**Python Overview**

Python is a general purpose, dynamically typed and interpreted, object oriented programming language that was created in the late 1980s by Guido van Rossum. Python’s design philosophy revolves around readability. It’s meant to be easy to read and easy to write. This is accomplished by using white-space to delineate code blocks instead of the more traditional curly brackets and semi-colons.

**How Python Runs**

Generally all python code is run using an interpreter. The most popular and original interpreter is called CPython, because it’s implemented in the c programming language. Several other interpreters exist however, many of which are implemented in languages other than C like Java and C#. The most common Python interpreter CPython, uses an automatic garbage collector to manage memory. And Python is widely known for having a non-traditional, minimalist syntax which is largely based on white space, and designed to be clean and readable.

**Python Versions**

When first getting into python it can be a bit confusing because unlike many other programming languages Python has two major, non-compatible versions that are currently widely used.

Python version 2.7.3, released in 2012, is the last iteration of version 2 that was released. This version is for the most part, backwards compatible with all previous versions.

In 2008, the Founder, Guido van Rossum decided to clean up the Python codebase and overhaul a lot of the things in Python 2 that he didn’t like, thus creating Python 3.

Python 3 was adopted slowly at first, mainly because it is not backwards compatible with Python 2, and there was a huge eco-system of package libraries written for Python 2 which now would not work in python 3.

But now-a-day’s the Python 3 ecosystem has for the most part caught up, making Python 3 the obvious choice for new developers looking to learn the language. Python 3 is also the version that will be taught in this tutorial.

Choosing an IDE

Many developers choose to write Python using a specialized integrated development environment. Three of the most popular are Eclipse, PyCharm and NetBeans.

**Installation**

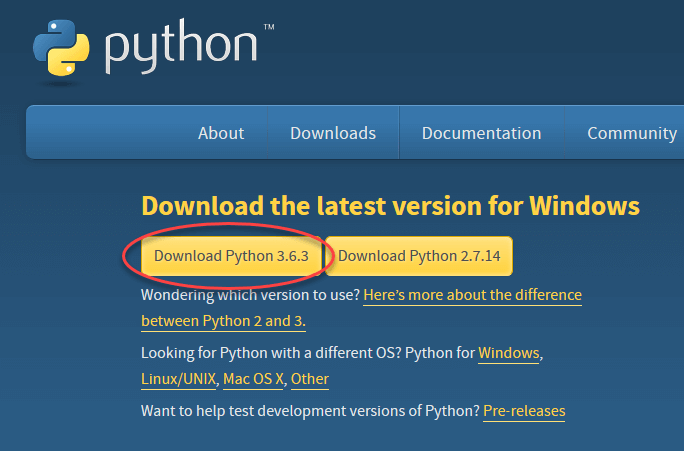
PyCharm is a cross-platform editor developed by JetBrains. Pycharm provides all the tools you need for productive Python development.

Below are the detailed steps for installing Python and PyCharm

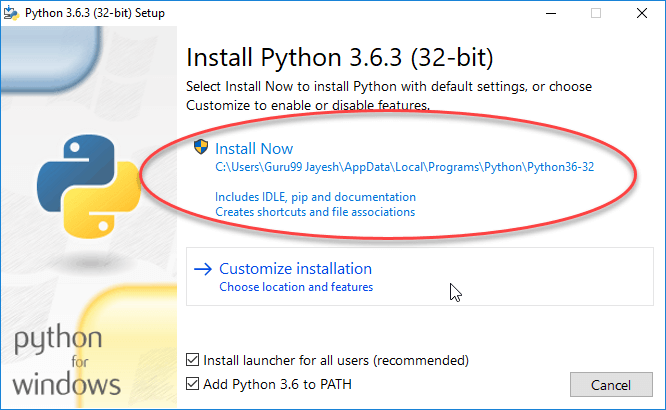
**How to Install Python IDE**

Below is a step by step process on how to download and install Python on Windows:

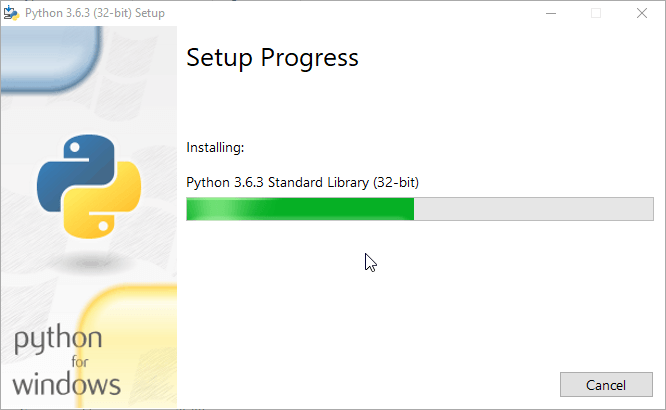
**Step 1)** To download and install Python, visit the official website of Python <https://www.python.org/downloads/> and choose your version. We have chosen Python version 3.6.3



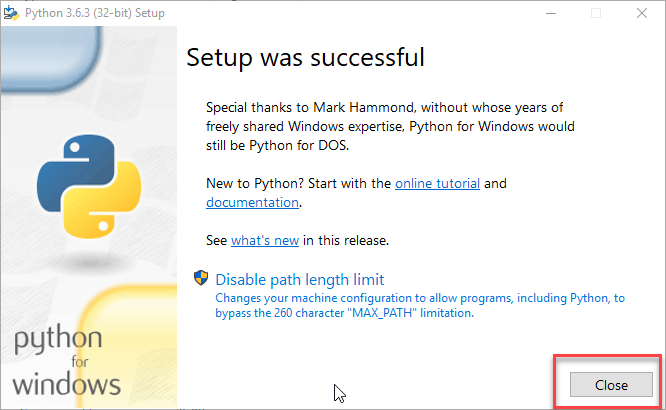
**Step 2)** Once the download is completed, run the .exe file to install Python. Now click on Install Now.



**Step 3)** You can see Python installing at this point.



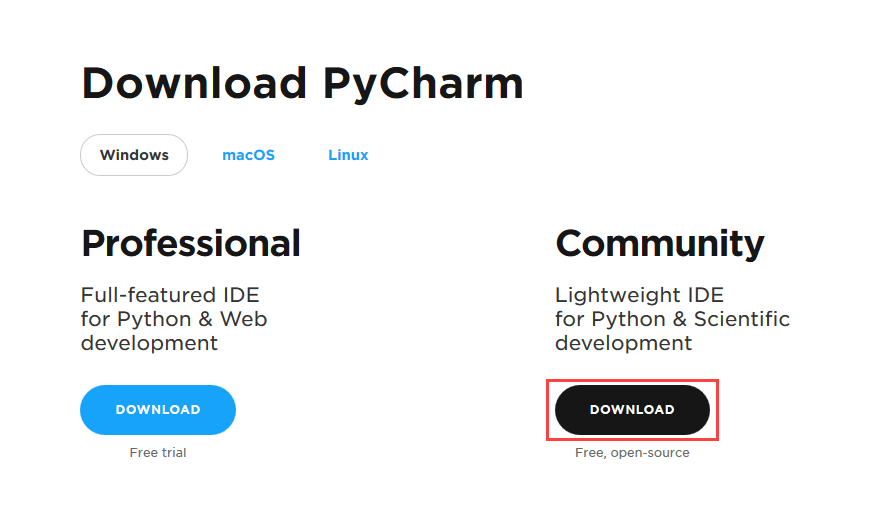
**Step 4)** When it finishes, you can see a screen that says the Setup was successful. Now click on “Close”.



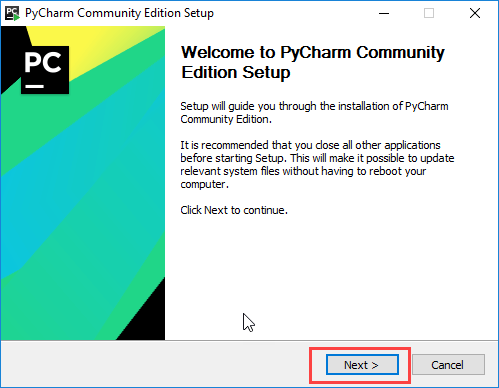
**How to Install Pycharm**

Here is a step by step process on how to download and install Pycharm IDE on Windows:

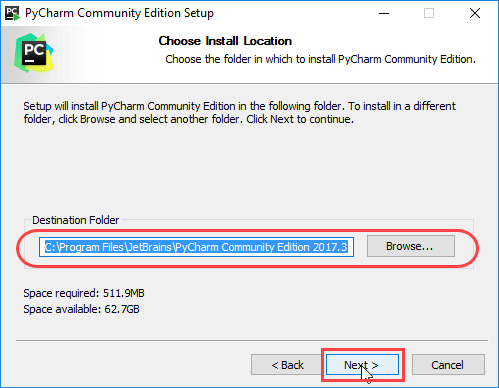
**Step 1)** To download PyCharm visit the website <https://www.jetbrains.com/pycharm/download/> and Click the “DOWNLOAD” link under the Community Section.



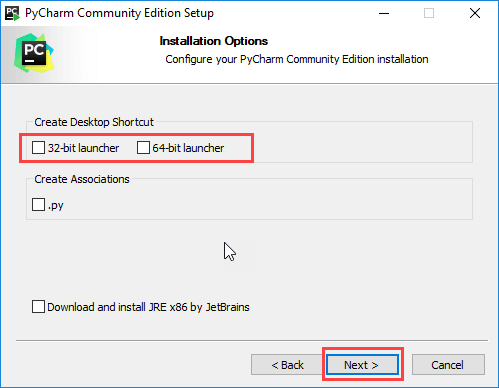
**Step 2)** Once the download is complete, run the exe for install PyCharm. The setup wizard should have started. Click “Next”.



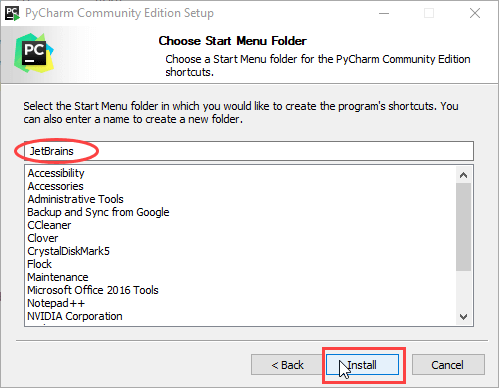
**Step 3)** On the next screen, Change the installation path if required. Click “Next”.



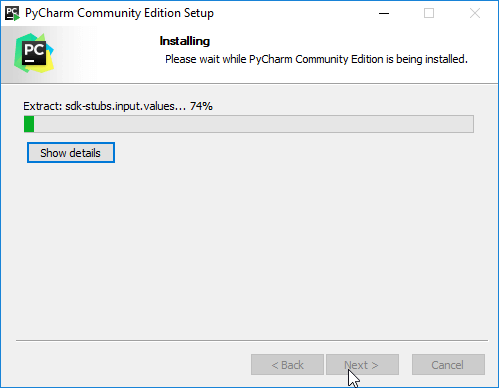
**Step 4)** On the next screen, you can create a desktop shortcut if you want and click on “Next”.



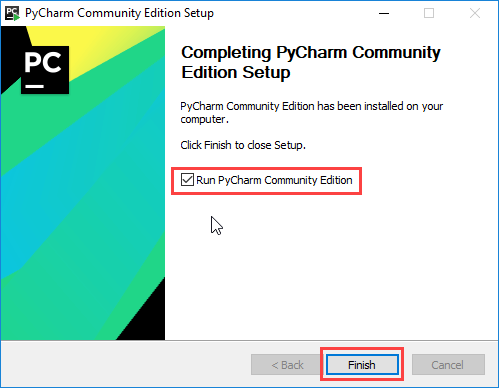
**Step 5)** Choose the start menu folder. Keep selected JetBrains and click on “Install”.



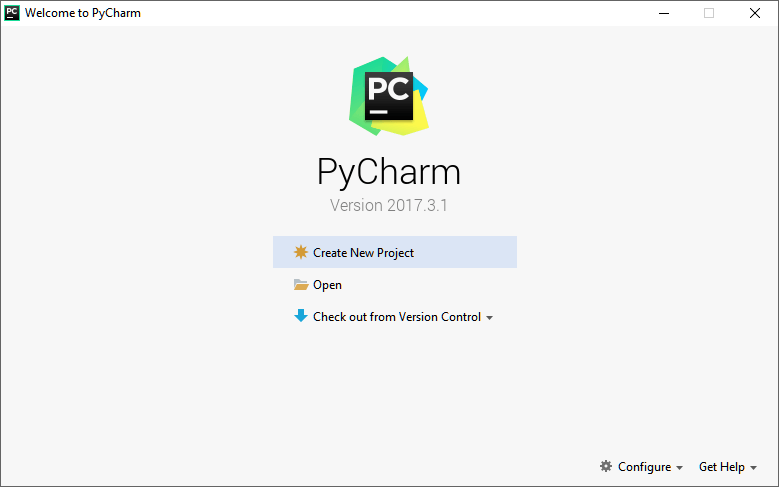
**Step 6)** Wait for the installation to finish.



**Step 7)** Once installation finished, you should receive a message screen that PyCharm is installed. If you want to go ahead and run it, click the “Run PyCharm Community Edition” box first and click “Finish”.

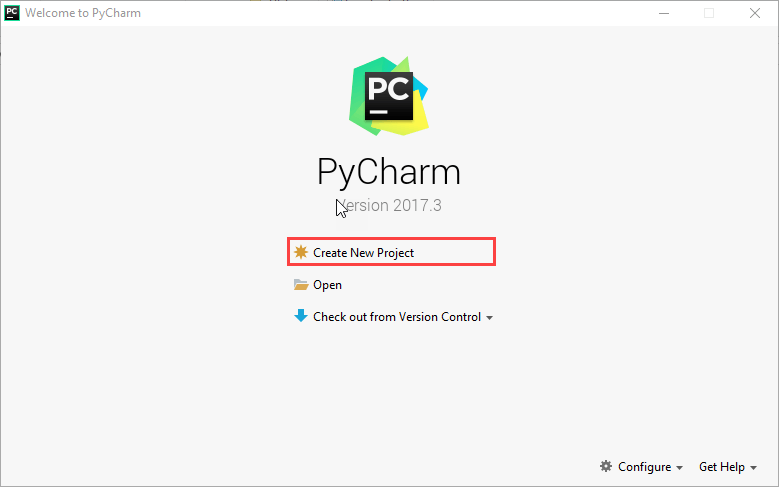


**Step 8)**After you click on “Finish,” the Following screen will appear.



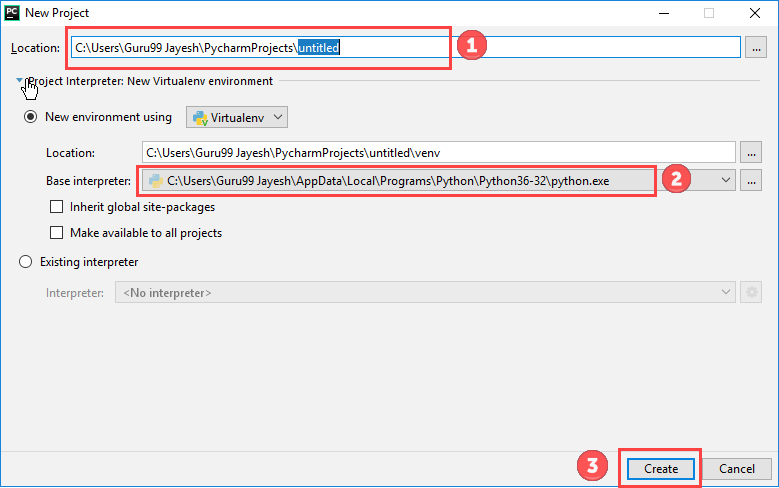
**Creating First Program**

**Step 1)** Open PyCharm Editor. You can see the introductory screen for PyCharm. To create a new project, click on “Create New Project”.

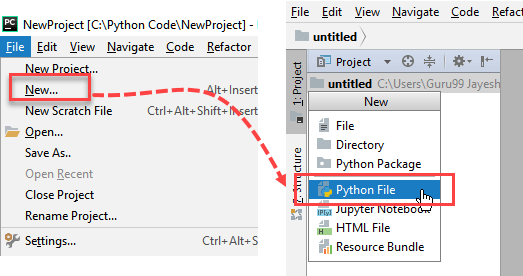


**Step 2)**You will need to select a location.

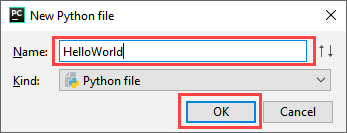
1. You can select the location where you want the project to be created. If you don’t want to change location than keep it as it is but at least change the name from “untitled” to something more meaningful, like “FirstProject”.
2. PyCharm should have found the Python interpreter you installed earlier.
3. Next Click the “Create” Button.



**Step 3)**Now Go up to the “File” menu and select “New”. Next, select “Python File”.



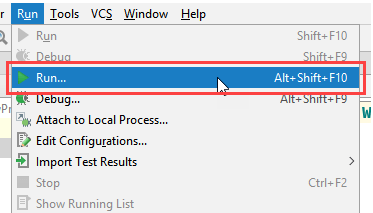
**Step 4)**A new pop up will appear. Now type the name of the file you want (Here we give “HelloWorld”) and hit “OK”.



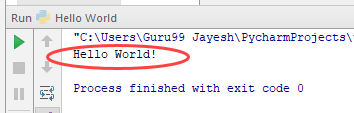
**Step 5)**Now type a simple program – print (‘Hello World!’).



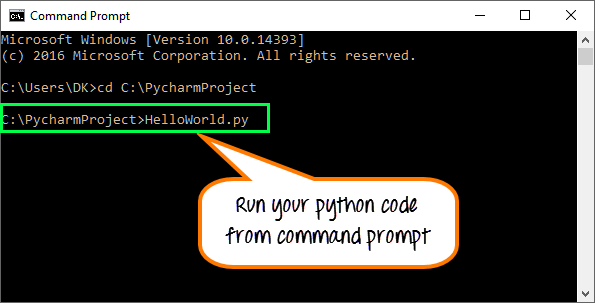
**Step 6)**Now Go up to the “Run” menu and select “Run” to run your program.



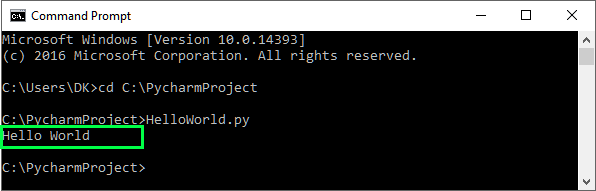
**Step 7)**You can see the output of your program at the bottom of the screen.



**Step 8)**Don’t worry if you don’t have Pycharm Editor installed, you can still run the code from the command prompt. Enter the correct path of a file in command prompt to run the program.



The output of the code would be



**Step 9)**If you are still not able to run the program, we have Python Editor for you.

Please run the given code at [Python Online Editor](https://www.guru99.com/execute-python-online.html)

print("Hello World")

**Code**

print("Hello")

print("World")

print("!")

**Drawing A Shape**

As we begin our programming journey in Python, we'll start by paying homage to a tradition widely shared among new programmers, which is learning how to print text out onto the screen. As we learn to talk to computers using a programming language like Python, it will help to know how to get them to talk back to us in the form of output, so let's take a look!

We can achieve this by using a print statement. All programming languages, Python included, have some form of a print statement which is essentially an instruction that tells the computer to output some text.

**A Simple Print Statement**

For our first program we'll simply instruct the computer to output the phrase Hello World! in Python.

print("Hello World!")

As you can see, this program isn't very complicated, in fact it's only one line, or instruction. All computer programs in Python are essentially just made up of a bunch of these little instructions. It's the job of the programmer to take simple instructions like the one above, and chain them together into longer and longer programs, which in turn, the computer will follow.

Just like you might write down the instructions for baking your favorite cookies (ie. a recipe), or you might give someone a set of directions to get to your house, you can also give a computer a set of instructions (ie. a program), and it will follow them.

If there's one thing you should take away from this lesson it's that programs are just long sets of instructions, it's as simple as that.

All the Hello World program above actually does is output the text "Hello World" on to the screen (try running it). But as we learn more and more of these instructions in Python, we can get the computer to do some truly amazing things. (To put it in perspective, this website is made up entirely of tiny instructions like the one above!).

**More Complex Printing**

Now that we've written our Hello World program and gotten our feet wet, let's combine a bunch of these print instructions together to draw a shape.

print(" /|")

print(" / |")

print(" / |")

print("/\_\_\_|")

As you can see the new program adds a few more instructions and even just by looking at the raw code, you can see how a shape might result from it. Run the program and check the output!

In actuality what's happening here is the computer is starting with the first line of Python code above and then continuing down from there. This is very important, the computer will always start with the first instruction we give it.

Just like you wouldn't start a recipe half way through, the computer won't start the program somewhere in the middle.

The order here is important because if the computer doesn't execute these statements in order, then the shape won't look very good. The computer starts with the first line, executes it, thus printing /| onto the screen. Only once it's done with that first instruction will it move onto the second instruction, printing / | out onto a new line.

It will continue executing each instruction, one after the other until it gets to the end of the program, at which time it will stop!

**Know the code**

When you're starting out, it's often good to execute the code mentally in your head before you have the computer run it. Step through each instruction manually pretending you're the computer, and verify that it does what you want.

As we go forward in the course there will be many instances where we want the computer to give us information. Maybe it's the answer to a question we ask it, or the result of adding two numbers together. These print statements will come in handy in almost every program we write!

Especially as a beginner it's important to get comfortable writing code. As it's most likely new to you, the best thing you can do is just take these lessons as a starting out point and branch off on your own for additional practice. Above we drew a triangle, but now maybe try drawing a square, or printing out a poem, or doing anything else you can think of.

When it comes to programming, the only way to get better and more comfortable is to practice.

**Advanced Instructions**

In this simple program we looked at the instructions for printing something out onto the screen. This is a good starting point, and is where most developers will start their journey in Python. But as we go forward in the course, we'll slowly start to learn about more and more powerful and useful instructions, which in turn, will allow us to write even more powerful and useful programs.

There's a lot more to see and learn, so let's head on to the next lesson and keep going!

Variables

The topic of this lesson, is in fact one of the most fundamental and important topics in all of computer programming, and once you master it, you'll be well on your way to becoming a Python developer!

The topic I'm talking about is variables! Variables are a Python developer's bread and butter. They make it really easy to write complex programs with lots and lots of instructions, and keep everything organized and straight.

Often times when you're writing a program in Python, there will be several pieces of data that you need to keep track of. Just like you're constantly keeping track of different pieces of information as you go about your day, like what you have to do at work, or when you're going to the gym, or what the name of the person who just moved in across the hall is, computers also need to keep track of various pieces of information.

Think for example of Facebook. Facebook keeps track of tons of information and displays it on their websites and apps. Things like names, birthdays, status', etc. In order for us to write more complex programs like these, that do more than print text out onto the screen, we'll need to learn how to manage all of the information in our programs and use variables.

Variables are essentially containers which allow a program to store different pieces of information inside of them. Once the information is stored inside the variable, that information can then be accessed throughout the program simply by referring to the variable's name.

Variables are containers where we can store pieces of data we want to work with in our programs

**Writing a Story**

To demonstrate let's take a look at a situation where variables would come in handy:

print("There once was a man named George")

print("He was 70 years old")

print("He really liked the name George")

print("But didn't like being 70")

The program above is very similar to the one we wrote in the previous lesson where we drew a shape. It's just printing out a little story about a man named George.

But let's say we wanted to change the name of the character in the story. Well in order to do that we would have to look through all of the code we wrote, and find every place where we wrote the name George and individually change it to the new name (hoping we didn't make any spelling mistake along the way).

Suppose that we also wanted to do the same thing with George's age. Maybe we wanted to make him a bit younger or older. Again we'd have to look through each line of code individually and change the age.

In a small program like this, making changes like these is fairly manageable, after all our story is only 4 lines. But imagine if instead of 4 we had 400 or 4000 lines, where the character's name and age were mentioned dozens or hundreds of times.

Going through and changing all of that would be a nightmare, and even if you used something like a find and replace, it wouldn't be reliable as it couldn't account for any misspellings or mis-agings.

And in this case we're only keeping track of two pieces of information about the character, suppose we also wanted to keep track of (and potentially change) his birthday, favorite book, favorite color, and top 5 friends. The more information and data we want to keep track of, and the longer our programs get, the more difficult it is to mange all of it. That's where variables come it.

Adding Variables to the Story

Remember, variables are essentially containers where we can store pieces of information. And we can use them to store, for example, our character's name and age. Checkout the modified program below which stores the name and age into variables, and then interweaves them throughout the story.

character\_name = "George"

character\_age = "70"

print("There once was a man named " + character\_name)

print("He was " + character\_age + " years old")

print("He really liked the name " + character\_name)

print("But didn't like being " + character\_age)

# You can also print variables using a ','

print("My variable is", character\_name)

What we've done above is very powerful. We've replaced each instance of the character's name and age in the story, with the variable that we stored them in. The values for the name and age are actually stored inside of these variables which we've given descriptive names, and those values can then be accessed by simply referring to the variable's name.

This is powerful for multiple reasons. First is that if we wanted to change the characters name, now all we have to do is simply change it up at the top of the program were we stored it in the variable.

We've now eliminated the need for searching through the program and finding each instance of the name or age and manually changing it. All the print statements are doing is just printing out what's stored inside the variable.

So if this story eventually became 4000 lines, and mentioned the character's name 100 times, we could easily change it by making a small change in that one spot at the beginning of the program.

Variables also allow us to change the pieces of information we're keeping track of throughout our programs. So imagine if halfway though the story the character's age changed (maybe they suddenly aged 10 years?). We could simply update the value in the variable to the new age, and the age would automatically update everywhere thereafter!

character\_name = "George"

character\_age = "70"

print("There once was a man named " + character\_name)

print("He was " + character\_age + " years old")

character\_age = "80"

print("He really liked the name " + character\_name)

print("But didn't like being " + character\_age)

If you run the program now, the character's age will be different in the two places its printed out.

Rules for variables

Now before we get too ahead of ourselves, I want to share some rules and best practices, specifically for naming these variables. In Python the naming rules are as follows:

Names are case-sensitive and may begin with:

letters, $, \_

After, may include

letters, numbers, $, \_

Convention says

Start with a lowercase word, then additional words are separated

by underscores

ex. my\_first\_variable

As long as you follow those rules you should be good to go. I'll also mention that you want to be as descriptive as possible when naming these variables. This is not only for your sake, but for the sake of everyone who has to read your code.

It's better to have a variable name that's long and descriptive (even though it might take more time to type out), than one that's just a single letter or confusing!

Trust me, do Python programmers everywhere a favor and don't be the developer who's lazy and doesn't take the time to name their variables correctly!

Python Programmers Love Variables

As I said in the beginning of the lesson, variables are a programmers best friend! As we move on from here in the course we'll be using variables to store and keep track of all the different pieces of information we encounter. Keep practicing and soon enough variables will become an invaluable and second nature part of your programming repertoire!

**Strings**

Let's talk about text, or as it's known to programmers, strings! In this lesson we'll take a look at one of Python's most powerful data types and how you can use them to write awesome programs.

As you continue on your journey to becoming a Python developer, one really useful skill you'll want to pickup along the way is the ability to work with strings in a variety of ways. Text is an important component of the programs we'll be writing in this course, so I want to introduce you to how strings are structured in Python, and show you some cool and interesting ways we can manipulate and utilize them.

**String basics**

First let's take a look at how we might store a string inside a variable:

greeting = "Hello"

Notice first the way the string is formatted. Whenever we create strings in Python we want to surround them with quotation marks. Any text inside the quotation marks is considered part of the string.

A string can contain any characters you want, including spaces, numbers, special characters, and even quotation marks (if you prefix them with a backslash \"). Strings can also be as long as you want, in fact many programs you'll encounter as a developer might have strings with thousands or even millions of characters in them!

**Indexes**

No converstaion about strings would be complete without talking about string indexes.

greeting = "Hello"

#indexes: 01234

All characters in a string have specific indexes associated to them, it's basically their location from the start of the string. Notice above I've labeled each character in the string with it's corresponding index (the line of code with the indexes is a comment, it doesn't get executed by the computer).

If you look closely though you might notice something weird. The first letter of the string has an index of 0, not as you might have expected 1. This isn't a mistake on my part, in fact all string indexes start at 0. So we would say that the first character of the string is at index position 0, the second character is at index position 1, etc.

This is something that tends to trip up new developers (myself included when I learned it). This way of indexing things, where you start at 0 instead of 1, is common across most programming languages, and is something you'll get used to over time.

Almost all counting in programming will start with 0 instead of 1, it's just something you have to get used to.

If you know the index of a character in a string, or want to know the character at a give index you can access one single character like so:

print( greeting[0] )

In the code above, we're accessing a character in the string at the specified index. You can use this syntax to access any of the characters in the string as long as a valid index is used.

**Length**

So now that we have a handle on string indexes, I'll show you how you can get the length of a string. Knowing the length of a particular string can come in handy in many circumstances:

print( len(greeting) )

If you execute the program above, you'll see the length of the string printed out.

Now there's one thing you need to understand which is the relationship between string length and string indexes. In Python, a string's length will always be one more than the index of it's last character. This is because we start the indexes at 0. So the string Hello has a length of 5, but the index of the o at the end is at position 4.

Understanding this relationship between string length and string indexes is really important, and will be necessary later on in the course when we dive into more advanced topics!

**More fun with strings**

Now that we've talked about how strings are structured and learned about the relationship between string length & indexes, we can build off these concepts to continue learning. Check out the video above for an explanation of some more cool things you can do with strings in Python.

Video Code

greeting = "Hello"

#indexes: 01234

print( len(greeting) )

print( greeting[0] )

print( greeting[-1] )

print( greeting.find("llo") )

print( greeting.find("z") )

print( greeting[2:] )

print( greeting[2:3] )

**Math**

Computers and numbers are like bread and butter. In fact the term computer actually used to refer to people who's job it was to do calculations and "compute" things all day. Needless to say we've come a long way since then, and modern computers are capable of performing billions of calculations in the time it takes a normal human to even write a number down.

Not to nerd out here too much, but when you get down to it, all computer instructions, all the code that we've written in this course, and all the code that's currently running on your computer, is actually just a collection of complex patterns of 0s & 1s running through your processor. And that same processor that's running everything on your computer is made up of a bunch of ALUs which stands for Arithmetic Logic Unit, basically a digital circuit which does basic math really really fast!

It's because computers are good at math, that you're even able to read this website right now. So as we forge forward and learn to become Python developers, it will help to spend some time learning how to leverage the amazing mathematical abilities of this awesome machine we're programming.

**Integers vs Floats**

The first thing I'd like to discuss is a very fundamental concept which is integral to mathematics as well as computer programming, which is the difference between an integer and a float.

These are two different types of numbers that we can work with in our programs, and it's important that you have a grasp of the differences so you can use them appropriately.

Let's start with integers which are whole numbers, or counting numbers. That means both negative numbers, positive numbers and zero, as long as they don't have decimal places after them.

Integers are often used to store thing that can't be broken up, like the number of minutes in the hour (you wouldn't say it 10:42.5). They can also be used for counting the number of times something has happened. This in particular will come in handy when we look at loops later on in this course.

Unlike integers, floats are any numbers with decimal places. Even if a float is technically a whole number like 42, in order for it to be a float it would have to have a .0 at the end of it 42.0.

Floats are often used when precision is a factor, like in scientific measurements, or in many cases when representing percentages (0.0 - 1.0).

Any time you're working with numbers in your programs, you always want to be aware of the type of number you're using, be it an integer or a float. Failure to do say may result in aspects of your program not performing as intended, or math calculations not giving precise enough answers.

**Math Operations**

Now that we have a grasp on the different types of numbers, let use them! You can tell a computer to perform any of the 4 basic math operations, addition, subtraction, multiplication and division by simply writing out the problem like you would normally.

Note that in Python we use the \* to indicate multiplication and the / to indicate division.

print( 2 \* 3 ) # Basic Arithmetic: +, -, /, \*

print( 2\*\*3 ) # Basic Arithmetic: +, -, /, \*

print( 10 % 3 ) # Modulus Op. : returns remainder of 10/3

print( 1 + 2 \* 3 ) # order of operations

print(10 / 3.0) # int's and doubles

Above are some common examples of math operation you might use.

Python will also respect order of operations, so you can group certain expressions using parenthesis to encode more complex math problems.

**Modulo Operator**

If you look in the code above you'll also notice an operator that you might not have seen before, indicated by a percent sign (%). This is what's referred to as the modulo operator and is a pretty common operation that is performed in Python programs.

The modulo operator will take the number on the left, divide it by the number on the right and then return the remainder. So if we ask the computer to perform 10 % 3 we would get back a 1, because 10/3 is 3 with a remainder of 1.

In the code above we are not making use of variables, but often times it's extra useful to store numbers in variables because numbers, unlike strings don't give us any context. If I see a random string in my program I can at least try to infer what it's purpose is by reading the text, the same can't always be said about numbers.

Advanced Math

Aside from all of the basic math operations and the modulo operation, Python also provides ways of doing more complex and difficult calculations with relative easy. Check out the video above for a full demonstration of advanced math in Python!

Video Code

print( 2 \* 3 ) # Basic Arithmetic: +, -, /, \*

print( 2\*\*3 ) # Basic Arithmetic: +, -, /, \*

print( 10 % 3 ) # Modulus Op. : returns remainder of 10/3

print( 1 + 2 \* 3 ) # order of operations

print(10 / 3.0) # int's and doubles

num = 10

num += 100 # +=, -=, /=, \*=

print(num)

++num

print(num)

# Math module has useful math methods

import math

print( pow(2, 3) )

print( math.sqrt(144) )

print( round(2.7) )

**Getting User Input**

Throughout this course we've taken a look at how we can make the computer give us information. We do this by using print statements. Print statements give us an idea of whats going on in the program at a given time, and are vital to writing good software.

But what about the flip side, what happens when we want to give the computer some information? Almost all applications that you interact with on a daily basis will provide ways for you to give them information. Just look up at the browser search bar above this website.

In this lesson we'll take a look at how to write programs in Python which accept user input.

The Code

Programs which get input from the user need to be written in a specific way. The program will essentially prompt the user for information, and then when the user hits enter, will take whatever they entered and be able to use it going forward in it's execution.

The code for accepting user input is actually not that much more complex than the code for printing. In Python, we can get user input like this:

name = input("Enter your name: ")

print("Hello", name + "!")

The code above simply prompts the user for information, and the prints out what they entered in.

One of the most important things to note here is that we're storing whatever the user entered into a variable. It's vital that we do this, after all, what's the point of asking the user for something if we're not gonna keep track of it.

**Execution flow**

As you play around with this program you'll notice that it actually stops it's flow of execution when awaiting input from the user. In pervious programs we're written they seem to execute and be done in a matter of milliseconds. But in this case if you wait 10 minutes before you decide to input something, the computer will wait along with you.

This is because when the computer see's that it's supposed to accept input it stops everything it's doing and waits. In a lot of cases, the input that the computer receives from the user is vital to it being able to perform it's task, whatever that may be.

Many programs you'll encounter (and eventually write) will rely on the user to tell them what to do and when it's okay for them to stop. Once we discover more advanced topics we'll eventually be able to write programs that have menus and give the user an option of what they want to do! But this is where it all starts.

**Wrapping Up**

Now that you have the basics down, play around for a bit and try to change the prompt, or maybe try asking the user for multiple pieces of information and use some of the string functions we talked about in the previous lessons to modify it.

There's still a lot to learn about getting user input. As of right now we're only receiving input in the form of strings. As we move forward in the course we'll take a look at how to allow the user to enter numbers, and other types of data.

**Creating A Calculator**

Code

num1 = int(input("Enter First Num: "))

num2 = int(input("Enter Second Num: "))

print(num1 + num2)

**Building A Mad Libs Game**

Code

color = input("Enter a color: ")

pluralNoun = input("Enter a plural noun: ")

celebrity = input("Enter a celebrity: ")

print("Roses are", color)

print(pluralNoun + " are blue")

print("I love", celebrity)

**Lists**

Up until this point in the course, we've been working with different types of data, and generally we've been using variables to store that data. Variables are great because we can take pieces of data that would otherwise be difficult to remember and keep track of and store them in a nice, named container.

But variables as we've seen them so far do have one short coming, and that's when it comes to storing large amounts of related data.

Many of the programs that you'll find the real world, and that we'll be writing in this course will deal with large amounts of data. It could be a list of names, an index of phone numbers, the answers to a test, or any number of collections of data like these.

When you have lots of data like this that's meant to be grouped together, you can't use normal variables to store it. If you did you would need to create hundreds or thousands of variables, and that's assuming that you even know how many pieces of data would be in the collection.

Luckily for us the problem of storing large amounts of data like this has been solved with something called an array.

An array is a container which acts a lot like the variables we've seen so far in this course, except they allow us to store more than one piece of data.

So if you wanted to keep track of a list of names or phone numbers in Python, instead of storing them in separate variables you could store them in one massive array!

**Creating an array**

Let's take a look at how to create an array in Python:

lucky\_numbers = [4, 8, "fifteen", 16, 23, 42.0]

# indexes 0 1 2 3 4 5

You'll notice that creating an array is very similar to creating a variable. In this case we're storing a list of numbers inside the array. Notice how they're all separated with commas. In this case we're populating the array with numbers right off the bat, but a lot of times in programs you'll start with an empty or half empty array and fill it up as the program executes.

**Array indexes**

You'll also notice that I've indicated the indexes of each element in the array. Just like strings, which assign indexes to each of their characters, arrays elements also have indexes.

We start with the first element at index position 0, the indexes then increment by one from there. Just like strings you have to get used to the first element being at index position 0. This is something that commonly trips up new programmers!

The array is an entity in itself, but if you want to access an element inside the array you can do so by referring to it directly by it's index. Let's take a look at accessing an element in an array:

lucky\_numbers[0] = 90

print(lucky\_numbers[0])

print(lucky\_numbers[1])

print(lucky\_numbers[-1])

print(lucky\_numbers[2:])

print(lucky\_numbers[2:4])

print(len(lucky\_numbers))

The elements inside the array are just normal strings, numbers or booleans. So once you've accessed one they can be treated like you would normally treat one of those data types. You can also modify the elements in an array by simply assigning them a new value as you can see above.

**Wrapping it up**

So there you have it, an introduction to arrays! Take a look at the video above for more information and to get a full step by step walk through of working with arrays!

**List Functions**

Code

friends = []

friends.append("Oscar")

friends.append("Angela")

friends.insert(1, "Kevin")

# friends.remove("Kevin")

print( friends )

print( friends.index("Oscar") )

print( friends.count("Angela") )

friends.sort()

print( friends )

friends.clear()

print( friends )

**Tuples**

Code

lucky\_numbers = (4, 8, "fifteen", 16, 23, 42.0)

# indexes 0 1 2 3 4 5

# lucky\_numbers[0] = 90 # tuples are immutable

print(lucky\_numbers[0])

print(lucky\_numbers[1])

print(lucky\_numbers[-1])

print(lucky\_numbers[2:])

print(lucky\_numbers[2:4])

print(len(lucky\_numbers))

**Functions**

In this lesson I wanna talk to you about a really important concept in Python which makes it really easy to keep our programs organized and re-usable. I'm talking of course about Functions!

Maybe you've heard the word Function floating around as you've begun your programming journey, but today we're gonna break down exactly what they are and I'll show you how to use them in Python.

Functions and Organization

So far in this course we've written a lot of code. All sorts of code to do all sorts of different things. And as we go forward and begin to write more and more complex and time consuming programs, one thing that is going to be vitally important is that we have good organization.

Without good organization a program can quickly get out of hand. Eventually you'll find that you're writing separate pieces of code in different parts of the file or project which do exactly the same thing!

There's a principle in software development which all good programmers try to follow called the DRY principle. It stands for "Don't repeat yourself". In other words, it means, don't write the same code twice!

As programmers, a Function is our best defence against violating the DRY principal. A Function is basically a collection of code which can be executed anywhere in our program and used over and over again.

In many programs there will be certain functionality that needs to be re-used multiple times.

Maybe you want to format a string a specific way, or you want to perform some complex math calculation on two numbers, or you want to parse through an array looking for a particular element. All of these are situations where a Function can be written which encapsulates the code needed to do those things, and can then be re-used throughout the program again and again.

sayHi() Function

To fully illustrate Functions lets take a look at an example of a simple Function which says hi to the user. Below is the Python code for our sayHi Function.

def sayHi():

print("Hello")

Right off the bat you'll notice a few things here. First is that just like variables we can actually give Functions descriptive names. In this case we've named the Function sayHi because that's what it's doing, that's the functionality that it's performing.

It's really important that you give these good names, otherwise your program will be significantly less read-able.

We follow the Function name with a set of open and closed parenthesis, and then below the top line (which we call the Function declaration), we can write some code. All of the code below the declaration is considered inside the Function.

Once we've written the code inside the Function (note: you can write as much code as you like), we can then execute that code wherever and whenever we want in our programs by calling the Function.

Calling a Function

When you call a Function you are essentially telling Python that you want it to stop what it's doing and go execute all of the code contained in the Function. Then once it's done, to come back and keep going on with the normal flow of the program. Let's look at how we can call a Function:

def sayHi():

print("Hello")

sayHi()

To call a Function we simple state the Function's name and then directly after, place an open and closed parenthesis. The open and closed parenthesis tells Python that you want to actually execute the Function.

Like I mentioned above, when a Function is called Python will stop what it's doing, and jump over to wherever that Function is and execute all of the code inside. Once it's finished with the code in the Function it will return to where it was and keep going.

So if we wanted to sayHi in multiple places in our program, instead of having to duplicate the code for doing that multiple times, we could instead just call the sayHi Function.

This is also advantageous because if we wanted to change the way our program said hi then we could just update the code in the Function body and the program would automatically update the way it's say hi all throughout our program!

Parameters and Arguments

Functions are powerful not only because they allow us to re-use the same code over and over, but because they also allow us to pass them information.

When a Function is called in a program, the code calling it can supply it with information specific to it's wants and needs. For example let's say that we wanted to modify our sayHi to say hi to a specific person. The way it's written right now it can't do that, all it does it just say hello.

In order to allow it to say hi to a specific person, we need to specify in the Function declaration that it accepts the name of a person from the caller. This is referred to as a parameter. Check out our updated sayHi Function below and see if you can spot the parameter.

def sayHi(name):

print("Hello " + name)

I'll give you a hint, look inside the parenthesis, notice how we've declared a variable there called name.

This variable is actually called a parameter, and it defines a piece of information that the Function accepts. Often times, Functions will accept parameters which help them to perform their task in a way that is more specific to the needs of the individual caller.

With this Function the way it is now, we can actually call it and have it say hi to whomever we want.

def sayHi(name):

print("Hello " + name)

sayHi("Mike")

Notice above that now when I call the sayHi Function, instead of leaving the parenthesis blank, I place a string inside "Mike". This string is what's called an argument. This argument will get sent to the Function when it is called and be stored in the name parameter specified in the sayHi Function.

This is an important point of terminology.

A parameter is a special variable that a Function will use to perform it's task in a way that is more specific to the needs of the caller. An argument is the value the caller gives to the Function which will get stored inside the parameter variable.

In the case above we only specified one single parameter, but you can specify as many as you want (although any more than 4 or 5 is generally considered too many). Let's take a look at this same Function but with an additional parameter age:

def sayHi(name):

print("Hello " + name + " you are " + str(age))

sayHi("Mike", 24)

This looks pretty similar to our previous example, but now in addition to the name parameter, I've also specified the age parameter. Multiple parameters are separated with a comma, and you can't have two parameters with the same name.

When we call the Function now we can pass in the age as well. Notice that the position of the arguments matches the position of the parameters; The name is first and the age is second in both cases, this is crucial!

**Wrapping it up**

So there you have it, an introduction to Functions. This is just the first step, but an important step towards mastering Functions and being able to really take advantage of them in our programs. Try writing some Functions of your own and see if you can come up with anything cool!

**Return Statements**

def add\_numbers(num1, num2=99):

return num1 + num2

sum = add\_numbers(4, 3)

print(sum)

**If Statements**

Up to this point in the course we've written some pretty cool programs! But in this lesson I want to introduce you to a concept that will allow our programs to be even more powerful.

If statements are a special structure in Python where we can actually allow our programs to make decisions.

By using an if statement, I could execute certain code when certain conditions are true, and I could execute other code when other conditions are true.

What are they?

Basically, if statements allow our programs to respond to the input that they are given. Depending on the state of the data in the program, Python will be able to respond to it. So when certain data is certain values we can do certain things, and when other data is other values we can do other things.

With if statements, our programs are able to become a lot smarter and we're able to do more things with them (which is always a good thing)!

Now, one of the things you might not realize is that you're already an expert at dealing with if statements. Maybe not so much in Python, but in everyday life.

As you go throughout your day, you're confronted with countless if statements, I've included a few common one's below:

I wake up

If I'm hungry

I eat breakfast

I leave my house

if it's cloudy

I bring an umbrella

otherwise

I bring sunglasses

Im at a restaurant

if I want meat

I order a steak

otherwise if I want pasta

I order spaghetti & meatballs

otherwise

I order a salad.

Above I've written out three if statements that you might encounter in your everyday life. While these aren't valid Python code, they are very similar to what we'll be looking at when we eventually do get to some code.

If

All if statements are made up of two parts, a boolean expression and an action. Remember that boolean values are just true or false. Boolean expressions therefore are expressions which can be evaluated to either a true or a false value.

Albert Einstein was a woman

Is a false Boolean expression

Abraham Lincoln was president of the United States

Is a true Boolean expression

In an If statement a boolean expression acts as a guard. Essentially its guarding a particular piece of code.

So IF the Boolean expression is true, THEN we want to execute the code contained in the statement (the action). Whereas IF the boolean expression is false, THEN we want to skip over the code contained in the expression. The action can be any segment of valid code, and as long as the boolean expression concludes to true, then it will be executed.

So taking for example the first if statement in the previous section:

I wake up

If I'm hungry

I eat breakfast

Here the boolean expression If i'm hungry acts as the boolean guard. Only if this is true will you eat breakfast (i.e. the action).

Why are They Useful?

If statements are useful because they allow a programmer to build a certain level of intelligence and decision making into a program. When writing programs, there’s a bunch of situations where it can be useful to execute certain code when certain conditions are met, and skip over certain code when conditions aren’t met.

As our programs get more complex, we'll be dealing with more and more data, and we'll want to use that data to inform the program on what it should do in certain situations.

If statements are super powerful and common in most programs. This website even, is build with hundreds of if statements (I should know I painstakingly wrote them all).

One of the important learning curves that all new developers have to overcome is to begin thinking in terms of if statements. But don't worry, if this isn't coming super naturally to you right now you'll get the hang of it after some practice.

Else

Now that we’ve learned about IF statements let’s talk about another piece to the puzzle which are called Else statements. An Else statement basically specifies what to do when the if statement returns false.

So IF a condition is false -> do something else. An else can only exist in combination with an if statement. Check out some real world Else statements below

IF the road is open

drive down it

ELSE

turn right

IF they have vanilla icecream

buy me some

ELSE

buy me chocolate icecream

IF it is 3 oclock

call me

ELSE

Text me

Else statements have the same basic concept as if statements, they just allow you to add more logic to the program.

Also notice that an else statement doesn’t have a boolean expression guarding it. That’s because the code inside an else statement is always executed when the boolean expression in the if statement returns false.

If Else

Finally, we can add even more logic into these by utilizing If Else statements. An if else statement is very similar to an if statement, much like an if statement it will check a boolean expression. If that boolean expression is true it will allow some code to be executed.

If else statement's boolean expressions are only evaluated however after all if statements or if else statements before them have evaluated to false.

In the beginning of this lesson we saw an example of an if else in action:

Im at a restaurant

if I want meat

I order a steak

otherwise if I want pasta

I order spaghetti & meatballs

otherwise

I order a salad.

Notice the if else statement sand-witched in between the initial if statement and the final else statement. Much like the if statement it has a boolean guard, if I want pasta, but this is only checked if the original if statement is false.

Python If Statements

So now that we have a grasp of the basic concepts, lets look at some code! Here's an example of an if statement in Python. Take a look at the video above for more about how all of this comes together!

Copyis\_student = False

is\_smart = False

if is\_student and is\_smart:

print("You are a student")

elif is\_student and not(is\_smart):

print("You are not a smart student")

else:

print("You are not a student and not smart")

# >, <, >=, <=, !=, ==

if 1 > 3:

print("number omparison was true")

if "dog" == "cat":

print("string omparison was true")

Code

Copyis\_student = False

is\_smart = False

if is\_student and is\_smart:

print("You are a student")

elif is\_student and not(is\_smart):

print("You are not a smart student")

else:

print("You are not a student and not smart")

# >, <, >=, <=, !=, ==

if 1 > 3:

print("number omparison was true")

if "dog" == "cat":

print("string omparison was true")

Better Calculator

num1 = int(input("num1: "))

op = input("Operator: ")

num1 = int(input("num1: "))

if op == "+":

print(num1 + num2)

elif op == "-":

print(num1 - num2)

elif op == "/":

print(num1 / num2)

elif op == "\*":

print(num1 \* num2)

else:

print("Invalid Operator")

Dictionaries

test\_grades = {

"Andy" : "B+",

"Stanley" : "C",

"Ryan" : "A",

3 : 95.2

}

print( test\_grades["Andy"] )

print( test\_grades.get("Ryan", "No Student Found") )

print( test\_grades[3] )

**While Loops**

There's an on-going gag on the popular show "The Simpsons", where Bart Simpson is forced to write lines on the chalkboard over and over agian to teach him a lesson. One of my favorites features Bart writing "I will not show off" on the blackboard hundreds of times.

I've always liked this gag, and I'm excited to say that I can use it to help explain a programming concept.

In the episode, when Bart wrote all those lines, he was literally writing the same thing 500 times for the whole afternoon. Now, imagine if instead of doing it himself, he asked a computer to do this same thing. After all computers can print out information to the screen a lot faster than humans can write it.

So lets say we wanted to ask a computer to write “I will not show off” 500 times, how might we do that?

Well you might say that we need a print statement, and you’re right. If we’re going to ask the computer to print to the screen, our best bet is a print statement. But remember we don’t just want it to print once, we want it to print 500 times. So maybe let’s write a program that looks like this psudo-code (this is not real Python code)

print "I will behave"

print "I will not show off"

print "I will not show off"

... x500

print "I will not show off"

By writing out 500 print statements, we can accomplish our task of printing out 500 times. But to do it we need to write 500 individual lines of code. Imagine if we wanted to print something one thousand times, or even one million times, it seems unreasonable that we would write one million lines of the same thing right?

Early programmers recognized this problem and created something called a loop.

A loop is a structure that’s built into all programming languages which tells the computer to execute a certain block of code a certain number of times, or until a certain condition is met. Loops are useful for all types of tasks and like If statements, are used in almost every program you’ll find.

In the majority of programming languages there are two types of loops, a While Loop and a For Loop.

In this lesson we’re going to take a look at while loops, learn how and when to use them, and how they can be used to create as efficient a program as possible.

How do they work?

While loops are the most basic type of loop available. A while loop, a lot like an If statement, is comprised of a Boolean expression and an action.

Basically a while loop says:

While something is true:

perform some action

Take the case of writing lines. A corresponding while loop might say something like this

While you haven’t written 500 lines:

write “I will not show off"

You can see here that our boolean expression “You haven’t written 500 lines”, first gets tested, then, if it’s true, we execute the code in the loop. But the important thing to note is that once the code in the loop gets executed, the computer then checks the boolean expression again, to see if anything has changed.

Following the execution of this loop we see that it will execute 500 times, and on the 501th iteration of the loop, it will terminate (by terminate I mean it will move onto the next line of code outside the loop).

While loops in Python

That’s essentially how while loops work, pretty simple, but extremely powerful. Take a look at a loop in Python below to get a feel for how a while loop is put together.

index = 1

while index <= 5:

print(index)

index += 1

As you can see above, on each iteration of the loop a variable is incremented to keep track of how many times we’ve printed to the screen.

The boolean expression, or the loop guard is exactly the same as we would write in an if statement, it's just that this boolean expression is evaluated before every iteration of the loop.

It basically determines whether or not we can keep looping again.

**Do While Loops**

Do While loops are basically while loops but in reverse. In a While loop you check the boolean condition first and then, if it’s true you execute the code in the loop body. But in a Do While Loop you first execute the code in the loop body and then check the boolean expression. Check out an example below

index = 1

while True:

print(index)

index += 1

if index > 5:

break;

Do While Loops are not used very often, but they can be extremely useful when trying to solve specific problems.

Guessing Game

secret\_word = "giraffe"

guess = ""

guess\_count = 0

guess\_limit = 3

out\_of\_guesses = False

while guess != secret\_word and not(out\_of\_guesses):

if guess\_count < guess\_limit:

guess = input("Enter a guess: ")

guess\_count += 1

else:

out\_of\_guesses = True

if out\_of\_guesses:

print("You Lose!")

else:

print("You Win!")

**For Loops**

for index in range(5):

print(index)

lucky\_nums = [4, 8, 15, 16, 23, 42]

for lucky\_num in lucky\_nums:

print(lucky\_num)

for letter in "Giraffe":

print(letter)

**Exponent Function**

def raise\_to\_power(base\_num, pow\_num):

result = 1

for index in range(pow\_num):

result \*= base\_num

return result

**2d Arrays & Nested Loops**

number\_grid = [ [1, 2], [3, 4] ]

number\_grid[0][1] = 99

print(number\_grid[0][0])

print(number\_grid[0][1])

for row in number\_grid:

for row in col:

print(col)

**Building A Translator**

def translate(phrase):

translation = ""

for letter in phrase:

if letter.lower() in "aeiou":

if letter.isupper():

translation = translation + "G"

else:

translation = translation + "g"

else:

translation = translation + letter

return translation

**Comments**

As you continue to sharpen your Python skills and get more and more comfortable writing programs, you'll eventually become really good at reading code. Just by looking at a line or two you'll be able to tell what the program is doing, and what's going to happen.

But, in a lot of cases, merely relying on the code to properly explain or document itself can be ineffective, and often times in the course of writing a program you'll want to mark things down or explain why you're doing things a certain way. This is where comments can be your best friend.

A comment is basically a piece of code in the program that Python will ignore. Unlike all of the really cool and interesting commands we've learned so far, comment's actually don't do anything... Well at least not as far as the computer is concerned.

Comments are generally used to make small notes to yourself or other developers, explain particularly difficult sections of code, or even temporarily remove parts of the code altogether.

You can write a comment like so

/'''

multi

line

comment

'''

# single

# line

# comment

As you can see there's two basic types of comments: single line and multi line.

Single Line Comments

In a single line comment the rules are that anything that comes after the # on the same line is ignored by Python when it executes. A lot of times developers will use these to temporarily remove a line of code by putting the # in front of it.

These single line comments are also often used to describe or give more context to a line of code. As a general rule it's always preferred that you write clear, descriptive code so that comments like this aren't necessary, but every once in a while a well placed comment can clear up confusion.

Generally a comment like the one described above, is placed either directly above the line of code it's describing, or immediately to the right of it. I always prefer to put these types of comments on top of the line of code because otherwise you'll find that one single line of code ends up being really long, which can get annoying.

Single line comments are also commonly used as TODO markers. If you see a comment that says TODO, that generally means that this is a part of the code which has yet to be implemented, and therefore the developer who’s working on the program has TO DO it.

Multi line comments

Multi line comments have a starting and an ending tag, as you can see in the code above. These are generally more convenient for commenting out large blocks of code.

A lot of times you’ll see developers put different kinds of documentation in comments like this. In fact, in many programs, comments are used to help generate documentation.

Sometimes you’ll encounter code files with more comments than actual executable code, and it’s because all of the official documentation for the codebase is generated in part from the comments in each file.

It’s important to distinguish between comments meant for documentation purposes, and comments meant to help explain messy code. Generally using a comment to cover up for bad code is a no no, but using comments in combination with some sort of documentation generating system is good!

Using Comments Wisely

There are no hard set rules for when you should and shouldn’t use comments when writing programs. In a lot of cases it’s just personal preference.

As I mentioned above however it’s usually a bad idea to rely on comments to make your code understandable. If you’re looking at a line of code, and can’t tell what it’s doing without reading a corresponding comment, then, except for in rare cases, that line of code can probably be re-written.

Personally, I really like using comments to comment out parts of my code when I’m writing it. As you write more and more complex programs, a lot of times you’ll run into problems where the code is crashing at certain points. In cases like this, I will generally comment out, that is to say, surround with comments, the code I think is causing the problem, and if the code starts working again then I’ve found the culprit.

Wrapping Up

At the end of the day comments are a great tool which can help with documenting code and giving you an easy way to quickly remove code without actually deleting it.

As we go forward in the course I would encourage you to start using comments to enhance your code and look for ways they can make your life easier as a developer!

Try Except

# code asks user for number and divides 10 by it

# enter '0' to trigger exception

try:

answer = 10 / int(input("Enter Number: "))

except ZeroDivisionError as e:

print(e)

except:

print("Caught any exception")

Reading Files

employee\_file = open("employees.txt", "r")

for employee in employee\_file.read\_lines():

print(employee)

employee\_file.close()

Writing Files

employee\_file = open("employees.txt", "w") # also try "a" for append

employee\_file.write("\nKelly - Customer Service")

employee\_file.close()

Modules

import module\_name

import module2\_name as module2

Classes & Objects

One of the best things about Python is it's ability to allow us to work with and keep track of a bunch of data. In fact data is the driving force behind most programs that you'll write. Programmers are constantly parsing strings, adding and subtracting numbers, counting things, and passing different pieces of data around.

In all major applications, being able to work with data in an organized and structured way is extremely important. Large applications are often faced with displaying, storing and managing complex pieces of information about a variety of things.

One of the limitations of our programming ability up to this point in the course is that we've only learned how to work with very limited types of data. In fact, if you think about it, we're only capable of representing and working with text, numbers and boolean values.

These limited types of data are going to pose a problem to us if we want to write more and more complex programs going forward. The problem is that outside of actuall numbers, text and true/false information, most of the things that we interact with and use in the real world can't be easily stored in a string, number or boolean.

Building a book

Imagine for example if we were writing a program for a library. In this program we wanted to keep track of and work with a bunch of books. The question arises, how do we store these books?

We can't store them in booleans, we might be able to store how many pages they have and their publishing dates as numbers, and maybe we could store the author's name and title in strings. But hopefully it's clear to you how there isn't an easy solution to representing a book with the limited data types we're giving. At the very least the book is spread out across several variables.

Object Oriented Programming (OOP)

In the early days of computer programming, developers just kinda bit the bullet and did their best to write programs in the way I just described. But eventually a better solution was devised called object oriented programming.

Object oriented programming allows developers to create their own data types. So in addition to strings, booleans and numbers, a developer could create a custom Book data type. This book data type could then be stored in variables, passed into functions, printed out, and anything else that the other basic data types could do.

By creating your own custom data types in your programs, you can theoretically model any real world item or entity and use it just like you would a string or a number.

Most real world applications you see, whether it's facebook, google, gruhhub, or the small programs we'll write going forward in the course, will use these custom data types to better organize complex entities like books.

Classes

So how do we create these custom data types? It's actually pretty simple and we can do it by creating a class. A class is a specification for a data type in our program. Because we're creating custom data types, we first have to describe what they look like and how they behave so Python knows what to do with them.

A good way to think of classes is as blueprints. They describe what the data type should look like and lay out all of the attributes and features that will be apart of the data type.

Python already provides us with our three base data types, booleans, strings (text) and numbers. When we create our custom data types, they will be made up of those base data types. Therefore when creating a class we can specify the attributes that the class will contain in terms of these base data types.

Let's take a look at what a class would look like for our Book:

class Book:

def \_\_init\_\_(self, title, author, numPages):

self.title = title

self.author = author

self.numPages = numPages

Remember, a class is a specification, a blueprint, it describes the structure and makeup of this new book data type.

In the above code, we've specified that a book is comprised of a title, author and number of pages. Notice how this book data type is comprised of the base data types that we've been provided by the language. Eventually we'll also be able to add functionality to this book class aswell, but let's save that for a future lesson.

Objects

So now that we've created our book class, Python knows what this new data type looks like, and we can start working with books in our program.

Since the book class is just a specification though, we can't use it in our program, instead we have to create what are called objects. An object is an instance of a class. In this case an object will be an actual book, with an actual title, author and number of pages.

This is an important terminology point. A class is a blueprint of a data type, an object is an instance of that blueprint.

Let's create a book object in Python:

class Book:

def \_\_init\_\_(self, title, author, numPages):

self.title = title

self.author = author

self.numPages = numPages

book1 = Book("Harry Potter", "JK Rowling", 500);

# book1.title = "Half-Blood Prince"

print(book1.title)

So now, we've created an object. It's a book with a title, author and number of pages! As far as Python is concerned this is basically the same as any of the other data types we've been working with throughout this course.

In the case of the book object, we've assigned it a number of attributes, which we can then access and modify just like we normally would.

The title and author are just strings, and the number of pages is a number. In order to access theses attributes we use a .. followed by the name of the attribute. This process is called de-referencing the book object.

Wrapping it up

So there you have it! Classes and Objects are notorious for tripping up new programmers, but as long as you learn to see them as just more complex custom data types, then a lot of the mystery disappears.

We created a book class and object, but remember that you can create classes to model anything. Try playing around with some of your own custom classes, or adding more attributes to our book! Also don't forget to check out the video above where I'll walk you through writing this code line by line!

Building A Quiz

class Question:

def \_\_init\_\_(self, prompt, answer):

self.prompt = prompt

self.answer = answer

question\_prompts = [

"What color are apples?\n(a) Red/Green\n(b)Orange",

"What color are bananas?\n(a) Red/Green\n(b)Yellow",

]

questions = [

Question(question\_prompts[0], "a"),

Question(question\_prompts[1], "b"),

]

def run\_quiz(questions):

score = 0

for question in questions:

answer = input(question.prompt)

if answer == question.answer:

score += 1

print("you got", score, "out of", len(questions))

run\_quiz(questions)

Class Methods

class Student:

def \_\_init\_\_(self, name, major, gpa):

self.name = name

self.major = major

self.gpa = gpa

def has\_honors(self):

if self.gpa >= 3.5:

return True

return False

Inheritance

class Chef:

def make\_chicken(self):

print("The chef makes chicken")

def make\_salad(self):

print("The chef makes salad")

def make\_special\_dish(self):

print("The chef makes bbq ribs")

class ItalianChef(Chef):

def make\_pasta(self):

print("The chef makes pasta")

def make\_special\_dish(self):

print("The chef makes chicken parm")

myChef = Chef()

myChef.make\_chicken()

myItalianChef = ItalianChef()

myItalianChef.make\_chicken()