that help developers build great software faster. Founded in 2012 with offices in New York and Cambridge, MA, DigitalOcean offers transparent and affordable pricing, an elegant user interface, and one of the largest libraries of open source resources available. For more information, please visit https://www.digitalocean.com or follow @digitalocean on Twitter.

Read this book online and receive server credit via https://do.co/python-book.

DigitalOcean Community Team

Director of Community: Etel Sverdlov

Technical Writers: Melissa Anderson, Brian Boucheron, Mark Drake, Justin Ellingwood, Katy Howard, Lisa Tagliaferri

Technical Editors: Brian Hogan, Hazel Virdó

How To Code in Python 3

21. Understanding Boolean Logic
22. Understanding Lists
23. How To Use List Methods
24. Understanding List Comprehensions
25. Understanding Tuples
26. Understanding Dictionaries
27. How To Import Modules
28. How To Write Modules
29. How To Write Conditional Statements
30. How To Construct While Loops
31. How To Construct For Loops
32. How To Use Break, Continue, and Pass Statements when Working with Loops
33. How To Define Functions
34. How To Use *args and **kwargs
35. How To Construct Classes and Define Objects
36. Understanding Class and Instance Variables
37. Understanding Inheritance
38. How To Apply Polymorphism to Classes
39. How To Use the Python Debugger
40. How To Debug Python with an Interactive Console
41. How To Use Logging
42. How To Port Python 2 Code to Python 3

Introduction

Why Learn To Code

Software and technology are becoming increasingly integrated into our everyday lives, allowing us to accomplish tasks, navigate to destinations, make purchases, and stay connected with friends. Because of how pervasive software now is to the human experience, it is important for all of us to learn some of the key foundational elements of computer programming. While some may choose to study computer science as part of their formal education, everyone can benefit from an understanding of algorithmic thinking and computational processes. Learning how the software that we use on a daily basis is made can allow us as end users to evaluate how and why these applications are developed, enabling us to think critically about these tools and how to improve them.

Just like any other product, computer programs are designed and developed by people who have unconscious biases, make errors, and may not be considering all aspects of a problem they are trying to solve. Though development teams may do thorough testing and work to create sophisticated and useful programs, they do not always meet the needs and expectations of all users. While not everyone needs to learn to code complex programs, learning how coding works can help shape the future of technology and increase the number of stakeholders, decision makers, and knowledge producers who can work to build better software for everyone.

Some of us may choose to solve challenging problems within the technology sector, but for those of us not working in computer science, a

programming background can still be a great asset to our professional fields. Computer programming provides many applications across domains, and can help us solve problems in specialities such as medicine, economics, sociology, history, and literature, to name a few. By integrating technology's methodologies into our own fields, we can leverage computational logic and software design and development practices in our work. When we synthesize knowledge across spheres and collaborate with people from different backgrounds, we can innovate in new, more inclusive ways that can enact meaningful impact across many communities.

Why Learn Python

Extremely versatile and popular among developers, Python is a good general-purpose language that can be used in a variety of applications. For those with an understanding of English, Python is a very human-readable programming language, allowing for quick comprehension. Because Python supports multiple styles including scripting and object-oriented programming, it is considered to be a multi-paradigm language that enables programmers to use the most suitable style to complete a project. Increasingly used in industry, Python offers a lot of potential for those who would like to begin coding while also being a good choice for those looking to pick up an additional programming language.

Learning the key concepts of Python can help you understand how programs work while also imparting foundational logic that can serve you in other domains. Understanding what Python and computer programming can offer you both as a user and as a developer is important as technology is further integrated into daily life.

As you work through this book, you will be able to increase your awareness of computer programming, improve your logical thinking, and eventually become a producer of software. Being able to create software that runs is a very rewarding endeavor, and can help you serve those around you by increasing their access and empowering them to become collaborators. The more communities involved in the creation of software development, the more communities there will be whose needs are served by software.

How To Use This Book

This book is designed to be used in a way that makes sense for you. While it is arranged to ramp up an emerging developer, do not be constrained by the order: feel free to move throughout the book in a way that makes sense for you. Once you are familiar with the concepts, you can continue to use the book as a source of reference.

If you use the book in the order it is laid out, you'll begin your exploration in Python by understanding the key differences between Python 3 and the previous versions of the language. From there, you'll set up a programming environment for your relevant local or server-based system, and begin by learning general Python code structure, syntax, and data types. Along the way, you'll gain a solid grounding in computational logic within Python, which can help you learn other programming languages. While the beginning of the book focuses on scripting in Python, the end of the book will take you through object-oriented coding in Python, which can make your code more modular, flexible, and complex without repetition. By the end of the book, you'll learn how to debug your Python code and finally how to port Python code across versions.

When you are done with the book, we encourage you to look at project-based tutorials to put your knowledge into play while creating projects that can help you solve problems. While you are working on these projects, you can continue to refer to the chapters in this book as reference material.

As part of your learning process and once you feel comfortable, we recommend that you contribute to an open-source project to improve programs and drive greater access via software and technical documentation pull requests or repository maintenance. Our community is bigger than just us and building software together can make sure that everyone has an opportunity to participate in the technology we use every day.

Python 2 vs Python 3: Practical Considerations

Python is an extremely readable and versatile programming language. With a name inspired by the British comedy group Monty Python, it was an important foundational goal of the Python development team to make the language fun to use. Easy to set up, and written in a relatively straightforward style with immediate feedback on errors, Python is a great choice for beginners.

As Python is a multiparadigm language — that is, it supports multiple programming styles including scripting and object-oriented — it is good for general purpose use. Increasingly used in industry by organizations such as United Space Alliance (NASA's main shuttle support contractor), and Industrial Light & Magic (the VFX and animation studio of Lucasfilm), Python offers a lot of potential for those looking to pick up an additional programming language.

Developed in the late 1980s and first published in 1991, Python was authored by Guido van Rossum, who is still very active in the community. Conceived as a successor to the ABC programming language, Python's first iteration already included exception handling, functions, and classes with inheritance. When an important Usenet newsgroup discussion forum called comp.lang.python was formed in 1994, Python's user base grew, paving the way for Python to become one of the most popular programming languages for open source development.

General Overview

Before looking into potential opportunities related to — and the key programmatic differences between — Python 2 and Python 3, let's take a look into the background of the more recent major releases of Python.

Python 2

Published in late 2000, Python 2 signalled a more transparent and inclusive language development process than earlier versions of Python with the implementation of PEP (Python Enhancement Proposal), a technical specification that either provides information to Python community members or describes a new feature of the language.

Additionally, Python 2 included many more programmatic features including a cycle-detecting garbage collector to automate memory management, increased Unicode support to standardize characters, and list comprehensions to create a list based on existing lists. As Python 2 continued to develop, more features were added, including unifying Python's types and classes into one hierarchy in Python version 2.2.

Python 3

Python 3 is regarded as the future of Python and is the version of the language that is currently in development. A major overhaul, Python 3 was released in late 2008 to address and amend intrinsic design flaws of previous versions of the language. The focus of Python 3 development was to clean up the codebase and remove redundancy, making it clear that there was only one way to perform a given task.

Major modifications to Python 3.0 included changing the `print` statement into a built-in function, improve the way integers are divided, and providing more Unicode support.

At first, Python 3 was slowly adopted due to the language not being backwards compatible with Python 2, requiring people to make a decision as to which version of the language to use. Additionally, many package libraries were only available for Python 2, but as the development team behind Python 3 has reiterated that there is an end of life for Python 2 support, more libraries have been ported to Python 3. The increased adoption of Python 3 can be shown by the number of Python packages that now provide Python 3 support, which at the time of writing includes 339 of the 360 most popular Python packages.

Python 2.7

Following the 2008 release of Python 3.0, Python 2.7 was published on July 3, 2010 and planned as the last of the 2.x releases. The intention behind Python 2.7 was to make it easier for Python 2.x users to port features over to Python 3 by providing some measure of compatibility between the two. This compatibility support included enhanced modules for version 2.7 like `unittest` to support test automation, `argparse` for parsing command-line options, and more convenient classes in `collections`.

Because of Python 2.7's unique position as a version in between the earlier iterations of Python 2 and Python 3.0, it has persisted as a very popular choice for programmers due to its compatibility with many robust libraries. When we talk about Python 2 today, we are typically referring to the Python 2.7 release as that is the most frequently used version.

Python 2.7, however, is considered to be a legacy language and its continued development, which today mostly consists of bug fixes, will cease completely in 2020.

Key Differences

While Python 2.7 and Python 3 share many similar capabilities, they should not be thought of as entirely interchangeable. Though you can write good code and useful programs in either version, it is worth understanding that there will be some considerable differences in code syntax and handling.

Below are a few examples, but you should keep in mind that you will likely encounter more syntactical differences as you continue to learn Python.

Print

In Python 2, `print` is treated as a statement instead of a function, which was a typical area of confusion as many other actions in Python require arguments inside of parentheses to execute. If you want your console to print out `Sammy the Shark is my favorite sea creature` in Python 2 you can do so with the following `print` statement:

```
print "Sammy the Shark is my favorite sea creature"
```

With Python 3, `print()` is now explicitly treated as a function, so to print out the same string above, you can do so simply and easily using the syntax of a function:

```
print("Sammy the Shark is my favorite sea creature")
```

This change made Python's syntax more consistent and also made it easier to change between different print functions. Conveniently, the

`print()` syntax is also backwards-compatible with Python 2.7, so your Python 3 `print()` functions can run in either version.

Division with Integers

In Python 2, any number that you type without decimals is treated as the programming type called integer. While at first glance this seems like an easy way to handle programming types, when you try to divide integers together sometimes you expect to get an answer with decimal places (called a float), as in:

```
5 / 2 = 2.5
```

However, in Python 2 integers were strongly typed and would not change to a float with decimal places even in cases when that would make intuitive sense.

When the two numbers on either side of the division `/` symbol are integers, Python 2 does floor division so that for the quotient `x` the number returned is the largest integer less than or equal to `x`. This means that when you write `5 / 2` to divide the two numbers, Python 2.7 returns the largest integer less than or equal to 2.5, in this case `2`:

```
a = 5 / 2
print a
```

Output

```
2
```

To override this, you could add decimal places as in `5.0 / 2.0` to get the expected answer `2.5`.

In Python 3, integer division became more intuitive, as in:

```
a = 5 / 2
print(a)
```

Output

```
2.5
```

You can still use `5.0 / 2.0` to return `2.5`, but if you want to do floor division you should use the Python 3 syntax of `//`, like this:

```
b = 5 // 2
print(b)
```

Output

```
2
```

This modification in Python 3 made dividing by integers much more intuitive and is a feature that is not backwards compatible with Python 2.7.

Unicode Support

When programming languages handle the string type — that is, a sequence of characters — they can do so in a few different ways so that computers can convert numbers to letters and other symbols.

Python 2 uses the ASCII alphabet by default, so when you type `"Hello, Sammy!"` Python 2 will handle the string as ASCII. Limited to a couple of hundred characters at best in various extended forms, ASCII is not a very flexible method for encoding characters, especially non-English characters.

To use the more versatile and robust Unicode character encoding, which supports over 128,000 characters across contemporary and historic scripts and symbol sets, you would have to type `u"Hello, Sammy!"`, with the `u` prefix standing for Unicode.

Python 3 uses Unicode by default, which saves programmers extra development time, and you can easily type and display many more characters directly into your program. Because Unicode supports greater linguistic character diversity as well as the display of emojis, using it as the default character encoding ensures that mobile devices around the world are readily supported in your development projects.

If you would like your Python 3 code to be backwards-compatible with Python 2, though, you can keep the `u` before your string.

Continued Development

The biggest difference between Python 3 and Python 2 is not a syntactical one, but the fact that Python 2.7 will lose continued support in 2020 and Python 3 will continue to be developed with more features and more bug fixes.

Recent developments have included formatted string literals, simpler customization of class creation, and a cleaner syntactical way to handle matrix multiplication.

Continued development of Python 3 means that developers can rely on having issues fixed in a timely manner, and programs can be more

effective with increased functionality being built in over time.

Additional Points to Consider

As someone starting Python as a new programmer, or an experienced programmer new to the Python language, you will want to consider what you are hoping to achieve in learning the language.

If you are hoping just to learn without a set project in mind, you will likely most want to take into account that Python 3 will continue to be supported and developed, while Python 2.7 will not.

If, however, you are planning to join an existing project, you will likely most want to see what version of Python the team is using, how a different version may interact with the legacy codebase, if the packages the project uses are supported in a different version, and what the implementation details of the project are.

If you are beginning a project that you have in mind, it would be worthwhile to investigate what packages are available to use and with which version of Python they are compatible. As noted above, though earlier versions of Python 3 had less compatibility with libraries built for versions of Python 2, many have ported over to Python 3 or are committed to doing so in the next four years.

Conclusion

Python is a versatile and well-documented programming language to learn, and whether you choose to work with Python 2 or Python 3, you will be able to work on exciting software projects.

Though there are several key differences, it is not too difficult to move from Python 3 to Python 2 with a few tweaks, and you will often find that Python 2.7 can easily run Python 3 code, especially when you are

starting out. You can learn more about this process by reading the tutorial How To Port Python 2 Code to Python 3.

It is important to keep in mind that as more developer and community attention focuses on Python 3, the language will become more refined and in-line with the evolving needs of programmers, and less support will be given to Python 2.7.

How To Install Python 3 and Set Up a Local Programming Environment on Ubuntu 16.04

This tutorial will get you up and running with a local Python 3 programming environment in Ubuntu 16.04.

Python is a versatile programming language that can be used for many different programming projects. First published in 1991 with a name inspired by the British comedy group Monty Python, the development team wanted to make Python a language that was fun to use. Easy to set up, and written in a relatively straightforward style with immediate feedback on errors, Python is a great choice for beginners and experienced developers alike. Python 3 is the most current version of the language and is considered to be the future of Python.

This tutorial will guide you through installing Python 3 on your local Linux machine and setting up a programming environment via the command line. This tutorial will explicitly cover the installation procedures for Ubuntu 16.04, but the general principles apply to any other distribution of Debian Linux.

Prerequisites

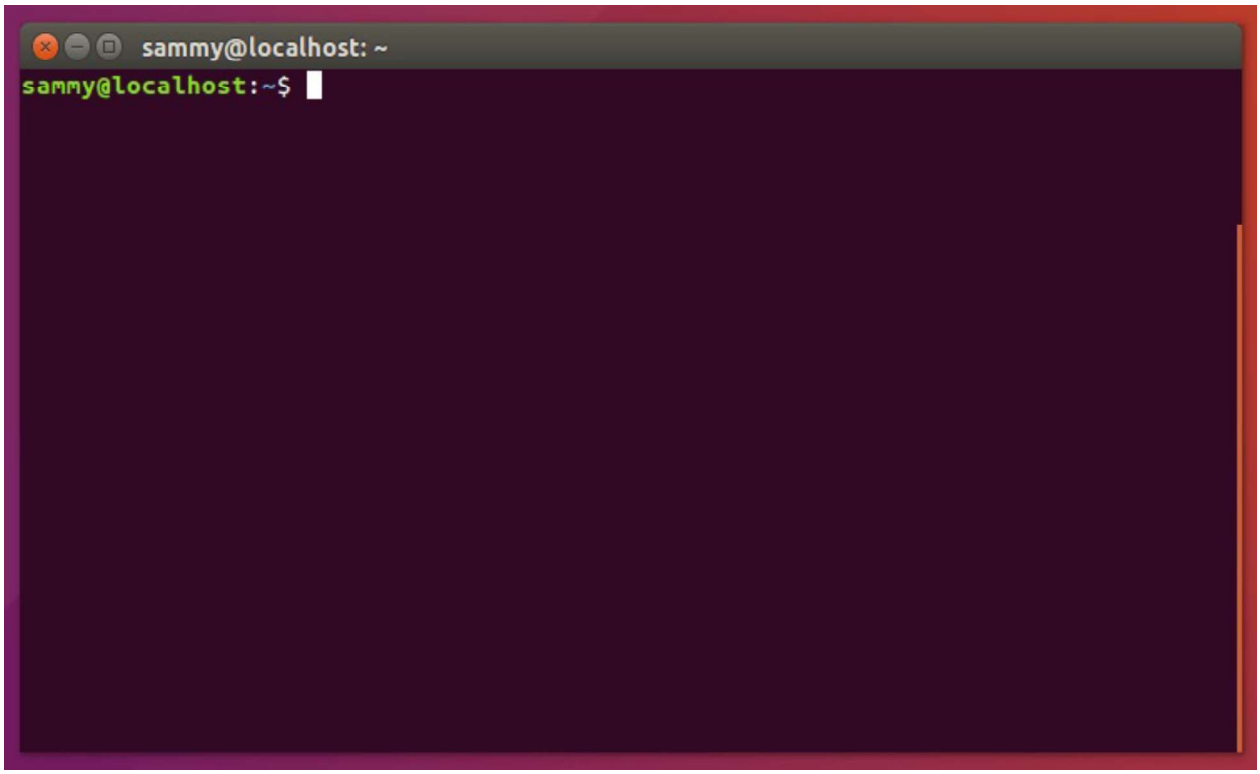
You will need a computer with Ubuntu 16.04 installed, as well as have administrative access to that machine and an internet connection.

Step 1 — Setting Up Python 3

We'll be completing our installation and setup on the command line, which is a non-graphical way to interact with your computer. That is,

instead of clicking on buttons, you'll be typing in text and receiving feedback from your computer through text as well. The command line, also known as a shell, can help you modify and automate many of the tasks you do on a computer every day, and is an essential tool for software developers. There are many terminal commands to learn that can enable you to do more powerful things. The article "An Introduction to the Linux Terminal" can get you better oriented with the terminal.

On Ubuntu 16.04, you can find the Terminal application by clicking on the Ubuntu icon in the upper-left hand corner of your screen and typing "terminal" into the search bar. Click on the Terminal application icon to open it. Alternatively, you can hit the `CTRL`, `ALT`, and `T` keys on your keyboard at the same time to open the Terminal application automatically.



Ubuntu Terminal

Ubuntu 16.04 ships with both Python 3 and Python 2 pre-installed. To make sure that our versions are up-to-date, let's update and upgrade the system with `apt-get`:

```
sudo apt-get update
sudo apt-get -y upgrade
```

The `-y` flag will confirm that we are agreeing for all items to be installed, but depending on your version of Linux, you may need to confirm additional prompts as your system updates and upgrades.

Once the process is complete, we can check the version of Python 3 that is installed in the system by typing:

```
python3 -V
```

You will receive output in the terminal window that will let you know the version number. The version number may vary, but it will look similar to this:

Output

```
Python 3.5.2
```

To manage software packages for Python, let's install pip:

```
sudo apt-get install -y python3-pip
```

A tool for use with Python, pip installs and manages programming packages we may want to use in our development projects. You can

install Python packages by typing:

```
pip3 install package_name
```

Here, `package_name` can refer to any Python package or library, such as Django for web development or NumPy for scientific computing. So if you would like to install NumPy, you can do so with the command `pip3 install numpy`.

There are a few more packages and development tools to install to ensure that we have a robust set-up for our programming environment:

```
sudo apt-get install build-essential libssl-dev
libffi-dev python-dev
```

Once Python is set up, and pip and other tools are installed, we can set up a virtual environment for our development projects.

Step 2 — Setting Up a Virtual Environment

Virtual environments enable you to have an isolated space on your computer for Python projects, ensuring that each of your projects can have its own set of dependencies that won't disrupt any of your other projects.

Setting up a programming environment provides us with greater control over our Python projects and over how different versions of packages are handled. This is especially important when working with third-party packages.

You can set up as many Python programming environments as you want. Each environment is basically a directory or folder in your

computer that has a few scripts in it to make it act as an environment.

We need to first install the venv module, part of the standard Python 3 library, so that we can create virtual environments. Let's install venv by typing:

```
sudo apt-get install -y python3-venv
```

With this installed, we are ready to create environments. Let's choose which directory we would like to put our Python programming environments in, or we can create a new directory with `mkdir`, as in:

```
mkdir environments
cd environments
```

Once you are in the directory where you would like the environments to live, you can create an environment by running the following command:

```
python3 -m venv my_env
```

Essentially, this sets up a new directory that contains a few items which we can view with the `ls` command:

```
ls my_env
```

Output

```
bin include lib lib64 pyvenv.cfg share
```

Together, these files work to make sure that your projects are isolated from the broader context of your local machine, so that system files and project files don't mix. This is good practice for version control and to ensure that each of your projects has access to the particular packages that it needs. Python Wheels, a built-package format for Python that can speed up your software production by reducing the number of times you need to compile, will be in the Ubuntu 16.04 `share` directory.

To use this environment, you need to activate it, which you can do by typing the following command that calls the activate script:

```
source my_env/bin/activate
```

Your prompt will now be prefixed with the name of your environment, in this case it is called **my_env**. Your prefix may look somewhat different, but the name of your environment in parentheses should be the first thing you see on your line:

```
(my_env) sammy@sammy:~/environments$
```

This prefix lets us know that the environment **my_env** is currently active, meaning that when we create programs here they will use only this particular environment's settings and packages.

Note: Within the virtual environment, you can use the command `python` instead of `python3`, and `pip` instead of `pip3` if you would prefer. If you use Python 3 on your machine outside of an environment, you will need to use the `python3` and `pip3` commands exclusively.

After following these steps, your virtual environment is ready to use.

Step 3 — Creating a Simple Program

Now that we have our virtual environment set up, let's create a simple "Hello, World!" program. This will make sure that our environment is working and gives us the opportunity to become more familiar with Python if we aren't already.

To do this, we'll open up a command-line text editor such as nano and create a new file:

```
(my_env) sammy@sammy:~/environments$ nano hello.py
```

Once the text file opens up in the terminal window we'll type out our program:

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

Exit nano by typing the `control` and `x` keys, and when prompted to save the file press `y`.

Once you exit out of nano and return to your shell, let's run the program:

```
(my_env) sammy@sammy:~/environments$python hello.py
```

The hello.py program that you just created should cause your terminal to produce the following output:

Output

```
Hello, World!
```

To leave the environment, simply type the command `deactivate` and you will return to your original directory.

Conclusion

Congratulations! At this point you have a Python 3 programming environment set up on your local Ubuntu machine and can begin a coding project!

To set up Python 3 on another computer, follow the local programming environment guides for Debian 8, CentOS 7, Windows 10, or macOS. You can also read about installing Python and setting up a programming environment on an Ubuntu 16.04 server, which is especially useful when working on development teams.

With your local machine ready for software development, you can continue to learn more about coding in Python by following "Understanding Data Types in Python 3" and "How To Use Variables in Python 3".

How To Install Python 3 and Set Up a Local Programming Environment on macOS

Python is a versatile programming language that can be used for many different programming projects. First published in 1991 with a name inspired by the British comedy group Monty Python, the development team wanted to make Python a language that was fun to use. Easy to set up, and written in a relatively straightforward style with immediate feedback on errors, Python is a great choice for beginners and experienced developers alike. Python 3 is the most current version of the language and is considered to be the future of Python.

This tutorial will guide you through installing Python 3 on your local macOS machine and setting up a programming environment via the command line.

Prerequisites

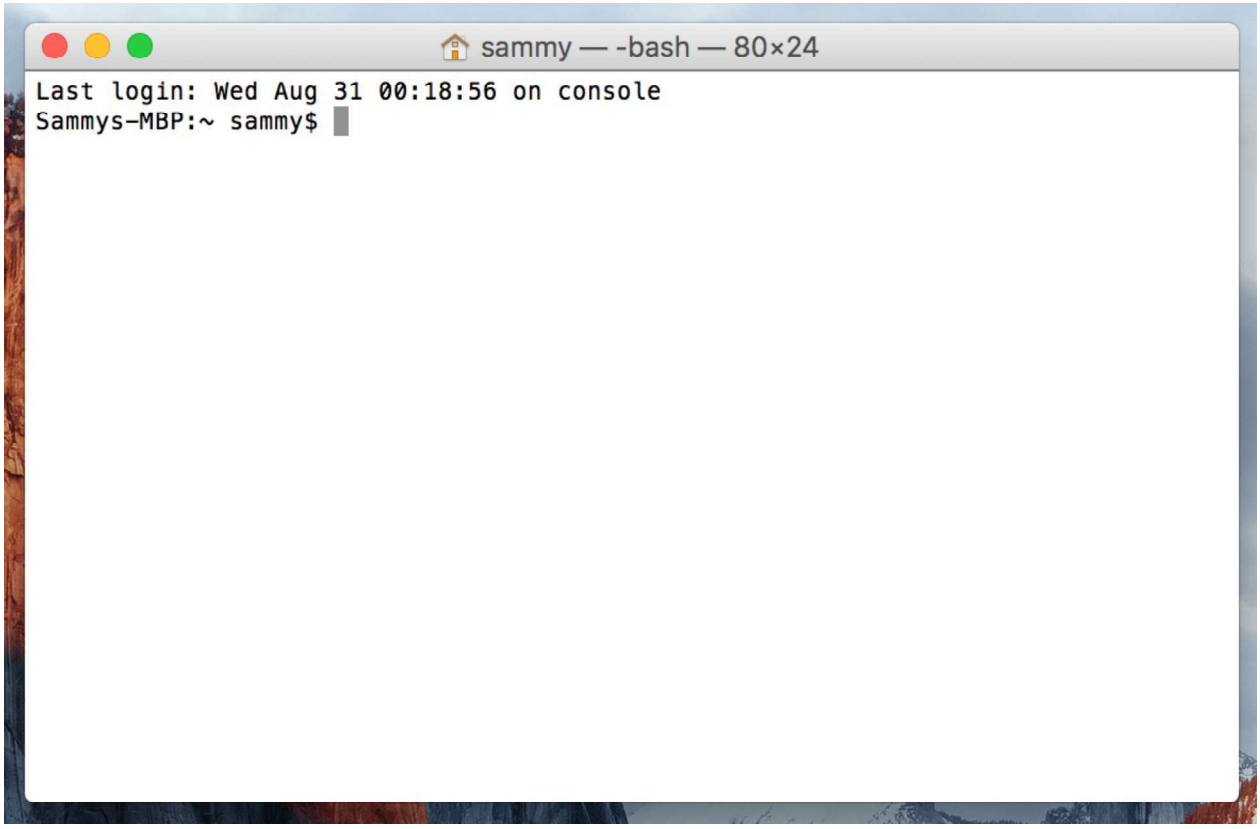
You will need a macOS computer with administrative access that is connected to the internet.

Step 1 — Opening Terminal

We'll be completing most of our installation and set up on the command line, which is a non-graphical way to interact with your computer. That is, instead of clicking on buttons, you'll be typing in text and receiving feedback from your computer through text as well. The command line, also known as a shell, can help you modify and automate many of the

tasks you do on a computer every day, and is an essential tool for software developers.

The macOS Terminal is an application you can use to access the command line interface. Like any other application, you can find it by going into Finder, navigating to the Applications folder, and then into the Utilities folder. From here, double-click the Terminal like any other application to open it up. Alternatively, you can use Spotlight by holding down the `command` and `spacebar` keys to find Terminal by typing it out in the box that appears.



macOS Terminal

There are many more Terminal commands to learn that can enable you to do more powerful things. The article "An Introduction to the Linux

Terminal" can get you better oriented with the Linux Terminal, which is similar to the macOS Terminal.

Step 2 — Installing Xcode

Xcode is an integrated development environment (IDE) that is comprised of software development tools for macOS. You may have Xcode installed already. To check, in your Terminal window, type:

```
xcode-select -p
```

If you receive the following output, then Xcode is installed:

Output

```
/Library/Developer/CommandLineTools
```

If you received an error, then in your web browser install Xcode from the App Store and accept the default options.

Once Xcode is installed, return to your Terminal window. Next, you'll need to install Xcode's separate Command Line Tools app, which you can do by typing:

```
xcode-select --install
```

At this point, Xcode and its Command Line Tools app are fully installed, and we are ready to install the package manager Homebrew.

Step 3 — Installing and Setting Up Homebrew

While the OS X Terminal has a lot of the functionality of Linux Terminals and other Unix systems, it does not ship with a good package manager. A package manager is a collection of software tools that work to automate installation processes that include initial software installation, upgrading and configuring of software, and removing software as needed. They keep installations in a central location and can maintain all software packages on the system in formats that are commonly used. Homebrew provides OS X with a free and open source software package managing system that simplifies the installation of software on OS X.

To install Homebrew, type this into your Terminal window:

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

Homebrew is made with Ruby, so it will be modifying your computer's Ruby path. The `curl` command pulls a script from the specified URL. This script will explain what it will do and then pauses the process to prompt you to confirm. This provides you with a lot of feedback on what the script is going to be doing to your system and gives you the opportunity to verify the process.

If you need to enter your password note that your keystrokes will not display in the Terminal window but they will be recorded, simply press the `return` key once you've entered your password. Otherwise press the letter `y` for "yes" whenever you are prompted to confirm the installation.

Let's walk through the flags that are associated with the `curl` command:

- The `-f` or `--fail` flag tells the Terminal window to give no HTML document output on server errors.
- The `-s` or `--silent` flag mutes `curl` so that it does not show the progress meter, and combined with the `-S` or `--show-error` flag it will ensure that `curl` shows an error message if it fails.
- The `-L` or `--location` flag will tell `curl` to redo the request to a new place if the server reports that the requested page has moved to a different location.

Once the installation process is complete, we'll put the Homebrew directory at the top of the `PATH` environment variable. This will ensure that Homebrew installations will be called over the tools that Mac OS X may select automatically that could run counter to the development environment we're creating.

You should create or open the `~/.bash_profile` file with the command-line text editor nano using the `nano` command:

```
nano ~/.bash_profile
```

Once the file opens up in the Terminal window, write the following:

```
export PATH=/usr/local/bin:$PATH
```

To save your changes, hold down the `control` key and the letter `o`, and when prompted press the `return` key. Now you can exit nano by holding the `control` key and the letter `x`.

For these changes to activate, in the Terminal window, type:

```
source ~/.bash_profile
```

Once you have done this, the changes you have made to the `PATH` environment variable will be effective.

We can make sure that Homebrew was successfully installed by typing:

```
brew doctor
```

If no updates are required at this time, the Terminal output will read:

Output

```
Your system is ready to brew.
```

Otherwise, you may get a warning to run another command such as `brew update` to ensure that your installation of Homebrew is up to date.

Once Homebrew is ready, you can install Python 3.

Step 4 — Installing Python 3

You can use Homebrew to search for everything you can install with the `brew search` command, but to provide us with a shorter list, let's instead search for just the available Python-related packages or modules:

```
brew search python
```

The Terminal will output a list of what you can install, like this:

Output

```
app-engine-python                micropython
python3
boost-python                     python
wxpython
gst-python                       python-markdown
zpython
homebrew/apache/mod_python
homebrew/versions/gst-python010
homebrew/python/python-dbus
Caskroom/cask/kk7ds-python-runtime
homebrew/python/vpython
Caskroom/cask/mysql-connector-python
```

Python 3 will be among the items on the list. Let's go ahead and install it:

```
brew install python3
```

The Terminal window will give you feedback regarding the installation process of Python 3, it may take a few minutes before installation is complete.

Along with Python 3, Homebrew will install pip, setuptools and wheel.

A tool for use with Python, we will use pip to install and manage programming packages we may want to use in our development projects. You can install Python packages by typing:

```
pip3 install package_name
```

Here, `package_name` can refer to any Python package or library, such as Django for web development or NumPy for scientific computing. So if you would like to install NumPy, you can do so with the command `pip3 install numpy`.

setuptools facilitates packaging Python projects, and wheel is a built-package format for Python that can speed up your software production by reducing the number of times you need to compile.

To check the version of Python 3 that you installed, you can type:

```
python3 --version
```

This will output the specific version of Python that is currently installed, which will by default be the most up-to-date stable version of Python 3 that is available.

To update your version of Python 3, you can first update Homebrew and then update Python:

```
brew update
brew upgrade python3
```

It is good practice to ensure that your version of Python is up-to-date.

Step 5 — Creating a Virtual Environment

Now that we have Xcode, Homebrew, and Python installed, we can go on to create our programming environment.

Virtual environments enable you to have an isolated space on your computer for Python projects, ensuring that each of your projects can have its own set of dependencies that won't disrupt any of your other projects.

Setting up a programming environment provides us with greater control over our Python projects and over how different versions of packages are handled. This is especially important when working with third-party packages.

You can set up as many Python programming environments as you would like. Each environment is basically a directory or folder in your computer that has a few scripts in it to make it act as an environment.

Choose which directory you would like to put your Python programming environments in, or create a new directory with `mkdir`, as in:

```
mkdir Environments
cd Environments
```

Once you are in the directory where you would like the environments to live, you can create an environment by running the following command:

```
python3.6 -m venv my_env
```

Essentially, this command creates a new directory (in this case called **my_env**) that contains a few items: - The `pyvenv.cfg` file points to the Python installation that you used to run the command. - The `lib` subdirectory contains a copy of the Python version and has a `site-`

`packages` subdirectory inside it that starts out empty but will eventually hold the relevant third-party modules that you install. - The `include` subdirectory compiles packages. - The `bin` subdirectory has a copy of the Python binary along with the activate shell script that is used to set up the environment.

Together, these files work to make sure that your projects are isolated from the broader context of your local machine, so that system files and project files don't mix. This is good practice for version control and to ensure that each of your projects has access to the particular packages that it needs.

To use this environment, you need to activate it, which you can do by typing the following command that calls the activate script:

```
source my_env/bin/activate
```

Your prompt will now be prefixed with the name of your environment, in this case it is called **my_env**:

```
(my_env) Sammys-MBP:~ sammy$
```

This prefix lets us know that the environment **my_env** is currently active, meaning that when we create programs here they will use only this particular environment's settings and packages.

Note: Within the virtual environment, you can use the command `python` instead of `python3`, and `pip` instead of `pip3` if you would prefer. If you use Python 3 on your machine outside of an environment, you'll need to use the `python3` and `pip3` commands exclusively, as `python` and `pip` will call an earlier version of Python.

After following these steps, your virtual environment is ready to use.

Step 6 — Creating a Simple Program

Now that we have our virtual environment set up, let's create a simple "Hello, World!" program. This will make sure that our environment is working and gives us the opportunity to become more familiar with Python if we aren't already.

To do this, we'll open up a command-line text editor such as nano and create a new file:

```
(my_env) Sammys-MBP:~ sammy$ nano hello.py
```

Once the text file opens up in Terminal we'll type out our program:

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

Exit nano by typing the `control` and `x` keys, and when prompted to save the file press `y`.

Once you exit out of nano and return to your shell, let's run the program:

```
(my_env) Sammys-MBP:~ sammy$ python hello.py
```

The hello.py program that you just created should cause Terminal to produce the following output:

Output

```
Hello, World!
```

To leave the environment, simply type the command `deactivate` and you'll return to your original directory.

Conclusion

Congratulations! At this point you have a Python 3 programming environment set up on your local Mac OS X machine and can begin a coding project!

To set up Python 3 on another computer, follow the local programming environment guides for Ubuntu 16.04, Debian 8, CentOS 7, or Windows 10. You can also read about installing Python and setting up a programming environment on an Ubuntu 16.04 server, which is especially useful when working on development teams.

With your local machine ready for software development, you can continue to learn more about coding in Python by following "Understanding Data Types in Python 3" and "How To Use Variables in Python 3".

How To Install Python 3 and Set Up a Local Programming Environment on Windows 10

Python is a versatile programming language that can be used for many different programming projects. First published in 1991 with a name inspired by the British comedy group Monty Python, the development team wanted to make Python a language that was fun to use. Easy to set up, and written in a relatively straightforward style with immediate feedback on errors, Python is a great choice for beginners and experienced developers alike. Python 3 is the most current version of the language and is considered to be the future of Python.

This tutorial will guide you through installing Python 3 on your local Windows 10 machine and setting up a programming environment via the command line.

Prerequisites

You will need a Windows 10 computer with administrative access that is connected to the internet.

Step 1 — Opening and Configuring PowerShell

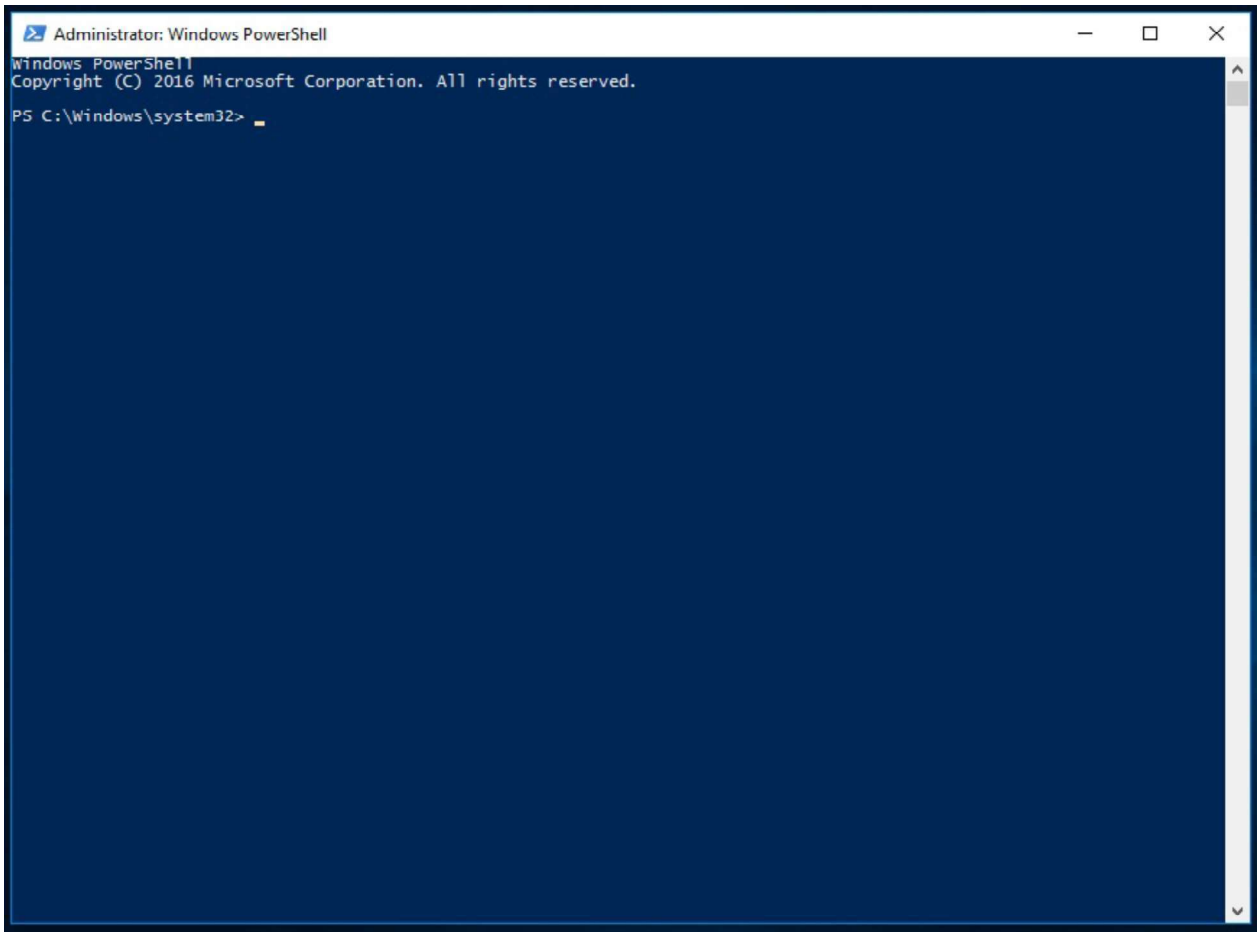
We'll be completing most of our installation and setup on a command-line interface, which is a non-graphical way to interact with your computer. That is, instead of clicking on buttons, you'll be typing in text and receiving feedback from your computer through text as well. The command line, also known as a shell, can help you modify and automate

many of the tasks you do on a computer every day, and is an essential tool for software developers.

PowerShell is a program from Microsoft that provides a command-line shell interface. Administrative tasks are performed by running cmdlets, which are pronounced command-lets, specialized classes of the .NET software framework that can carry out operations. Open-sourced in August 2016, PowerShell is now available across platforms, for both Windows and UNIX systems (including Mac and Linux).

To find Windows PowerShell, you can right-click on the Start menu icon on the lower left-hand corner of your screen. When the menu pops up, you should click on "Search," then type "PowerShell" into the search bar. When you are presented with options, right-click on "Windows PowerShell," the Desktop app. For our purposes, we'll select "Run as Administrator." When you are prompted with a dialogue box that asks "Do you want to allow this app to make changes to your PC?" click on "Yes."

Once you do this, you'll see a text-based interface that has a string of words that looks like this:



Windows 10 PowerShell

We can switch out of the system folder by typing the following command:

```
cd ~
```

Then we'll be in a directory such as `PS C:\Users\Sammy`.

To continue with our installation process, we are going to set up some permissions through PowerShell. Configured to run in the most secure mode by default, there are a few levels of permissions that you can set up as an administrator:

- Restricted is the default execution policy, under this mode you will not be able to run scripts, and PowerShell will work only as an interactive shell.
- AllSigned will enable you to run all scripts and configuration files that are signed by a trusted publisher, meaning that you could potentially open your machine up to the risk of running malicious scripts that happen to be signed by a trusted publisher.
- RemoteSigned will let you run scripts and configuration files downloaded from the internet signed by trusted publishers, again opening your machine up to vulnerabilities if these trusted scripts are actually malicious.
- Unrestricted will run all scripts and configuration files downloaded from the internet as soon as you confirm that you understand that the file was downloaded from the internet. In this case no digital signature is required so you could be opening your machine up to the risk of running unsigned and potentially malicious scripts downloaded from the internet.

We are going to use the RemoteSigned execution policy to set the permission for the current user that allows the PowerShell to accept downloaded scripts that we trust without making the permissions as broad as they would be with an Unrestricted permission. In the PowerShell, let's type:

```
Set-ExecutionPolicy -Scope CurrentUser
```

PowerShell will then prompt us to provide an execution policy, and since we want to use RemoteSigned, we'll type:

```
RemoteSigned
```

Once we press `enter` we'll be asked if we do want to change the execution policy. Type the letter `y` for "yes," and allow the changes to take effect. We can confirm that this worked by asking for the current permissions across the machine by typing:

```
Get-ExecutionPolicy -List
```

You should receive output that looks something like this:

Output

```
        Scope ExecutionPolicy
        ----- ---------------
MachinePolicy       Undefined
   UserPolicy       Undefined
      Process       Undefined
  CurrentUser    RemoteSigned
 LocalMachine       Undefined
```

This confirms that the current user can run trusted scripts downloaded from the internet. We can now move on to downloading the files we will need to set up our Python programming environment.

Step 2 — Installing the Package Manager Chocolatey

A package manager is a collection of software tools that work to automate installation processes that include the initial installation, upgrading and configuring of software, and removing software as

needed. They keep software installations in a central location and can maintain all software packages on the system in formats that are commonly used.

Chocolatey is a command-line package manager built for Windows that works like `apt-get` does on Linux. Available in an open-source version, Chocolatey will help you quickly install applications and tools, and we will be using it to download what we need for our development environment.

Before we install the script, let's read it to confirm that we are happy with the changes it will make to our machine. To do this, we will use the .NET scripting framework to download and display the Chocolatey script within the terminal window. We'll create a WebClient object called `$script` (you can call it whatever you want as long as you use `$` as the first character), that shares Internet connection settings with Internet Explorer:

```
$script = New-Object Net.WebClient
```

Let's look at the options that we have available to us by piping the object to the `Get-Member` class to return all members (properties and methods) of this WebClient object:

```
$script | Get-Member
```

Snippet of Ouput

```
 . . .
DownloadFileAsync          Method     void
DownloadFileAsync(uri address, string fileName), void
```

```
DownloadFileAsync(ur...
DownloadFileTaskAsync     Method
System.Threading.Tasks.Task
DownloadFileTaskAsync(string address, string fileNa...
DownloadString            Method     string
DownloadString(string address), string
DownloadString(uri address) #method we will use
DownloadStringAsync       Method     void
DownloadStringAsync(uri address), void
DownloadStringAsync(uri address, Sy...
DownloadStringTaskAsync   Method
System.Threading.Tasks.Task[string]
DownloadStringTaskAsync(string address), Sy…
 . . .
```

Looking over the output, we can identify the `DownloadString` method that we can use to display the script and signature in the PowerShell window. Let's implement this method:

```
$script.DownloadString("https://chocolatey.org/install
.ps1")
```

After we inspect the script, we can install Chocolatey by typing the following into PowerShell:

```
iwr https://chocolatey.org/install.ps1 -
UseBasicParsing | iex
```

The cmdlet `iwr` or `Invoke-WebRequest` allows us to extract data from the web. This will pass the script to the `iex` or `Invoke-Expression` cmdlet, which will execute the contents of the script, running the installation script for the Chocolatey package manager.

Allow PowerShell to install Chocolatey. Once it is fully installed, we can begin installing additional tools with the `choco` command.

If we need to upgrade Chocolatey at any time in the future, we can run the following command:

```
choco upgrade chocolatey
```

With our package manager installed, we can go on to install the rest of what we need for our Python 3 programming environment.

Step 3 — Installing the Text Editor nano (Optional)

We are now going to install nano, a text editor that uses a command line interface, which we can use to write programs directly within PowerShell. This is not a compulsory step, as you can alternatively use a text editor with a graphical user interface such as Notepad, but nano will get us more accustomed to using PowerShell.

Let's use Chocolatey to install nano:

```
choco install -y nano
```

Here we used the `-y` flag so that we confirm automatically that we want to run the script without being prompted.

Once nano is installed, we will be able to use the `nano` command to create new text files and will eventually use it to write our first Python

program.

Step 4 — Installing Python 3

Just like we did with nano above, we will use Chocolatey to install Python 3:

```
choco install -y python3
```

PowerShell will now install Python 3, generating output within PowerShell during that process.

Once the process is completed, you should see the following output:

Output

```
Environment Vars (like PATH) have changed.
Close/reopen your shell to
 See the changes (or in powershell/cmd.exe just type
'refreshenv').
The install of python3 was successful.
 Software installed as 'EXE', install location is
likely default.

Chocolatey installed 1/1 packages. 0 packages failed.
 See the log for details
(C:\ProgramData\chocolatey\logs\chocolatey.log).
```

With the installation is finished, you'll want to confirm that Python is installed and ready to go. To see the changes, use the command

`refreshenv` or close and re-open PowerShell as an Administrator, then check the version of Python available to you on your local machine:

```
python -V
```

You should get output such as:

Output

```
Python 3.5.1
```

Alongside Python, pip will be installed, which will manage software packages for Python. Let's ensure that pip is up-to-date by upgrading it:

```
python -m pip install --upgrade pip
```

With Chocolatey, we can call Python 3 with the `python` command. We will use the `-m` flag to run the library module as a script, terminating the option list, and from there use `pip` to install its upgrade.

Once Python is installed and pip updated, we can set up a virtual environment for our development projects.

Step 5 — Setting Up a Virtual Environment

Now that we have Chocolatey, nano, and Python installed, we can go on to create our programming environment with the venv module.

Virtual environments enable you to have an isolated space on your computer for Python projects, ensuring that each of your projects can have its own set of dependencies that won't disrupt any of your other projects.

Setting up a programming environment provides us with greater control over our Python projects and over how different versions of packages are handled. This is especially important when working with third-party packages.

You can set up as many Python programming environments as you want. Each environment is basically a directory or folder in your computer that has a few scripts in it to make it act as an environment.

Choose which directory you would like to put your Python programming environments in, or create a new directory with `mkdir`, as in:

```
mkdir Environments
cd Environments
```

Once you are in the directory where you would like the environments to live, you can create an environment by running the following command:

```
python -m venv my_env
```

Using the `python` command, we will run the `venv` library module to create the virtual environment that in this case we have called **`my_env`**.

Essentially, `venv` sets up a new directory that contains a few items which we can view with the `ls` command:

```
ls my_env
```

Output

```
Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----         8/22/2016   2:20 PM                
Include
d-----         8/22/2016   2:20 PM                Lib
d-----         8/22/2016   2:20 PM                
Scripts
-a----         8/22/2016   2:20 PM            107 
pyvenv.cfg
```

Together, these files work to make sure that your projects are isolated from the broader context of your local machine, so that system files and project files don't mix. This is good practice for version control and to ensure that each of your projects has access to the particular packages that it needs.

To use this environment, you need to activate it, which you can do by typing the following command that calls the activate script in the `Scripts` directory:

```
my_env\Scripts\activate
```

Your prompt will now be prefixed with the name of your environment, in this case it is called **my_env**:

```
(my_env) PS C:\Users\Sammy\Environments>
```

This prefix lets us know that the environment **my_env** is currently active, meaning that when we create programs here they will use only

this particular environment's settings and packages.

Step 6 — Creating a Simple Program

Now that we have our virtual environment set up, let's create a simple "Hello, World!" program. This will make sure that our environment is working and gives us the opportunity to become more familiar with Python if we aren't already.

To do this, we'll open up nano and create a new file:

```
(my_env) PS C:\Users\Sammy> nano hello.py
```

Once the text file opens up in Terminal we'll type out our program:

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

Exit nano by typing the `control` and `x` keys, and when prompted to save the file press `y` then the `enter` key.

Once you exit out of nano and return to your shell, let's run the program:

```
(my_env) PS C:\Users\Sammy> python hello.py
```

The hello.py program that you just created should cause Terminal to produce the following output:

Output

```
Hello, World!
```

To leave the environment, simply type the command `deactivate` and you will return to your original directory.

Conclusion

Congratulations! At this point you should have a Python 3 programming environment set up on your local Windows 10 machine and can begin a coding project!

To set up Python 3 on another computer, follow the local programming environment guides for Ubuntu 16.04, Debian 8, CentOS 7, or macOS. You can also read about installing Python and setting up a programming environment on an Ubuntu 16.04 server, which is especially useful when working on development teams.

With your local machine ready for software development, you can continue to learn more about coding in Python by following "Understanding Data Types in Python 3" and "How To Use Variables in Python 3".

How To Install Python 3 and Set Up a Local Programming Environment on CentOS 7

Python is a versatile programming language that can be used for many different programming projects. First published in 1991 with a name inspired by the British comedy group Monty Python, the development team wanted to make Python a language that was fun to use. Easy to set up, and written in a relatively straightforward style with immediate feedback on errors, Python is a great choice for beginners and experienced developers alike. Python 3 is the most current version of the language and is considered to be the future of Python.

This tutorial will guide you through installing Python 3 on your local CentOS 7 machine and setting up a programming environment via the command line.

Prerequisites

You will need a CentOS 7 computer with a non-root superuser account that is connected to the internet.

Step 1 — Preparing the System

We will be completing this installation through the command line. If your CentOS 7 computer starts up with a Graphical User Interface (GUI) desktop, you can gain access to the command line interface through the Menu, by navigating to Applications, then Utilities, and then clicking on Terminal. If you need more guidance on the terminal, be sure to read through the article "An Introduction to the Linux Terminal."

Before we begin with the installation, let's make sure to update the default system applications to have the latest versions available.

We will be using the open-source package manager tool yum, which stands for Yellowdog Updater Modified. This is a commonly used tool for working with software packages on Red Hat based Linux systems like CentOS. It will let you easily install and update, as well as remove software packages on your computer.

Let's first make sure that yum is up to date by running this command:

```
sudo yum -y update
```

The `-y` flag is used to alert the system that we are aware that we are making changes, preventing the terminal from prompting us to confirm.

Next, we will install yum-utils, a collection of utilities and plugins that extend and supplement yum:

```
sudo yum -y install yum-utils
```

Finally, we'll install the CentOS Development Tools, which are used to allow you to build and compile software from source code:

```
sudo yum -y groupinstall development
```

Once everything is installed, our setup is in place and we can go on to install Python 3.

Step 2 — Installing and Setting Up Python 3

CentOS is derived from RHEL (Red Hat Enterprise Linux), which has stability as its primary focus. Because of this, tested and stable versions of applications are what is most commonly found on the system and in downloadable packages, so on CentOS you will only find Python 2.

Since instead we would like to install the most current upstream stable release of Python 3, we will need to install IUS, which stands for Inline with Upstream Stable. A community project, IUS provides Red Hat Package Manager (RPM) packages for some newer versions of select software.

To install IUS, let's install it through `yum`:

```
sudo yum -y install
https://centos7.iuscommunity.org/ius-release.rpm
```

Once IUS is finished installing, we can install the most recent version of Python:

```
sudo yum -y install python36u
```

When the installation process of Python is complete, we can check to make sure that the installation was successful by checking for its version number with the `python3.6` command:

```
python3.6 -V
```

With a version of Python 3.6 successfully installed, we will receive the following output:

Output

```
Python 3.6.1
```

We will next install pip, which will manage software packages for Python:

```
sudo yum -y install python36u-pip
```

A tool for use with Python, we will use pip to install and manage programming packages we may want to use in our development projects. You can install Python packages by typing:

```
sudo pip3.6 install package_name
```

Here, `package_name` can refer to any Python package or library, such as Django for web development or NumPy for scientific computing. So if you would like to install NumPy, you can do so with the command `pip3.6 install numpy`.

Finally, we will need to install the IUS package python36u-devel, which provides us with libraries and header files we will need for Python 3 development:

```
sudo yum -y install python36u-devel
```

The venv module will be used to set up a virtual environment for our development projects in the next step.

Step 3 — Setting Up a Virtual Environment

Now that we have Python installed and our system set up, we can go on to create our programming environment with venv.

Virtual environments enable you to have an isolated space on your computer for Python projects, ensuring that each of your projects can have its own set of dependencies that won't disrupt any of your other projects.

Setting up a programming environment provides us with greater control over our Python projects and over how different versions of packages are handled. This is especially important when working with third-party packages.

You can set up as many Python programming environments as you want. Each environment is basically a directory or folder in your computer that has a few scripts in it to make it act as an environment.

Choose which directory you would like to put your Python programming environments in, or create a new directory with `mkdir`, as in:

```
mkdir environments
cd environments
```

Once you are in the directory where you would like the environments to live, you can create an environment by running the following command:

```
python3.6 -m venv my_env
```

Essentially, this command creates a new directory (in this case called **my_env**) that contains a few items that we can see with the `ls` command:

```
bin include lib lib64 pyvenv.cfg
```

Together, these files work to make sure that your projects are isolated from the broader context of your local machine, so that system files and project files don't mix. This is good practice for version control and to ensure that each of your projects has access to the particular packages that it needs.

To use this environment, you need to activate it, which you can do by typing the following command that calls the activate script in the `bin` directory:

```
source my_env/bin/activate
```

Your prompt will now be prefixed with the name of your environment, in this case it is called **my_env**:

```
(my_env) [sammy@localhost] environments]$
```

This prefix lets us know that the environment **my_env** is currently active, meaning that when we create programs here they will use only this particular environment's settings and packages.

Note: Within the virtual environment, you can use the command `python` instead of `python3.6`, and `pip` instead of `pip3.6` if you would prefer. If you use Python 3 on your machine outside of an environment, you will need to use the `python3.6` and `pip3.6` commands exclusively.

After following these steps, your virtual environment is ready to use.

Step 4 — Creating a Simple Program

Now that we have our virtual environment set up, let's create a simple "Hello, World!" program. This will make sure that our environment is working and gives us the opportunity to become more familiar with Python if we aren't already.

To do this, we'll open up a command-line text editor such as vim and create a new file:

```
(my_env) [sammy@localhost] environments]$ vi hello.py
```

Once the text file opens up in our terminal window, we will have to type `i` to enter insert mode, and then we can write our first program:

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

Now press `ESC` to leave insert mode. Next, type `:x` then `ENTER` to save and exit the file.

We are now ready to run our program:

```
(my_env) [sammy@localhost] environments]$ python
hello.py
```

The hello.py program that you just created should cause the terminal to produce the following output:

Output

```
Hello, World!
```

To leave the environment, simply type the command `deactivate` and you'll return to your original directory.

Conclusion

Congratulations! At this point you have a Python 3 programming environment set up on your local CentOS 7 machine and can begin a coding project!

To set up Python 3 on another computer, follow the local programming environment guides for Ubuntu 16.04, Debian 8, macOS, or Windows 10. You can also read about installing Python and setting up a programming environment on an Ubuntu 16.04 server, which is especially useful when working on development teams.

With your local machine ready for software development, you can continue to learn more about coding in Python by following "Understanding Data Types in Python 3" and "How To Use Variables in Python 3".

How To Install Python 3 and Set Up a Programming Environment on an Ubuntu 16.04 Server

This tutorial will get your Ubuntu 16.04 or Debian 8 server set up with a Python 3 programming environment. Programming on a server has many advantages and makes it easier for teams to collaborate on a development project. The general principles of this tutorial will apply to any distribution of Debian Linux.

Python is a versatile programming language that can be used for many different programming projects. First published in 1991 with a name inspired by the British comedy group Monty Python, the development team wanted to make Python a language that was fun to use. Easy to set up, and written in a relatively straightforward style with immediate feedback on errors, Python is a great choice for beginners and experienced developers alike. Python 3 is the most current version of the language and is considered to be the future of Python.

This tutorial will guide you through installing Python 3 on a Debian Linux server and setting up a programming environment.

Prerequisites

Before you begin, you'll need a server with Ubuntu 16.04, Debian 8, or another version of Debian Linux installed. You'll also need a sudo non-root user, which you can set up by following one of the tutorials below:

- Initial Server Setup with Ubuntu 16.04
- Initial Server Setup with Debian 8

If you're not already familiar with a terminal environment, you may find the article "An Introduction to the Linux Terminal" useful for becoming better oriented with the terminal.

Step 1 — Setting Up Python 3

Ubuntu 16.04, Debian 8, and other versions of Debian Linux ship with both Python 3 and Python 2 pre-installed. To make sure that our versions are up-to-date, let's update and upgrade the system with `apt-get`:

```
sudo apt-get update
sudo apt-get -y upgrade
```

The `-y` flag will confirm that we are agreeing for all items to be installed, but depending on your version of Linux, you may need to confirm additional prompts as your system updates and upgrades.

Once the process is complete, we can check the version of Python 3 that is installed in the system by typing:

```
python3 -V
```

You'll receive output in the terminal window that will let you know the version number. The version number may vary depending on whether you are on Ubuntu 16.04, Debian 8, or another version of Linux, but it will look similar to this:

Output

```
Python 3.5.2
```

To manage software packages for Python, let's install pip:

```
sudo apt-get install -y python3-pip
```

A tool for use with Python, pip installs and manages programming packages we may want to use in our development projects. You can install Python packages by typing:

```
pip3 install package_name
```

Here, **`package_name`** can refer to any Python package or library, such as Django for web development or NumPy for scientific computing. So if you would like to install NumPy, you can do so with the command `pip3 install numpy`.

There are a few more packages and development tools to install to ensure that we have a robust set-up for our programming environment:

```
sudo apt-get install build-essential libssl-dev
libffi-dev python3-dev
```

Once Python is set up, and pip and other tools are installed, we can set up a virtual environment for our development projects.

Step 2 — Setting Up a Virtual Environment

Virtual environments enable you to have an isolated space on your server for Python projects, ensuring that each of your projects can have its own set of dependencies that won't disrupt any of your other projects.

Setting up a programming environment provides us with greater control over our Python projects and over how different versions of packages are handled. This is especially important when working with third-party packages.

You can set up as many Python programming environments as you want. Each environment is basically a directory or folder on your server that has a few scripts in it to make it act as an environment.

We need to first install the venv module, part of the standard Python 3 library, so that we can invoke the pyvenv command which will create virtual environments for us. Let's install venv by typing:

```
sudo apt-get install -y python3-venv
```

With this installed, we are ready to create environments. Let's choose which directory we would like to put our Python programming environments in, or we can create a new directory with `mkdir`, as in:

```
mkdir environments
cd environments
```

Once you are in the directory where you would like the environments to live, you can create an environment by running the following command:

```
pyvenv my_env
```

Essentially, `pyvenv` sets up a new directory that contains a few items which we can view with the `ls` command:

```
ls my_env
```

Output

```
bin include lib lib64 pyvenv.cfg share
```

Together, these files work to make sure that your projects are isolated from the broader context of your local machine, so that system files and project files don't mix. This is good practice for version control and to ensure that each of your projects has access to the particular packages that it needs. Python Wheels, a built-package format for Python that can speed up your software production by reducing the number of times you need to compile, will be in the Ubuntu 16.04 `share` directory but in Debian 8 it will be in each of the `lib` directories as there is no `share` directory.

To use this environment, you need to activate it, which you can do by typing the following command that calls the activate script:

```
source my_env/bin/activate
```

Your prompt will now be prefixed with the name of your environment, in this case it is called **my_env**. Depending on what version of Debian Linux you are running, your prefix may look somewhat different, but the name of your environment in parentheses should be the first thing you see on your line:

```
(my_env) sammy@ubuntu:~/environments$
```

This prefix lets us know that the environment **my_env** is currently active, meaning that when we create programs here they will use only this particular environment's settings and packages.

Note: Within the virtual environment, you can use the command `python` instead of `python3`, and `pip` instead of `pip3` if you would prefer. If you use Python 3 on your machine outside of an environment, you will need to use the `python3` and `pip3` commands exclusively.

After following these steps, your virtual environment is ready to use.

Step 3 — Creating a Simple Program

Now that we have our virtual environment set up, let's create a simple "Hello, World!" program. This will make sure that our environment is working and gives us the opportunity to become more familiar with Python if we aren't already.

To do this, we'll open up a command-line text editor such as nano and create a new file:

```
(my_env) sammy@ubuntu:~/environments$ nano hello.py
```

Once the text file opens up in the terminal window we'll type out our program:

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

Exit nano by typing the `control` and `x` keys, and when prompted to save the file press `y`.

Once you exit out of nano and return to your shell, let's run the program:

```
(my_env) sammy@ubuntu:~/environments$ python hello.py
```

The hello.py program that you just created should cause your terminal to produce the following output:

Output

```
Hello, World!
```

To leave the environment, simply type the command `deactivate` and you will return to your original directory.

Conclusion

Congratulations! At this point you have a Python 3 programming environment set up on your Debian Linux server and you can now begin a coding project!

To set up Python 3 on another computer, follow the local programming environment guides for Ubuntu 16.04, Debian 8, Windows 10, or macOS.

With your server set up for software development, you can continue to learn more about coding in Python by following "Understanding Data Types in Python 3" and "How To Use Variables in Python 3".

How To Write Your First Python 3 Program

The “Hello, World!” program is a classic and time-honored tradition in computer programming. Serving as a simple and complete first program for beginners, as well as a good program to test systems and programming environments, “Hello, World!” illustrates the basic syntax of programming languages.

This tutorial will walk you through writing a “Hello, World” program in Python 3.

Prerequisites

You should have Python 3 installed as well as a local programming environment set up on your computer.

If you don’t have one set up, you can use one of the installation and setup guides below that is appropriate for your operating system:

- Ubuntu 16.04 or Debian 8
- CentOS 7
- Mac OS X
- Windows 10

Writing the “Hello, World!” Program

To write the “Hello, World!” program, let’s open up a command-line text editor such as nano and create a new file:

```
nano hello.py
```

Once the text file opens up in the terminal window we'll type out our program:

hello.py

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

Let's break down the different components of the code.

`print()` is a function that tells the computer to perform an action. We know it is a function because it uses parentheses. `print()` tells Python to display or output whatever we put in the parentheses. By default, this will output to the current terminal window.

Some functions, like the `print()` function, are built-in functions included in Python by default. These built-in functions are always available for us to use in programs that we create. We can also define our own functions that we construct ourselves through other elements.

Inside the parentheses of the `print()` function is a sequence of characters — `Hello, World!` — that is enclosed in quotation marks. Any characters that are inside of quotation marks are called a string.

Once we are done writing our program, we can exit nano by typing the `control` and `x` keys, and when prompted to save the file press `y`.

Once you exit out of nano you'll return to your shell.

Running the "Hello, World!" Program

With our "Hello, World!" program written, we are ready to run the program. We'll use the `python3` command along with the name of our program file. Let's run the program:

```
python3 hello.py
```

The hello.py program that you just created will cause your terminal to produce the following output:

Output

```
Hello, World!
```

Let's go over what the program did in more detail.

Python executed the line `print("Hello, World!")` by calling the `print()` function. The string value of `Hello, World!` was passed to the function.

In this example, the string `Hello, World!` is also called an argument since it is a value that is passed to a function.

The quotes that are on either side of `Hello, World!` were not printed to the screen because they are used to tell Python that they contain a string. The quotation marks delineate where the string begins and ends.

Since the program ran, you can now confirm that Python 3 is properly installed and that the program is syntactically correct.

Conclusion

Congratulations! You have written the "Hello, World!" program in Python 3.

From here, you can continue to work with the `print()` function by writing your own strings to display, and can also create new program files.

Keep learning about programming in Python by reading our full tutorial series How To Code in Python 3.

How To Work with the Python Interactive Console

The Python interactive console (also called the Python interpreter or Python shell) provides programmers with a quick way to execute commands and try out or test code without creating a file.

Providing access to all of Python's built-in functions and any installed modules, command history, and auto-completion, the interactive console offers the opportunity to explore Python and the ability to paste code into programming files when you are ready.

This tutorial will go over how to work with the Python interactive console and leverage it as a programming tool.

Entering the Interactive Console

The Python interactive console can be accessed from any local computer or server with Python installed.

The command you generally will want to use to enter into the Python interactive console for your default version of Python is:

```
python
```

If you have set up a programming environment, you can launch the environment and access the version of Python and modules you have installed in that environment by first entering into that environment:

```
cd environments
```

```
. my_env/bin/activate
```

Then typing the `python` command:

```
(my_env) sammy@ubuntu:~/environments$ python
```

In this case, the default version of Python is Python 3.5.2, which is displayed in the output once we enter the command, along with the relevant copyright notice and some commands you can type for extra information:

Output

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

The primary prompt for the next command is three greater-than signs (>>>):

```
>>>
```

You can target specific versions of Python by appending the version number to your command, with no spaces:

```
python2.7
```

Output

```
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

Here, we received the output that Python 2.7.12 will be used. If this is our default version of Python 2, we could also have entered into this interactive console with the command `python2`.

Alternatively, we can call the default Python 3 version with the following command:

```
python3
```

Output

```
Python 3.5.2 (default, Nov 17 2016, 17:05:23)
[GCC 5.4.0 20160609] on linux
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

We could have also called the above interactive console with the command `python3.5`.

With the Python interactive console running, we can move onto working with the shell environment for Python.

Working with the Python Interactive Console

The Python interactive interpreter accepts Python syntax, which you place following the >>> prefix.

We can, for example, assign values to variables:

```
>>> birth_year = 1868
```

Once we have assigned the integer value of `1868` to the variable `birth_year`, we will press return and receive a new line with the three greater-than signs as a prefix:

```
>>> birth_year = 1868
```

We can continue to assign variables and then perform math with operators to get calculations returned:

```
>>> birth_year = 1868
>>> death_year = 1921
>>> age_at_death = death_year - birth_year
>>> print(age_at_death)
53
>>>
```

As we would with a script in a file, we assigned variables, subtracted one variable from the other, and asked the console to print the variable that represents the difference.

Just like in any form of Python, you can also use the interactive console as a calculator:

```
>>> 203 / 20
10.15
>>>
```

Here, we divided the integer `203` by `20` and were returned the quotient of `10.15`.

Multiple Lines

When we are writing Python code the will cover multiple lines, the interpreter will use the secondary prompt for continuation lines, three dots (`...`).

To break out of these continuation lines, you will need to press `ENTER` twice.

We can see what this looks like in the following code that assigns two variables and then uses a conditional statement to determine what to print out to the console:

```
>>> sammy = 'Sammy'
>>> shark = 'Shark'
>>> if len(sammy) > len(shark):
...     print('Sammy codes in Java.')
... else:
...     print('Sammy codes in Python.')
...
Sammy codes in Python.
>>>
```

In this case the lengths of the two strings are equal, so the `else` statement prints. Note that you will need to keep Python indenting convention of four whitespaces, otherwise you will receive an error:

```
>>> if len(sammy) > len(shark):
... print('Sammy codes in Java.')
  File "<stdin>", line 2
    print('Sammy codes in Java.')
        ^
IndentationError: expected an indented block
>>>
```

You can not only experiment with code across multiple lines in the Python console, you can also import modules.

Importing Modules

The Python interpreter provides a quick way for you to check to see if modules are available in a specific programming environment. You can do this by using the `import` statement:

```
>>> import matplotlib
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ImportError: No module named 'matplotlib'
```

In the case above, the module matplotlib was not available within the current programming environment.

In order to install it, we'll need to leave the interactive interpreter and install with pip as usual:

```
(my_env) sammy@ubuntu:~/environments$ pip install matplotlib
```

Output

```
Collecting matplotlib
  Downloading matplotlib-2.0.2-cp35-cp35m-manylinux1_x86_64.whl (14.6MB)
...
Installing collected packages: pyparsing, cycler, python-dateutil, numpy, pytz, matplotlib
Successfully installed cycler-0.10.0 matplotlib-2.0.2 numpy-1.13.0 pyparsing-2.2.0 python-dateutil-2.6.0 pytz-2017.2
```

Once the matplotlib module along with its dependencies are successfully installed, you can go back into the interactive interpreter:

```
(my_env) sammy@ubuntu:~/environments$ python
```

```
>>> import matplotlib
```

At this point you will receive no error message and can use the installed module either within the shell or within a file.

Leaving the Python Interactive Console

There are two main ways to leave the Python interactive console, either with a keyboard shortcut or a Python function.

The keyboard shortcut `CTRL + D` in *nix-based systems or `CTRL + Z` then the `CTRL` key in Windows systems will interrupt your console and return you to your original terminal environment:

```
...
>>> age_at_death = death_year - birth_year
>>> print(age_at_death)
53
>>>
sammy@ubuntu:~/environments$
```

Alternatively, the Python function `quit()` will quit out of the interactive console and also bring you back to the original terminal environment that you were previously in:

```
>>> octopus = 'Ollie'
>>> quit()
sammy@PythonUbuntu:~/environments$
```

When you use the function `quit()`, it will show up in your history file, but the keyboard shortcut `CTRL + D` will not be recorded:

File: /home/sammy/.python_history

```
...
age_at_death = death_year - birth_year
print(age_at_death)
```

```
octopus = 'Ollie'
quit()
```

Quitting the Python interpreter can be done either way, depending on what makes sense for your workflow and your history needs.

Accessing History

One of the useful things about the Python interactive console is that all of your commands are logged to the `.python_history` file in *nix-based systems, which you can look at in a text editor like nano, for instance:

```
nano ~/.python_history
```

Once opened with a text editor, your Python history file will look something like this, with your own Python command history:

File: /home/sammy/.python_history

```
import pygame
quit()
if 10 > 5:
    print("hello, world")
else:
    print("nope")
sammy = 'Sammy'
shark = 'Shark'
...
```

Once you are done with your file, you can press `CTRL + X` to leave nano.

By keeping track of all of your Python history, you can go back to previous commands and experiments, and copy and paste or modify that code for use in Python programming files or in a Jupyter Notebook.

Conclusion

The Python interactive console provides a space to experiment with Python code. You can use it as a tool for testing, working out logic, and more.

For use with debugging Python programming files, you can use the Python `code` module to open up an interactive interpreter within a file, which you can read about in our guide How To Debug Python with an Interactive Console.

How To Write Comments

Comments are lines that exist in computer programs that are ignored by compilers and interpreters. Including comments in programs makes code more readable for humans as it provides some information or explanation about what each part of a program is doing.

Depending on the purpose of your program, comments can serve as notes to yourself or reminders, or they can be written with the intention of other programmers being able to understand what your code is doing.

In general, it is a good idea to write comments while you are writing or updating a program as it is easy to forget your thought process later on, and comments written later may be less useful in the long term.

Comment Syntax

Comments in Python begin with a hash mark (#) and whitespace character and continue to the end of the line.

Generally, comments will look something like this:

```
# This is a comment
```

Because comments do not execute, when you run a program you will not see any indication of the comment there. Comments are in the source code for humans to read, not for computers to execute.

In a "Hello, World!" program, a comment may look like this:

hello.py

```
# Print “Hello, World!” to console
print("Hello, World!")
```

In a `for` loop that iterates over a list, comments may look like this:

sharks.py

```
# Define sharks variable as a list of strings
sharks = ['hammerhead', 'great white', 'dogfish',
'frilled', 'bullhead', 'requiem']

# For loop that iterates over sharks list and prints
each string item
for shark in sharks:
    print(shark)
```

Comments should be made at the same indent as the code it is commenting. That is, a function definition with no indent would have a comment with no indent, and each indent level following would have comments that are aligned with the code it is commenting.

For example, here is how the `again()` function from the How To Make a Simple Calculator Program in Python 3 tutorial is commented, with comments following each indent level of the code:

calculator.py

```
...
# Define again() function to ask user if they want to
use the calculator again
def again():
```

```
    # Take input from user
    calc_again = input('''
Do you want to calculate again?
Please type Y for YES or N for NO.
''')

    # If user types Y, run the calculate() function
    if calc_again == 'Y':
        calculate()

    # If user types N, say good-bye to the user and
end the program
    elif calc_again == 'N':
        print('See you later.')

    # If user types another key, run the function
again
    else:
        again()
```

Comments are made to help programmers, whether it is the original programmer or someone else using or collaborating on the project. If comments cannot be properly maintained and updated along with the code base, it is better to not include a comment rather than write a comment that contradicts or will contradict the code.

When commenting code, you should be looking to answer the why behind the code as opposed to the what or how. Unless the code is

particularly tricky, looking at the code can generally tell what the code is doing or how it is doing it.

Block Comments

Block comments can be used to explain more complicated code or code that you don't expect the reader to be familiar with. These longer-form comments apply to some or all of the code that follows, and are also indented at the same level as the code.

In block comments, each line begins with the hash mark and a single space. If you need to use more than one paragraph, they should be separated by a line that contains a single hash mark.

Here is an example of a block comment that defines what is happening in the `main()` function defined below:

```
# The main function will parse arguments via the
parser variable.  These
# arguments will be defined by the user on the
console.  This will pass
# the word argument the user wants to parse along with
the filename the
# user wants to use, and also provide help text if the
user does not
# correctly pass the arguments.

def main():
  parser = argparse.ArgumentParser()
  parser.add_argument(
      "word",
```

```
      help="the word to be searched for in the text
file."
  )
  parser.add_argument(
      "filename",
      help="the path to the text file to be searched
through"
  )
...
```

Block comments are typically used when operations are less straightforward and are therefore demanding of a thorough explanation. You should try to avoid over-commenting the code and should tend to trust other programmers to understand Python unless you are writing for a particular audience.

Inline Comments

Inline comments occur on the same line of a statement, following the code itself. Like other comments, they begin with a hash mark and a single whitespace character.

Generally, inline comments look like this:

```
[code]  # Inline comment about the code
```

Inline comments should be used sparingly, but can be effective for explaining tricky or non-obvious parts of code. They can also be useful if you think you may not remember a line of the code you are writing in the

future, or if you are collaborating with someone who you know may not be familiar with all aspects of the code.

For example, if you don't use a lot of math in your Python programs, you or your collaborators may not know that the following creates a complex number, so you may want to include an inline comment about that:

```
z = 2.5 + 3j  # Create a complex number
```

Inline comments can also be used to explain the reason behind doing something, or some extra information, as in:

```
x = 8  # Initialize x with an arbitrary number
```

Comments that are made in line should be used only when necessary and when they can provide helpful guidance for the person reading the program.

Commenting Out Code for Testing

In addition to using comments as a way to document code, the hash mark can also be used to comment out code that you don't want to execute while you are testing or debugging a program you are currently creating. That is, when you experience errors after implementing new lines of code, you may want to comment a few of them out to see if you can troubleshoot the precise issue.

Using the hash mark can also allow you to try alternatives while you're determining how to set up your code. For example, you may be deciding between using a `while` loop or a `for` loop in a Python game, and can

comment out one or the other while testing and determining which one may be best:

guess.py

```
import random

number = random.randint(1, 25)

# number_of_guesses = 0

for i in range(5):
# while number_of_guesses < 5:
    print('Guess a number between 1 and 25:')
    guess = input()
    guess = int(guess)

    # number_of_guesses = number_of_guesses + 1

    if guess < number:
        print('Your guess is too low')

    if guess > number:
        print('Your guess is too high')

    if guess == number:
        break

if guess == number:
```

```
    print('You guessed the number!')

else:
    print('You did not guess the number. The number
was ' + str(number))

<figure class="code">
```

Commenting out code with the hash mark can allow you to try out different programming methods as well as help you find the source of an error through systematically commenting out and running parts of a program.

Conclusion

Using comments within your Python programs helps to make your programs more readable for humans, including your future self. Including appropriate comments that are relevant and useful can make it easier for others to collaborate with you on programming projects and make the value of your code more obvious.

From here, you may want to read about Python's Docstrings in PEP 257 to provide you with more resources to properly document your Python projects.

Understanding Data Types

In Python, like in all programming languages, data types are used to classify one particular type of data. This is important because the specific data type you use will determine what values you can assign to it and what you can do to it (including what operations you can perform on it).

In this tutorial, we will go over the important data types native to Python. This is not an exhaustive investigation of data types, but will help you become familiar with what options you have available to you in Python.

Background

One way to think about data types is to consider the different types of data that we use in the real world. An example of data in the real world are numbers: we may use whole numbers (0, 1, 2, …), integers (…, -1, 0, 1, …), and irrational numbers (π), for example.

Usually, in math, we can combine numbers from different types, and get some kind of an answer. We may want to add 5 to π, for example:

```
5 + π
```

We can either keep the equation as the answer to account for the irrational number, or round π to a number with a brief number of decimal places, and then add the numbers together:

```
5 + π = 5 + 3.14 = 8.14
```

But, if we start to try to evaluate numbers with another data type, such as words, things start to make less sense. How would we solve for the following equation?

```
sky + 8
```

For computers, each data type can be thought of as being quite different, like words and numbers, so we will have to be careful about how we use them to assign values and how we manipulate them through operations.

Numbers

Any number you enter in Python will be interpreted as a number; you are not required to declare what kind of data type you are entering. Python will consider any number written without decimals as an integer (as in `138`) and any number written with decimals as a float (as in `138.0`).

Integers

Like in math, integers in computer programming are whole numbers that can be positive, negative, or 0 (…, `-1`, `0`, `1`, …). An integer can also be known as an `int`. As with other programming languages, you should not use commas in numbers of four digits or more, so when you write 1,000 in your program, write it as `1000`.

We can print out an integer in a simple way like this:

```
print(-25)
```

Output

```
-25
```

Or, we can declare a variable, which in this case is essentially a symbol of the number we are using or manipulating, like so:

```
my_int = -25
print(my_int)
```

Output

```
-25
```

We can do math with integers in Python, too:

```
int_ans = 116 - 68
print(int_ans)
```

Output

```
48
```

Integers can be used in many ways within Python programs, and as you continue to learn more about the language you will have a lot of opportunities to work with integers and understand more about this data type.

Floating-Point Numbers

A floating-point number or a float is a real number, meaning that it can be either a rational or an irrational number. Because of this, floating-point

numbers can be numbers that can contain a fractional part, such as `9.0` or `-116.42`. Simply speaking, for the purposes of thinking of a `float` in a Python program, it is a number that contains a decimal point.

Like we did with the integer, we can print out a floating-point number in a simple way like this:

```
print(17.3)
```

Output

```
17.3
```

We can also declare a variable that stands in for a float, like so:

```
my_flt = 17.3
print(my_flt)
```

Output

```
17.3
```

And, just like with integers, we can do math with floats in Python, too:

```
flt_ans = 564.0 + 365.24
print(flt_ans)
```

Output

```
929.24
```

With integers and floating-point numbers, it is important to keep in mind that $3 \neq 3.0$, as `3` refers to an integer while `3.0` refers to a float.

Booleans

The Boolean data type can be one of two values, either True or False. Booleans are used to represent the truth values that are associated with the logic branch of mathematics, which informs algorithms in computer science.

Whenever you see the data type Boolean, it will start with a capitalized B because it is named for the mathematician George Boole. The values `True` and `False` will also always be with a capital T and F respectively, as they are special values in Python.

Many operations in math give us answers that evaluate to either True or False:

- greater than
    - $500 > 100$ `True`
    - $1 > 5$ `False`
- less than
    - $200 < 400$ `True`
    - $4 < 2$ `False`
- equal
    - $5 = 5$ `True`
    - $500 = 400$ `False`

Like with numbers, we can store a Boolean value in a variable:

```
my_bool = 5 > 8
```

We can then print the Boolean value with a call to the `print()` function:

```
print(my_bool)
```

Since 5 is not greater than 8, we will receive the following output:

Ouput

```
False
```

As you write more programs in Python, you will become more familiar with how Booleans work and how different functions and operations evaluating to either True or False can change the course of the program.

Strings

A string is a sequence of one or more characters (letters, numbers, symbols) that can be either a constant or a variable. Strings exist within either single quotes `'` or double quotes `"` in Python, so to create a string, enclose a sequence of characters in quotes:

```
'This is a string in single quotes.'
```

```
"This is a string in double quotes."
```

You can choose to use either single quotes or double quotes, but whichever you decide on you should be consistent within a program.

The simple program "Hello, World!" demonstrates how a string can be used in computer programming, as the characters that make up the phrase `Hello, World!` are a string.

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

As with other data types, we can store strings in variables:

```
hw = "Hello, World!"
```

And print out the string by calling the variable:

```
print(hw)
```

Ouput

```
Hello, World!
```

Like numbers, there are many operations that we can perform on strings within our programs in order to manipulate them to achieve the results we are seeking. Strings are important for communicating information to the user, and for the user to communicate information back to the program.

Lists

A list is a mutable, or changeable, ordered sequence of elements. Each element or value that is inside of a list is called an item. Just as strings are

defined as characters between quotes, lists are defined by having values between square brackets `[ ]`.

A list of integers looks like this:

```
[-3, -2, -1, 0, 1, 2, 3]
```

A list of floats looks like this:

```
[3.14, 9.23, 111.11, 312.12, 1.05]
```

A list of strings:

```
['shark', 'cuttlefish', 'squid', 'mantis shrimp']
```

If we define our string list as `sea_creatures`:

```
sea_creatures = ['shark', 'cuttlefish', 'squid',
'mantis shrimp']
```

We can print them out by calling the variable:

```
print(sea_creatures)
```

And we see that the output looks exactly like the list that we created:

Output

```
['shark', 'cuttlefish', 'squid', 'mantis shrimp']
```

Lists are a very flexible data type because they are mutable in that they can have values added, removed, and changed. There is a data type that is similar to lists but that can't be changed, and that is called a tuple.

Tuples

A tuple is used for grouping data. It is an immutable, or unchangeable, ordered sequence of elements.

Tuples are very similar to lists, but they use parentheses `( )` instead of square brackets and because they are immutable their values cannot be modified.

A tuple looks like this:

```
('blue coral', 'staghorn coral', 'pillar coral')
```

We can store a tuple in a variable and print it out:

```
coral = ('blue coral', 'staghorn coral', 'pillar
coral')
print(coral)
```

Ouput

```
('blue coral', 'staghorn coral', 'pillar coral')
```

Like in the other data types, Python prints out the tuple just as we had typed it, with parentheses containing a sequence of values.

Dictionaries

The dictionary is Python's built-in mapping type. This means that dictionaries map keys to values and these key-value pairs are a useful way to store data in Python. A dictionary is constructed with curly braces on either side `{ }`.

Typically used to hold data that are related, such as the information contained in an ID, a dictionary looks like this:

```
{'name': 'Sammy', 'animal': 'shark', 'color': 'blue',
'location': 'ocean'}
```

You will notice that in addition to the curly braces, there are also colons throughout the dictionary. The words to the left of the colons are the keys. Keys can be made up of any immutable data type. The keys in the dictionary above are: `'name', 'animal', 'color', 'location'`.

The words to the right of the colons are the values. Values can be comprised of any data type. The values in the dictionary above are: `'Sammy', 'shark', 'blue', 'ocean'`.

Like the other data types, let's store the dictionary inside a variable, and print it out:

```
sammy = {'name': 'Sammy', 'animal': 'shark', 'color':
'blue',  'location': 'ocean'}
print(sammy)
```

Ouput

```
{'color': 'blue', 'animal': 'shark', 'name': 'Sammy',
'location': 'ocean'}
```

If we want to isolate Sammy's color, we can do so by calling `sammy['color']`. Let's print that out:

```
print(sammy['color'])
```

Output

```
blue
```

As dictionaries offer key-value pairs for storing data, they can be important elements in your Python program.

Conclusion

At this point, you should have a better understanding of some of the major data types that are available for you to use in Python. Each of these data types will become important as you develop programming projects in the Python language.

You can learn about each of the data types above in more detail by reading the following specific tutorials: - Numbers - Booleans - Strings - Lists - Tuples - Dictionaries

Once you have a solid grasp of data types available to you in Python, you can learn how to convert data types.