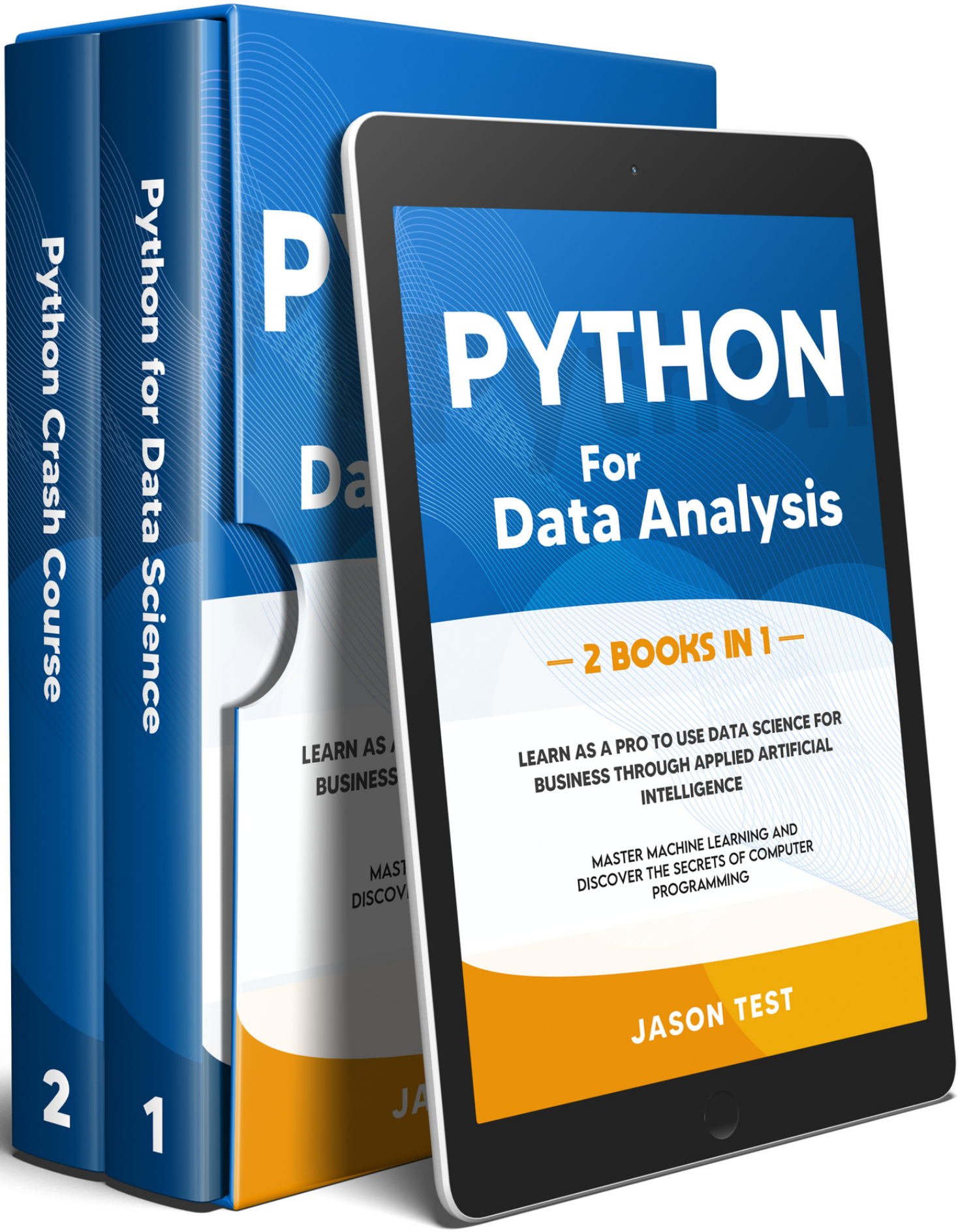

Python Crash Course
2
Python for Data Science
1
PYTHON
For
Data Analysis
— 2 BOOKS IN 1 —
LEARN AS A PRO TO USE DATA SCIENCE FOR BUSINESS THROUGH APPLIED ARTIFICIAL INTELLIGENCE
MASTER MACHINE LEARNING AND DISCOVER THE SECRETS OF COMPUTER PROGRAMMING
JASON TEST


Python for Data
Python Crash

PYTHON
FOR DATA ANALYSIS

2 BOOKS IN 1

The ultimate guide to learn as a PRO to use data science for business trough applied artificial intelligence. Master machine learning and discover the secrets of computer programming .

JASON TEST

PYTHO N FOR DATA SCIENCE

JASON TEST

TABLE OF CONTEN T

INTRODUCTION

Data Science has been very popular over the last couple of years. The main focus of this sector is to incorporate significant data into business and marketing strategies that will help a business expand. And get to a logical solution, the data can be stored and explored. Originally only the leading IT corporations were engaged throughout this field, but today information technology is being used by companies operating in different sectors and fields such as e-commerce, medical care, financial services, and others. Software processing programs such as Hadoop, R code, SAS, SQL, and plenty more are available. Python is, however, the most famous and easiest to use data and analytics tools. It is recognized as the coding world's Swiss Army Knife since it promotes structured coding, object-oriented programming, the operational programming language, and many others. Python is the most widely used programming language in the world and is also recognized as the most high - level language for data science tools and techniques, according to the 2018 Stack Overflow study.

In the Hacker rank 2018 developer poll, which is seen in their love-hate ranking, Python has won the developer's hearts. Experts in data science expect to see an increase in the Python ecosystem, with growing popularity. And although your journey to study Python programming may just start, it's nice to know that there are also plentiful (and increasing) career options.

Data analytics Python programming is extensively used and, along with being a flexible and open-source language, becomes one of the favorite programming languages. Its large libraries are used for data processing, and even for a beginner data analyst, they are very easy to understand. Besides being open-source, it also integrates easily with any infrastructure that can be used to fix the most complicated problems. It is used by most banks for data crunching, organizations for analysis and processing, and weather prediction firms such as Climate monitor analytics often use it. The annual wage for a Computer Scientist is $127,918, according to Indeed. So here's the good news, the figure is likely to increase. IBM's experts forecast a 28 percent increase in data scientists' demands by 2020. For data science, however, the future is bright, and Python is just one slice of the golden pie.

Luckily mastering Python and other principles of programming are as practical as ever.

DATA SCIENCE AND ITS SIGNIFICANCE

Data Science has come a long way from the past few years, and thus, it becomes an important factor in understanding the workings of multiple companies. Below are several explanations that prove data science will still be an integral part of the global market.

1. The companies would be able to understand their client in a more efficient and high manner with the help of Data Science. Satisfied customers form the foundation of every company, and they play an important role in their successes or failures. Data Science allows companies to engage with customers in the advance way and thus proves the product's improved performance and strength.

2. Data Science enables brands to deliver powerful and engaging visuals. That's one of the reasons it's famous. When products and companies make inclusive use of this data, they can share their experiences with their audiences and thus create better relations with the item.

3. Perhaps one Data Science's significant characteristics are that its results can be generalized to almost all kinds of industries, such as travel, health care, and education. The companies can quickly determine their problems with the help of Data Science, and can also adequately address them

4. Currently, data science is accessible in almost all industries, and nowadays, there is a huge amount of data existing in the world, and if used adequately, it can lead to victory or failure of any project. If data is used properly, it will be important in the future to achieve the product 's goals.

5. Big data is always on the rise and growing. Big data allows the enterprise to address complicated Business, human capital, and capital management problems effectively and quickly using different resources that are built routinely.

6. Data science is gaining rapid popularity in every other sector and therefore plays an important role in every product's functioning and performance. Thus, the data scientist's role is also enhanced as they will

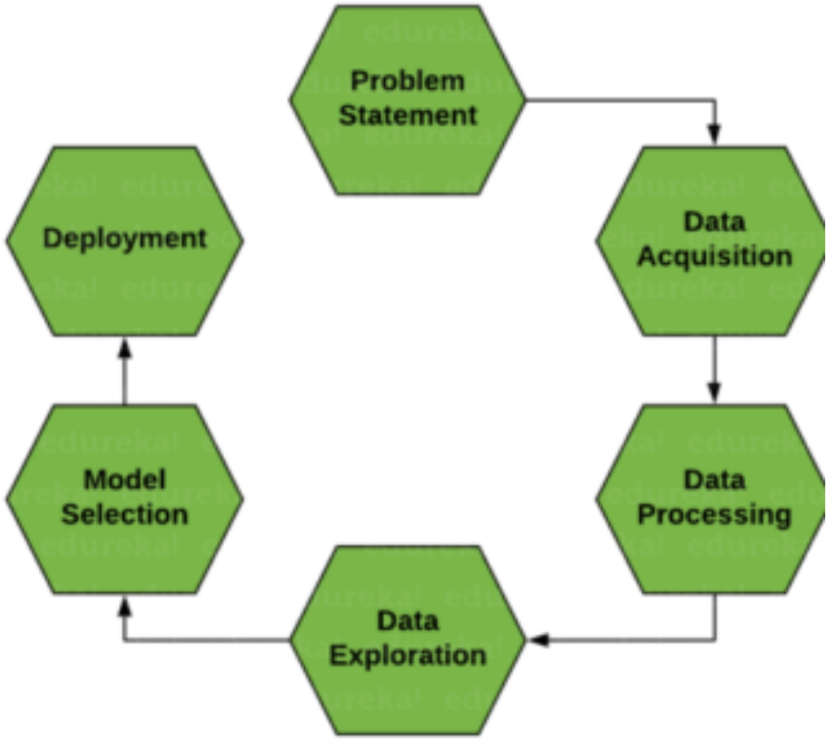
conduct an essential function of managing data and providing solutions to particular issues.

7. Computer technology has also affected the supermarket sectors. To understand this, let's take an example the older people had a fantastic interaction with the local seller. Also, the seller was able to meet the customers' requirements in a personalized way. But now this attention was lost due to the emergence and increase of supermarket chains. But the sellers are able to communicate with their customers with the help of data analytics.

8. Data Science helps companies build that customer connection. Companies and their goods will be able to have a better and deeper understanding of how clients can utilize their services with the help of data science.

Data Technology Future: Like other areas are continually evolving, the importance of data technology is increasingly growing as well. Data science impacted different fields. Its influence can be seen in many industries, such as retail, healthcare, and education. New treatments and technologies are being continually identified in the healthcare sector, and there is a need for quality patient care. The healthcare industry can find a solution with the help of data science techniques that helps the patients to take care with. Education is another field where one can clearly see the advantage of data science. Now the new innovations like phones and tablets have become an essential characteristic of the educational system. Also, with the help of data science, the students are creating greater chances, which leads to improving their knowledge.

Data Science Life Cycle:



Data Structures

A data structure may be selected in computer programming or designed to store data for the purpose of working with different algorithms on it. Every other data structure includes the data values, data relationships, and functions between the data that can be applied to the data and information.

Features of data structures

Sometimes, data structures are categorized according to their characteristics. Possible functions are:

- Linear or non-linear: This feature defines how the data objects are organized in a sequential series, like a list or in an unordered sequence, like a table.
- Homogeneous or non-homogeneous: This function defines how all data objects in a collection are of the same type or of different kinds.
- Static or dynamic: This technique determines to show to assemble the data structures. Static data structures at compilation time have fixed sizes, structures, and destinations in the memory.

Dynamic data types have dimensions, mechanisms, and destinations of memory that may shrink or expand depending on the application.

Data structure Types

Types of the data structure are determined by what sorts of operations will be needed or what kinds of algorithms will be implemented. This includes:

Arrays: An array stores a list of memory items at adjacent locations. Components of the same category are located together since each element's position can be easily calculated or accessed. Arrays can be fixed in size or flexible in length.

Stacks: A stack holds a set of objects in linear order added to operations. This order may be past due in first out (LIFO) or first-out (FIFO).

Queues: A queue stores a stack-like selection of elements; however, the sequence of activity can only be first in the first out. Linked lists: In a linear order, a linked list stores a selection of items. In a linked list, every unit or node includes a data item as well as a reference or relation to the next element in the list.

Trees: A tree stocks an abstract, hierarchical collection of items. Each node is connected to other nodes and can have several sub-values, also known as a child.

Graphs: A graph stores a non-linear design group of items. Graphs consist of a limited set of nodes, also called vertices, and lines connecting them, also known as edges. They are useful for describing processes in real life, such as networked computers.

Tries: A tria or query tree is often a data structure that stores strings as data files, which can be arranged in a visual graph.

Hash tables: A hash table or hash chart is contained in a relational list that labels the keys to variables. A hash table uses a hashing algorithm to transform an index into an array of containers containing the desired item of data. These data systems are called complex because they can contain vast quantities of interconnected data. Examples of primal, or fundamental, data structures are integer, float, boolean, and character.

Utilization of data structures

Data structures are generally used to incorporate the data types in physical forms. This can be interpreted into a wide range of applications, including a binary tree showing a database table. Data structures are used in the programming languages to organize code and information in digital storage. Python databases and dictionaries, or JavaScript array and objects, are popular coding systems used to gather and analyze data. Also, data structures are a vital part of effective software design. Significance of Databases Data systems is necessary to effectively handle vast volumes of data, such as data stored in libraries, or indexing services.

Accurate data configuration management requires memory allocation identifier, data interconnections, and data processes, all of which support the data structures. In addition, it is important to not only use data structures but also to select the correct data structure for each assignment.

Choosing an unsatisfactory data structure could lead to slow running times or disoriented code. Any considerations that need to be noticed when choosing a data system include what type of information should be processed, where new data will be put, how data will be organized, and how much space will be allocated for the data.

PYTHON BASICS

ou can get all the knowledge about the Python programming language in 5 simple steps.

Y Step 1: Practice Basics in Python

It all starts somewhere. This first step is where the basics of programming Python will be learned. You are always going to want an introduction to data science. Jupyter Notebook, which comes pre-portioned with Python libraries to help you understand these two factors, which make it one of the essential resources which you can start using early on your journey.

Step 2: Try practicing Mini-Python Projects

We strongly believe in learning through shoulders-on. Try programming stuff like internet games, calculators, or software that gets Google weather in your area. Creating these mini-projects can help you understand Python. Projects like these are standard for any coding languages, and a fantastic way to strengthen your working knowledge. You will come up with better and advance API knowledge, and you will continue site scraping with advanced techniques. This will enable you to learn Python programming more effectively, and the web scraping method will be useful to you later when collecting data.

Stage 3: Learn Scientific Libraries on Python

Python can do anything with data. Pandas, Matplotliband, and NumPyare are known to be the three best used and most important Python Libraries for data science. NumPy and Pandas are useful for data creation and development. Matplotlib is a library for analyzing the data, creating flow charts and diagrams as you would like to see in Excel or Google Sheets.

Stage 4: Create a portfolio

A portfolio is an absolute need for professional data scientists. These projects must include numerous data sets and leave important perspectives to readers that you have gleaned. Your portfolio does not have a specific theme; finding datasets that inspire you, and then finding a way to place them together. Showing projects like these provide some collaboration to

fellow data scientists, and demonstrates future employers that you have really taken the chance to understand Python and other essential coding skills. Some of the good things about data science are that, while showcasing the skills you've learned, your portfolio serves as a resume, such as Python programming.

Step 5: Apply Advanced Data Science Techniques

Eventually, the target is to strengthen your programming skills. Your data science path will be full of continuous learning, but you can accomplish specialized tutorials to make sure you have specialized in the basic programming of Python. You need to get confident with clustering models of regression, grouping, and k-means. You can also leap into machine learning-using sci-kit lessons to bootstrap models and create neural network models. At this point, developer programming could include creating models using live data sources. This type of machine learning technique adjusts s its assumptions over time.

How significant is Python for Data Science?

Efficient and simple to use – Python is considered a tool for beginners, and any student or researcher with only basic understanding could start working on it. Time and money spent debugging codes and constraints on different project management are also minimized. The time for code implementation is less compared to other programming languages such as C, Java, and C #, which makes developers and software engineers spend far more time working on their algorithms.

Library Choice-Python offers a vast library and machine learning and artificial intelligence database. Scikit Learn, TensorFlow, Seaborn, Pytorch, Matplotlib, and many more are among the most popular libraries. There are many online tutorial videos and resources on machine learning and data science, which can be easily obtained.

Scalability – Python has proven itself to be a highly scalable and faster language compared to other programming languages such as c++, Java, and R. It gives flexibility in solving problems that can't be solved with other computer languages. Many companies use it to develop all sorts of rapid techniques and systems.

#Visual Statistics and Graphics-Python provides a number of visualization tools. The Matplotlib library provides a reliable framework on

which those libraries such as gg plot, pandas plotting, PyTorch, and others are developed. These services help create graphs, plot lines ready for the Web, visual layouts, etc.

How Python is used for Data Science

#First phase – First of all, we need to learn and understand what form a data takes. If we perceive data to be a huge Excel sheet with columns and crows lakhs, then perhaps you should know what to do about that? You need to gather information into each row as well as column by executing some operations and searching for a specific type of data. Completing this type of computational task can consume a lot of time and hard work. Thus, you can use Python's libraries, such as Pandas and Numpy, that can complete the tasks quickly by using parallel computation.

#Second phase – The next hurdle is to get the data needed. Since data is not always readily accessible to us, we need to dump data from the network as needed. Here the Python Scrap and brilliant Soup libraries can enable us to retrieve data from the internet.

#Third phase – We must get the simulation or visual presentation of the data at this step. Driving perspectives gets difficult when you have too many figures on the board. The correct way to do that is to represent the data in graph form, graphs, and other layouts. The Python Seaborn and Matplotlib libraries are used to execute this operation.

#Fourth phase – The next stage is machine-learning, which is massively complicated computing. It includes mathematical tools such as the probability, calculus, and matrix operations of columns and rows over lakhs. With Python's machine learning library Scikit- Learn, all of this will become very simple and effective.

Standard Library

The Python Standard library consists of Python's precise syntax, token, and semantic. It comes packaged with deployment core Python. When we started with an introduction, we referenced this. It is written in C and covers features such as I / O and other core components. Together all of the versatility renders makes Python the language it is. At the root of the basic library, there are more than 200 key modules. Python ships that library. But

aside from this library, you can also obtain a massive collection of several thousand Python Package Index (PyPI) components.

1. Matplotlib

‘Matplotlib’ helps to analyze data, and is a library of numerical plots. For Data Science, we discussed in Python.

2. Pandas

‘Pandas’ is a must for data-science as we have said before. It provides easy, descriptive, and versatile data structures to deal with organized (tabulated, multilayered, presumably heterogeneous) and series data with ease (and fluidly).

3. Requests

‘Requests’ is a Python library that allows users to upload HTTP/1.1 requests, add headers, form data, multipart files, and simple Python dictionary parameters. In the same way, it also helps you to access the response data.

4. NumPy

It has basic arithmetic features and a rudimentary collection of scientific computing.

5. SQLAlchemy

It has sophisticated mathematical features, and SQLAlchemy is a basic mathematical programming library with well-known trends at a corporate level. It was created to make database availability efficient and high-performance.

6. BeautifulSoup

This may be a bit on the slow side. BeautifulSoup seems to have a superb library for beginner XML- and HTML- parsing.

7. Pyglet

Pyglet is an outstanding choice when designing games with an object-oriented programming language interface. It also sees use in the development of other visually rich programs for Mac OS X, Windows, and Linux in particular. In the 90s, they turned to play Minecraft on their PCs whenever people were bored. Pyglet is the mechanism powering Minecraft.

8. SciPy

Next available is SciPy, one of the libraries we spoke about so often. It does have a range of numerical routines that are user-friendly and effective. Those provide optimization routines and numerical integration procedures.

9. Scrapy

If your objective is quick, scraping at the high-level monitor and crawling the network, go for Scrapy. It can be used for data gathering activities for monitoring and test automation.

10. PyGame

PyGame offers an incredibly basic interface to the system-independent graphics, audio, and input libraries of the Popular Direct Media Library (SDL).

11. Python Twisted

Twisted is an event-driven networking library used in Python and authorized under the MIT open-source license.

12. Pillow

Pillow is a PIL (Python Imaging Library) friendly fork but is more user efficient. Pillow is your best friend when you're working with pictures.

13. pywin32

As the name suggests, this gives useful methods and classes for interacting with Windows.

14. wxPython

For Python, it's a wrapper around wxWidgets.

15. iPython

iPython Python Library provides a parallel distributed computing architecture. You will use it to create, run, test, and track parallel and distributed programming.

16. Nose

The nose provides alternate test exploration and test automation running processes. This intends to mimic the behavior of the py.test as much as possible.

17. Flask

Flask is a web framework, with a small core and several extensions.

18. SymPy

It is a library of open-source symbolic mathematics. SymPy is a full-fledged Computer Algebra System (CAS) with a very simple and easily understood code that is highly expandable. It is implemented in python, and therefore, external libraries are not required.

19. Fabric

As well as being a library, Fabric is a command-line tool to simplify the use of SSH for installation programs or network management activities. You can run local or remote command line, upload/download files, and even request input user going, or abort activity with it.

20. PyGTK

PyGTK allows you to create programs easily using a Python GUI (Graphical User Interface).

Operators and Expressions

Operators

In Python, operators are special symbols that perform mathematical operation computation. The value in which the operator is running on is called the operand.

Arithmetic operators

It is used by arithmetic operators to perform mathematical operations such as addition, subtraction, multiplying, etc.

Comparison operators

Comparison operators can be used for value comparisons. Depending on the condition, it returns either True or False.

Logical operators

Logical operators are and, or, not.

Operator	Meaning	Example
And	True if both operands are true	x and y
Or	True if either of the operands is true	x or y
Not	True if the operand is false (complements the operand)	not x

Bitwise operators

Bitwise operators operate as if they became binary-digit strings on operands. Bit by bit they work, and therefore the name. For example, in binary two is10, and in binary seven is 111.

Assignment operators

Python language's assignment operators are used to assign values to the variables. a = 5 is a simple task operator assigning 'a' value of 5 to the right of the variable 'a' to the left. In Python, there are various compound operators such as a + = 5, which adds to the variable as well as assigns the same later. This equals a= a + 5.

Special operators

Python language gives other different types of operators, such as the operator of the identity or the operator of membership. Examples of these are mentioned below.

Identity operators

'Is' and 'is not' are Python Identity Operators. They are used to test if there are two values or variables in the same memory section. Two equal variables do not mean they are equivalent.

Membership operator

The operators that are used to check whether or not there exists a value/variable in the sequence such as string, list, tuples, sets, and dictionary. These operators return either True or False if a variable is found in the list, it returns True, or else it returns False

Expressions

An expression is a mix of values, variables, operators, and function calls. There must be an evaluation of the expressions. When you ask Python to print a phrase, the interpreter will evaluate the expression and show the output.

Arithmetic conversions

Whenever an arithmetic operator interpretation below uses the phrase "the numeric arguments are converted to a common type," this means the execution of the operator for the built-in modes operates as follows
If one argument is a complex quantity, then the other is converted to a complex number; If another argument is a floating-point number, the other

argument is transformed to a floating-point; Or else both will be integers with no need for conversion.

Atoms

Atoms are the most important expressional components. The smallest atoms are literals or abstract identities. Forms contained in parentheses, brackets, or braces are also syntactically known as atoms. Atoms syntax is:

atom ::= identifier | enclosure| literal

enclosure ::= list_display| parenth_form| dict_display | set_display

Identifiers (Names)

A name is an identifier that occurs as an atom. See section Lexical Description Identifiers and Keywords and group Naming and binding for naming and binding documents. Whenever the name is connected to an entity, it yields the entity by evaluating the atom. When a name is not connected, an attempt to assess it elevates the exception for NameError.

Literals

Python provides logical string and bytes and numerical literals of different types:

literal::= string literal | bytes literal

| integer | float number | image number

Assessment of a literal yield with the predicted set an object of that type (bytes, integer, floating-point number, string, complex number). In the scenario of floating-point and imaginary (complex) literals, the value can be approximated.

Parenthesized forms

A parenthesized type is an available set of parentheses for the expression:

parenth_form ::= "(" [starred_expression] ")"

A list of parenthesized expressions yields whatever the list of expressions produces: if the list includes at least one comma, it produces a tuple. If not, it yields the sole expression that forms up the list of expressions. A null pair of parentheses generates an incomplete object of tuples. As all tuples are immutable, the same rules would apply as for literals (i.e., two empty tuple occurrences does or doesn't yield the same entity).

Displays for lists, sets, and dictionaries

For the construction of a list, Python uses a series or dictionary with a particular syntax called "displays," each in complementary strands:
The contents of the container are listed explicitly, or They are calculated using a series of instructions for looping and filtering, named a 'comprehension.' Common features of syntax for comprehensions are:

comprehension ::= assignment_expressioncomp_for

comp_for ::= ["async"] "for" target_list "in" or_test [comp_iter]

comp_iter ::= comp_for | comp_if

comp_if ::= "if" expression_nocond [comp_iter]

A comprehension contains one single sentence ready for at least one expression for clause, and zero or more for or if clauses. Throughout this situation, the components of the container are those that will be generated by assuming each of the for or if clauses as a block, nesting from left to right, and determining the phase for creating an entity each time the inner core block is approached.

List displays

A list view is a probably empty sequence of square brackets including expressions:

list_display ::= "[" [starred_list | comprehension] "]"

A list display generates a new column object, with either a list of expressions or a comprehension specifying the items. When a comma-separated database of expressions is provided, its elements are assessed from left to right and positioned in that order in the category entity. When Comprehension is provided, the list shall be built from the comprehension components.

Set displays

Curly braces denote a set display and can be distinguished from dictionary displays by the lack of colons dividing data types:

set_display ::= "{" (starred_list | comprehension) "}"

A set show offers a new, mutable set entity, with either a series of expressions or a comprehension defining the contents. When supplied with a comma-separated list of expressions, its elements are evaluated from left to right and assigned to the set entity. Whenever a comprehension is

provided, the set is formed from the comprehension-derived elements. Unable to build an empty set with this {}; literal forms a blank dictionary.

Dictionary displays

A dictionary view is a potentially empty sequence of key pairs limited to curly braces:

dict_display ::= "{" [key_datum_list | dict_comprehension] "}"

key_datum_list ::= key_datum ("," key_datum)* [","]

key_datum ::= expression ":" expression | "**" or_expr

dict_comprehension ::= expression ":" expression comp_for

The dictionary view shows a new object in the dictionary. When a comma-separated series of key / datum pairs is provided, they are analyzed from left to right to identify dictionary entries: each key entity is often used as a key to hold the respective datum in the dictionary. This implies you can clearly state the very same key numerous times in the key /datum catalog, but the last one given will become the final dictionary's value for that key.

Generator expressions

A generator expression is the compressed syntax of a generator in the parenthesis :

generator_expression ::= "(" expression comp_for ")"

An expression generator produces an entity that is a new generator. Its syntax will be the same as for comprehensions, except for being enclosed in brackets or curly braces rather than parentheses. Variables being used generator expression are assessed sloppily when the generator object (in the same style as standard generators) is called by the __next__() method. Conversely, the iterate-able expression in the leftmost part of the clause is evaluated immediately, such that an error that it produces is transmitted at the level where the expression of the generator is characterized, rather than at the level where the first value is recovered.

For instance: (x*y for x in range(10) for y in range(x, x+10)).

Yield expressions

yield_atom ::= "(" yield_expression ")"

yield_expression ::= "yield" [expression_list | "from" expression]

The produced expression is used to define a generator function or async generator function, and can therefore only be used in the function definition body. Using an expression of yield in the body of a function tends to cause that function to be a generator, and to use it in the body of an asynchronous def function induces that co-routine function to become an async generator. For example:

```
def gen():  # defines a generator function
yield 123
asyncdefagen(): # defines an asynchronous generator function
yield 123
```

Because of their adverse effects on the carrying scope, yield expressions are not allowed as part of the impliedly defined scopes used to enforce comprehensions and expressions of generators.

Input and Output of Data in Python

Python Output Using print() function

To display data into the standard display system (screen), we use the print() function. We may issue data to a server as well, but that will be addressed later. Below is an example of its use.

```
>>>>print('This sentence is output to the screen')
```

Output:

This sentence is output to the screen

Another example is given:

```
a = 5
print('The value of a is,' a)
```

Output:

The value of a is 5

Within the second declaration of print(), we will note that space has been inserted between the string and the variable value a. By default, it contains this syntax, but we can change it.

The actual syntax of the print() function will be:

```
print(*objects, sep=' ', end='\n', file=sys.stdout, flush=False)
```

Here, the object is the value(s) that will be printed. The sep separator between values is used. This switches into a character in space. Upon printing all the values, the finish is printed. It moves into a new section by design. The file is the object that prints the values, and its default value is sys.stdout (screen). Below is an example of this.

```
print(1, 2, 3, 4)
print(1, 2, 3, 4, sep='*')
print(1, 2, 3, 4, sep='#', end='&')
```

Run code

Output:

```
1 2 3 4
1*2*3*4
1#2#3#4&
```

Output formatting

Often we want to style our production, so it looks appealing. It can be done using the method str.format(). This technique is visible for any object with a string.

```
>>> x = 5; y = 10
>>>print('The value of x is {} and y is {}'.format(x,y))
```

Here the value of x is five and y is 10

Here, they use the curly braces{} as stand-ins. Using numbers (tuple index), we may specify the order in which they have been printed.

```
print('I love {0} and {1}'.format('bread','butter'))
print('I love {1} and {0}'.format('bread','butter'))
```

Run Code

Output:

```
I love bread and butter
I love butter and bread
```

People can also use arguments with keyword to format the string.

```
>>>print('Hello {name}, {greeting}'.format(greeting = 'Goodmorning', name = 'John'))
```

Hello John, Goodmorning

Unlike the old sprint() style used in the C programming language, we can also format strings. To accomplish this, we use the ‘%’ operator.

```
>>> x = 12.3456789
>>>print('The value of x is %3.2f' %x)
The value of x is 12.35
>>>print('The value of x is %3.4f' %x)
The value of x is 12.3457
```

Python Indentation

Indentation applies to the spaces at the start of a line of the compiler. Whereas indentation in code is for readability only in other programming languages, but the indentation in Python is very important. Python supports the indent to denote a code block.

Example

```
if 5 > 2:
print("Five is greater than two!")
```

Python will generate an error message if you skip the indentation:

Example

Syntax Error:

```
if 5 > 2:
print("Five is greater than two!")
```

Python Input

Our programs have been static. Variables were described or hard-coded in the source code. We would want to take the feedback from the user to allow flexibility. We have the input() function in Python to enable this. input() is syntax as:

```
input([prompt])
```

While prompt is the string we want to show on the computer, this is optional.

```
>>>num = input('Enter a number: ')
Enter a number: 10
>>>num
'10'
```

Below, we can see how the value 10 entered is a string and not a number. To transform this to a number we may use the functions int() or float().

```
>>>int('10')
10
>>>float('10')
10.0
```

The same method can be done with the feature eval(). Although it takes eval much further. It can even quantify expressions, provided that the input is a string

```
>>>int('2+3')
Traceback (most recent call last):
  File "<string>", line 301, in runcode
  File "<interactive input>", line 1, in <module>
ValueError: int() base 10 invalid literal: '2+3'
>>>eval('2+3')
5
```

Python Import

As our software gets larger, splitting it up into separate modules is a smart idea. A module is a file that contains definitions and statements from Python. Python packages have a filename, and the .py extension begins with it. Definitions may be loaded into another module or to the integrated Python interpreter within a module. To do this, we use the keyword on import.

For instance, by writing the line below, we can import the math module:

```
import math
```

We will use the module as follows:

```
import math
print(math.pi)
```

Run Code

Output

```
3.141592653589793
```

So far, all concepts are included in our framework within the math module. Developers can also only import certain particular attributes and functions, using the keyword.

For instance:

```
>>>from math import pi
>>>pi
3.141592653589793
```

Python looks at multiple positions specified in sys.path during the import of a module. It is a list of positions in a directory.

```
>>> import sys
>>>sys.path
['',
'C:\\Python33\\Lib\\idlelib',
'C:\\Windows\\system32\\python33.zip',
'C:\\Python33\\DLLs',
'C:\\Python33\\lib',
'C:\\Python33',
'C:\\Python33\\lib\\site-packages']
```

We can insert our own destination to that list as well.

FUNCTIONS

You utilize programming functions to combine a list of instructions that you're constantly using or that are better self-contained in sub-program complexity and are called upon when required. Which means a function is a type of code written to accomplish a given purpose. The function may or may not need various inputs to accomplish that particular task. Whenever the task is executed, one or even more values can or could not be returned by the function. Basically there exist three types of functions in Python language:

1. Built-in functions, including help() to ask for help, min() to just get the minimum amount, print() to print an attribute to the terminal. More of these functions can be found here.
2. User-Defined Functions (UDFs) that are functions created by users to assist and support them out;
3. Anonymous functions, also labeled lambda functions since they are not defined with the default keyword.

Defining A Function: User Defined Functions (UDFs)

The following four steps are for defining a function in Python:

1. Keyword def can be used to declare the function and then use the function name to backtrack.
2. Add function parameters: They must be within the function parentheses. Finish off your line with a colon.
3. Add statements which should be implemented by the functions.

When the function should output something, end your function with a return statement. Your task must return an object None without return declaration. Example:

```
1. def hello():
2.  print("Hello World")
3.return
```

It is obvious as you move forward, the functions will become more complex: you can include for loops, flow control, and more to make things more fine-grained:

```
def hello():
name = str(input("Enter your name: "))
if name:
print ("Hello " + str(name))
else:
print("Hello World")
return
hello()
```

In the feature above, you are asking the user to give a name. When no name is provided, the 'Hello World' function will be printed. Otherwise, the user will receive a custom "Hello" phrase. Also, consider you can specify one or more parameters for your UDFs function. When you discuss the segment Feature Statements, you will hear more about this. Consequently, as a result of your function, you may or may not return one or more values.

The return Statement

Note that since you're going to print something like that in your hello) (UDF, you don't really have to return it. There'll be no distinction between the above function and this one:

Example:

```
1. defhello_noreturn():
2. print("Hello World")
```

Even so, if you'd like to keep working with the result of your function and try a few other functions on it, you'll need to use the return statement to simply return a value, like a string, an integer. Check out the following scenario in which hello() returns a "hello" string while the hello_noreturn() function returns None:

```
1.      def hello():
2.       print("Hello World")
3.            return("hello")
4.       defhello_noreturn():
```

```
5.              print("Hello World")
6.       # Multiply the output of `hello()` with 2
7.         hello() * 2
8.       # (Try to) multiply the output of `hello_noreturn()` with 2
9.          hello_noreturn() * 2
```

The secondary part gives you an error because, with a None, you cannot perform any operations. You will get a TypeError that appears to say that NoneType (the None, which is the outcome of hello_noreturn()) and int (2) cannot do the multiplication operation. Tip functions leave instantly when a return statement is found, even though that means they will not return any result:

```
1.  def run():
2. for x in range(10):
3. if x == 2:
4. return
5.  print("Run!")
6.      run()
```

Another factor worth noting when dealing with the ‘return expression’ is many values can be returned using it. You consider making use of tuples for this. Recall that this data structure is very comparable to a list's: it can contain different values. Even so, tuples are immutable, meaning you can't alter any amounts stored in it! You build it with the aid of dual parentheses). With the assistance of the comma and the assignment operator, you can disassemble tuples into different variables.

Read the example below to understand how multiple values can be returned by your function:

```
1.   # Define `plus()`
2.     def plus(a,b):
3.sum = a + b
4.return (sum, a)

5.   # Call `plus()` and unpack variables
6.    sum, a = plus(3,4)
```

```
7.      # Print `sum()`
8.          print(sum)
```

Notice that the return statement sum, 'a' will result in just the same as the return (sum, a): the earlier simply packs total and an in a tuple it under hood!

How To Call A Function

You've already seen a lot of examples in previous sections of how one can call a function. Trying to call a function means executing the function you have described-either directly from the Python prompt, or by a different function (as you have seen in the "Nested Functions" portion). Call your new added hello() function essentially by implementing hello() as in the DataCamp Light chunk as follows:

```
1.  hello()
```

Adding Docstrings to Python Functions

Further valuable points of Python's writing functions: docstrings. Docstrings define what your function does, like the algorithms it conducts or the values it returns. These definitions act as metadata for your function such that anybody who reads the docstring of your feature can understand what your feature is doing, without having to follow all the code in the function specification. Task docstrings are placed right after the feature header in the subsequent line and are set in triple quote marks. For your hello() function, a suitable docstring is 'Hello World prints.'

```
def hello():
"""Prints "Hello World".
Returns:
    None
"""
print("Hello World")
return
```

Notice that you can extend docstrings more than the one provided here as an example. If you want to study docstrings in more depth information, you should try checking out some Python library Github repositories like scikit-learn or pandas, in which you'll find lots of good examples!

Function Arguments in Python

You probably learned the distinction between definitions and statements earlier. In simple terms, arguments are the aspects that are given to any function or method call, while their parameter identities respond to the arguments in the function or method code. Python UDFs can take up four types of arguments:

1. Default arguments
2. Required arguments
3. Keyword arguments
4. Variable number of arguments

Default Arguments

Default arguments would be those who take default data if no value of the argument is delivered during the call function. With the assignment operator =, as in the following case, you may assign this default value:

```
#Define `plus()` function
def plus(a,b = 2):
return a + b
# Call `plus()` with only `a` parameter
plus(a=1)
# Call `plus()` with `a` and `b` parameters
plus(a=1, b=3)
```

Required Arguments

Because the name sort of brings out, the claims a UDF needs are those that will be in there. Such statements must be transferred during the function call and are absolutely the right order, such as in the example below:

```
# Define `plus()` with required arguments
def plus(a,b):
return a + b
```

Calling the functions without getting any additional errors, you need arguments that map to 'a' as well as the 'b' parameters. The result will not be

unique if you swap round the 'a' and 'b,' but it could be if you modify plus() to the following:

```
1. # Define `plus()` with required arguments
2. def plus(a,b):
3.return a/b
```

Keyword Arguments

You will use keyword arguments in your function call if you'd like to make sure you list all the parameters in the correct order. You use this to define the statements by the name of the function. Let's take an example above to make it a little simpler:

```
1. # Define `plus()` function
2. def plus(a,b):
3.return a + b
4. # Call `plus()` function with parameters
5. plus(2,3)
6. # Call `plus()` function with keyword arguments
7.  plus(a=1, b=2)
```

Notice that you can also alter the sequence of the parameters utilizing keywords arguments and still get the same outcome when executing the function:

```
1. # Define `plus()` function
2. def plus(a,b):
3.return a + b
4. # Call `plus()` function with keyword arguments
5.  plus(b=2, a=1)
```

Global vs. Local Variables

Variables identified within a function structure usually have a local scope, and those specified outside have a global scope. This shows that the local variables are specified within a function block and can only be retrieved through that function, while global variables can be retrieved from all the functions in the coding:

```
1. # Global variable `init`
```

2. init = 1

3. # Define `plus()` function to accept a variable number of arguments

4. def plus(*args):

5. # Local variable `sum()`

6.total = 0

7.fori in args:

8.total += i

9.return total

10.# Access the global variable

11.print("this is the initialized value " + str(init))

12.# (Try to) access the local variable

13.print("this is the sum " + str(total))

You will find that you can get a NameError that means the name 'total' is not specified as you attempt to print out the total local variable that was specified within the body of the feature. In comparison, the init attribute can be written out without any complications.

Anonymous Functions in Python

Anonymous functions are often termed lambda functions in Python since you are using the lambda keyword rather than naming it with the standard-def keyword.

1. double = lambda x: x*2
2. double(5)

The anonymous or lambda feature in the DataCamp Light chunk above is lambda x: x*2. X is the argument, and x*2 is the interpretation or instruction that is analyzed and given back. What is unique about this function, and it has no tag, like the examples you saw in the first section of the lecture for this function. When you had to write the above function in a UDF, you would get the following result:

def double(x):

return x*2

Let us see another example of a lambda function where two arguments are used:

```
# `sum()` lambda function
  sum = lambda x, y: x + y;
 # Call the `sum()` anonymous function
  sum(4,5)
# "Translate" to a UDF
 def sum(x, y):
returnx+y
```

When you need a function with no name for a short interval of time, you utilize anonymous functions and this is generated at runtime. Special contexts where this is important are when operating with filter(), map() and redu():

```
from functools import reduce
 my_list = [1,2,3,4,5,6,7,8,9,10]
 # Use lambda function with `filter()`
 filtered_list = list(filter(lambda x: (x*2 > 10), my_list))
 # Use lambda function with `map()`
mapped_list = list(map(lambda x: x*2, my_list))
 # Use lambda function with `reduce()`
 reduced_list = reduce(lambda x, y: x+y, my_list)
    print(filtered_list)
   print(mapped_list)
  print(reduced_list)
```

As the name states the filter() function it help filters the original list of inputs my_list based on a criterion > 10.By contrast, with map(), you implement a function to every components in the my_listlist. You multiply all of the components with two in this scenario. Remember that the function reduce() is a portion of the functools library. You cumulatively are using this function to the components in the my_list() list, from left to right, and in this situation decrease the sequence to a single value 55.

Using main() as a Function

If you have got any knowledge with other programming languages like java, you'll notice that executing functions requires the main feature. As

you've known in the above examples, Python doesn't really require this. However, it can be helpful to logically organize your code along with a main() function in your python code- - all of the most important components are contained within this main() function.

You could even simply achieve and call a main() function the same as you did with all of those above functions:

```
1. # Define `main()` function
2. def main():
3. hello()
4. print("This is the main function")
5. main()
```

After all, as it now appears, when you load it as a module, the script of your main ( ) function will indeed be called. You invoke the main() function whenever name == ' main ' to ensure this does not happen.

That implies the source above code script becomes:

```
1.# Define `main()` function
2.def main():
3.hello()
4.print("This is a main function")
5.# Execute `main()` function
6.  if __name__ == '__main__':
7. main()
```

Remember that in addition to the main function, you too have a init function, which validates a class or object instance. Plainly defined, it operates as a constructor or initializer, and is termed automatically when you start a new class instance. With such a function, the freshly formed object is assigned to the self-parameter that you've seen in this guide earlier.

Consider the following example:

```
class Dog:
    """
    Requires:
legs – legs for a dog to walk.
```

```
color – Fur color.
    """
def __init__(self, legs, color):
self.legs = legs
self.color = color
def bark(self):
bark = "bark" * 2
return bark
if __name__ == "__main__":
dog = Dog(4, "brown")
bark = dog.bark()
print(bark)
```

LISTS AND LOOPS

Lists

A list is often a data structure in Python, which is an ordered list of elements that is mutable or modifiable. An item is named for each element or value inside a list. Just like strings are defined like characters between quotations, lists are specified by square brackets '[ ]' having values.

Lists are nice to have because you have other similar principles to deal with. They help you to hold data that are relevant intact, compress the code, and run the same numerous-value methods and processes at once.

It could be helpful to get all the several lists you have on your computer when beginning to think about Python lists as well as other data structures that are types of collections: Your assemblage of files, song playlists, browser bookmarks, emails, video collections that you can access through a streaming platform and much more.

We must function with this data table, taken from data collection of the Mobile App Store (RamanathanPerumal):

Name	price	currency	rating_count	rating
Instagram	0.0	USD	2161558	4.5
Clash of Clans	0.0	USD	2130805	4.5
Temple Run	0.0	USD	1724546	4.5
Pandora – Music & Radio	0.0	USD	1126879	4.0
Facebook	0.0	USD	2974676	3.5

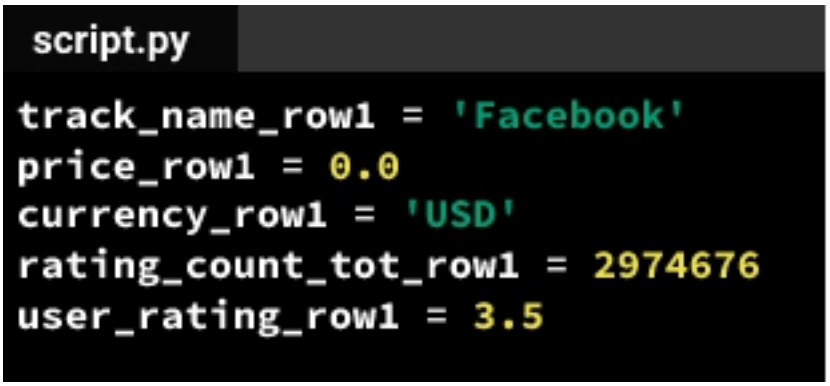
Every value is a data point in the table. The first row (just after titles of the columns) for example has 5 data points:

- Facebook
- 0.0

- USD
- 2974676
- 3.5

Dataset consists of a collection of data points. We can consider the above table as a list of data points. Therefore we consider the entire list a dataset. We can see there are five rows and five columns to our data set.

Utilizing our insight of the Python types, we could perhaps consider we can store each data point in their own variable — for example, here's how we can store the data points of the first row:



Above, we stored:

- Text for the string as “Facebook.”
- Float 0.0 as a price
- Text for the string as “USD.”
- Integer 2,974,676 as a rating count
- Float 3.5 for user rating

A complicated process would be to create a variable for every data point in our data set. Luckily we can use lists to store data more effectively. So in the first row, we can draw up a list of data points:

script.py

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
print(row_1)
type(row_1)
```

Output

```
['Facebook', 0.0, 'USD', 2974676, 3.5]
list
```

For list creation, we:

- Separating each with a comma while typing out a sequence of data points: 'Facebook,' 0.0, 'USD,' 2974676, 3.5
- Closing the list with square brackets: ['Facebook', 0.0, 'USD', 2974676, 3.5]
- After the list is created, we assign it to a variable named row_1and the list is stored in the computer's memory.

For creating data points list, we only need to:

- Add comma to the data points for separation.
- Closing the list with brackets.

See below as we create five lists, in the dataset each row with one list:

```
row_1 =]['FACEBOOK', 0.0, 'usd', 2974676, 3.5]

row_2 = ['INSTAGRAM', 0.0, 'usd', 2161558, 4.5

row_3 = ['CLASH OF CLANS', 0.0, 'usd', 2130805, 4.5]

row_4 = ['TEMPLE RUN', 0.0, 'usd', 1724546, 4.5]

row_5 =['PANDORA', 0.0, 'usd', 1126879, 4.0]
```

Index of Python Lists

A list could include a broader array of data types. A list containing [4, 5, 6] includes the same types of data (only integers), while the list ['Facebook', 0.0, 'USD,' 2974676, 3.5] contains many types of data:

- Consisting Two types of floats (0.0, 3.5)
- Consisting One type of integer (2974676)
- Consisting two types of strings ('Facebook,' 'USD')

`['FACEBOOK', 0.0, 'usd', 2974676, 3.5]` list got 5 data points. For the length of a list, len() command can be used:

script.py

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
print(len(row_1))

list_1 = [1, 6, 0]
print(len(list_1))

list_2 = []
print(len(list_2))
```

Output

```
5
3
0
```

For smaller lists, we can simply count the data points on our displays to figure the length, but perhaps the len() command will claim to be very useful anytime you function with lists containing many components, or just need to compose data code where you really don't know the length in advance.

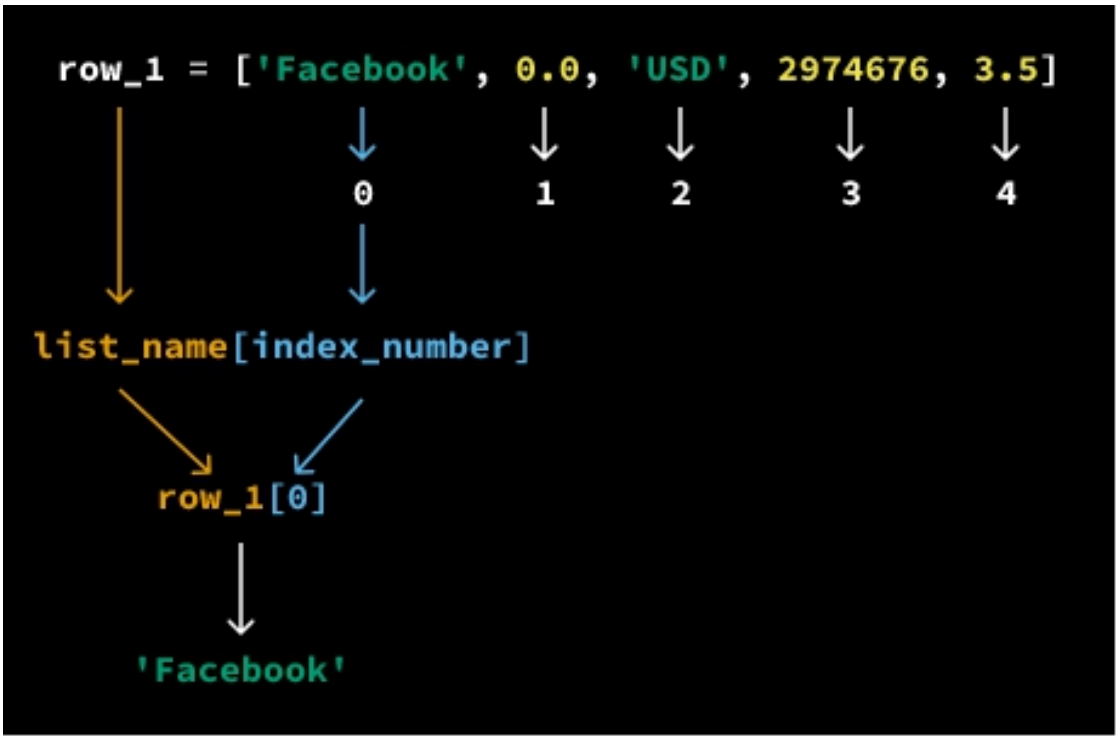
Every other component (data point) in a list is linked to a particular number, termed the index number. The indexing also begins at 0, which means that the first element should have the index number 0, the 2nd element the index number 1, etc.

To locate a list element index rapidly, determine its location number in the list and then subtract it by 1. The string 'USD,' for instance, is the third item in the list (stance number 3), well its index number must be two because 3 – 1 = 2.

The index numbers enable us to locate a single item from a list. Going backward through the list row 1 from the example above, by executing code row 1[0], we can obtain the first node (the string 'Facebook') of index number 0.

```
script.py
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
row_1[0]

Output
'Facebook'
```

The Model list_name[index number] follows the syntax for locating specific list components. For example, the title of our list above is row_1 and the index number of a first element is 0, we get row_1[0] continuing to follow the list_name[index number] model, in which the index number 0 is in square brackets just after the name of the variable row_1.



The method to retrieve each element in row_1:

script.py

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
          0        1     2        3       4
                  Index numbers

print(row_1[0])
print(row_1[1])
print(row_1[2])
print(row_1[3])
print(row_1[4])
```

Output

```
Facebook
0.0
USD
2974676
3.5
```

Retrieval of list elements makes processes easier to execute. For example, Facebook and Instagram ratings can be selected, and the aggregate or distinction between the two can be found:

script.py

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
row_2 = ['Instagram', 0.0, 'USD', 2161558, 4.5]

difference = row_2[4] - row_1[4]
average_rating = (row_1[4] + row_2[4]) / 2

print(difference)
print(average_rating)
```

Output

```
1.0
4.0
```

Try Using list indexing to retrieve and then average the number of ratings with the first 3 rows:

ratings_1 = row_1[3]

ratings_2 = row_2[3]

ratings_3 = row_3[3]

total = ratings_1 + ratings_2 + ratings_3

average = total / 3

print(average)

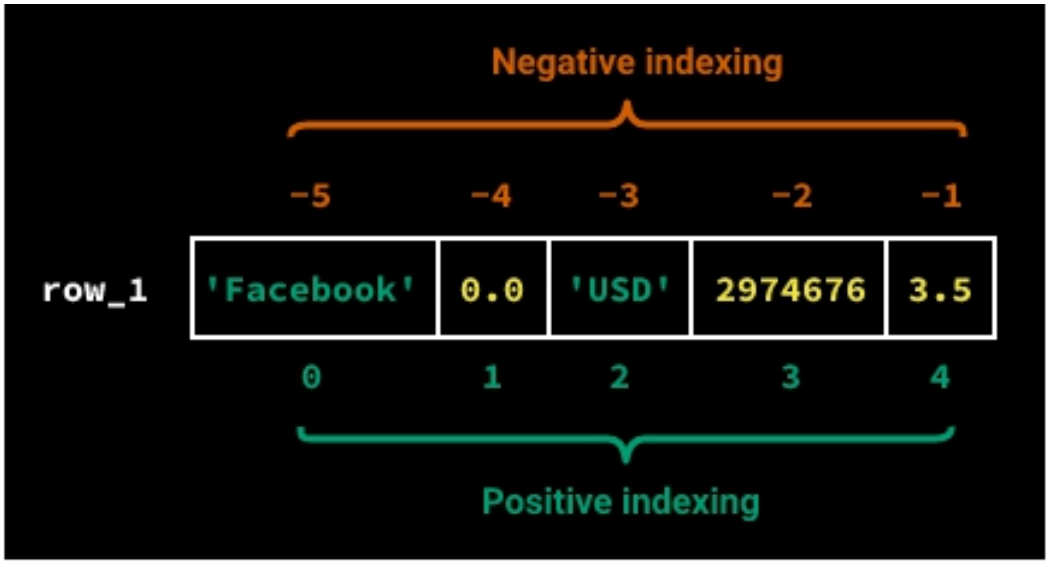
2422346.3333333335

Using Negative Indexing with Lists

There are two indexing systems for lists in Python:

1. Positive indexing: The index number of the first element is 0; the index number of the second element is 1 and furthermore.

2. Negative indexing: The index number of the last element is -1; the index number of the second element is -2 and furthermore.



In exercise, we mostly use positive indexing to obtain elements of the list. Negative indexing is helpful whenever we want to pick the last element in such a list, mostly if the list is lengthy, and by calculating, we cannot figure out the length.

script.py

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]

print(row_1[-1])
print(row_1[4])
```

```
Output
3.5
3.5
```

Note that when we use an index number just outside of the scope of the two indexing schemes, we are going to have an IndexError.

```
script.py
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
row_1[6]
```

```
Output
IndexError: list index out of range
```

```
script.py
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
row_1[-7]
```

```
Output
IndexError: list index out of range
```

How about using negative indexing to remove from each of the top 3 rows the user rating (the very last value) and afterwards average it.

row_1 [-1]=rating_1

row_2[-1]=rating_2

row_3[-1]=rating_3

rating_1 + rating_2 + rating_3=total_rating

total_rating / 3= average_rating

print(average)

2422346.33333

Slice Python Lists

Rather than selecting the list elements separately, we can pick two consecutive elements using a syntax shortcut:

script.py

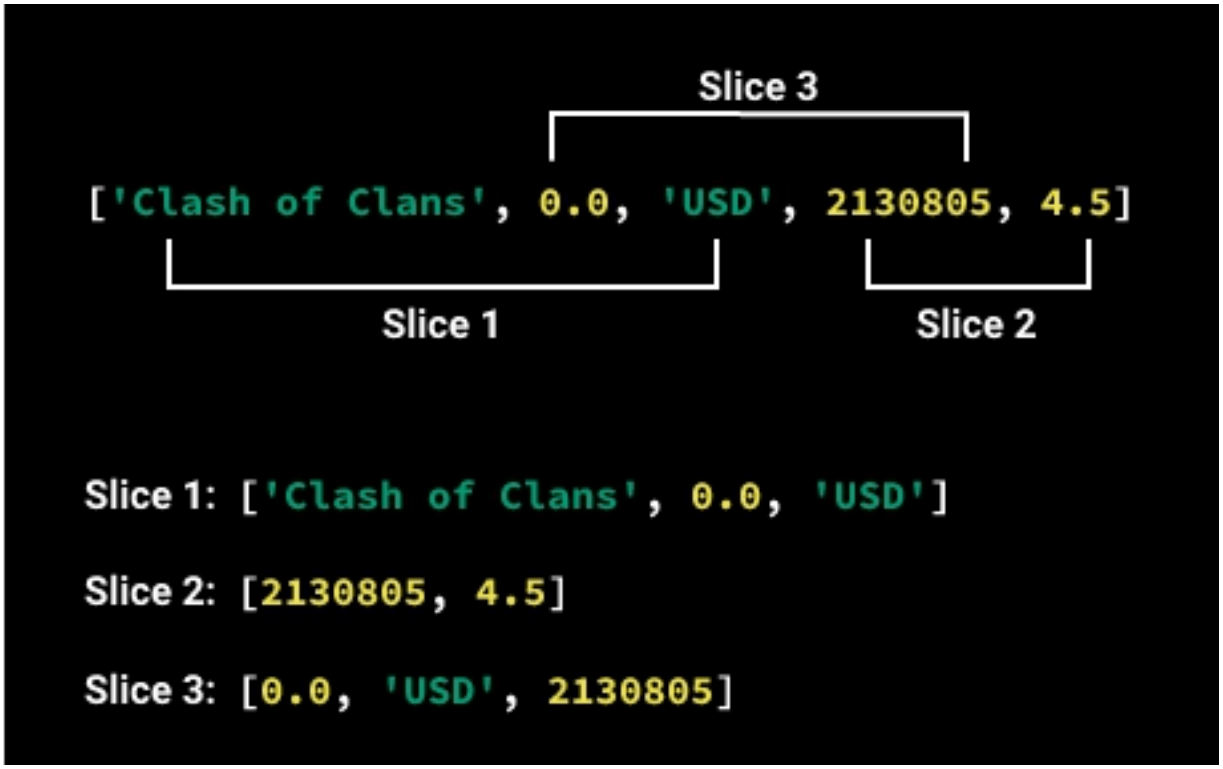
```
row_3 = ['Clash of Clans', 0.0, 'USD', 2130805, 4.5]

cc_pricing_data = row_3[0:3] ⟵ syntax shortcut
print(cc_pricing_data)
```

```
Output
['Clash of Clans', 0.0, 'USD']
```

While selecting the first n elements from a list called a list (n stands for a number), we can use the list syntax shortcut [0: n]. In the above example, we had to choose from the list row 3 the first three elements, so we will use row 3[0:3].

When the first three items were chosen, we sliced a portion of the set. For this function, the collection method for a section of a list is known as list slicing.

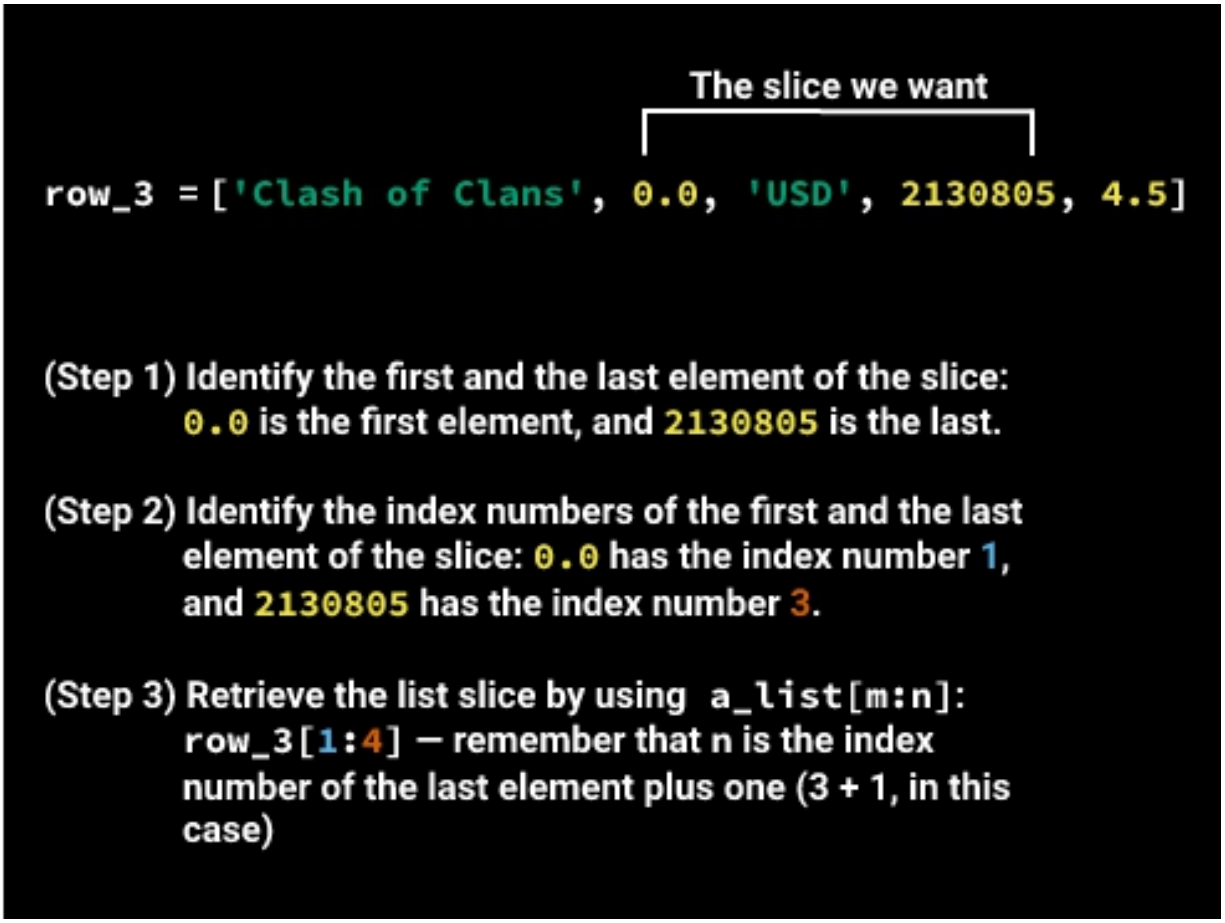
List slice can be done in many ways:



Retrieving any list slice we need:

- Firstly identify the first and last elements of the slice.
- The index numbers of the first and last element of the slice must then be defined.
- Lastly, we can use the syntax a list[m: n] to extract the list slice we require, while:

'm' means the index number of both the slice's 1st element; and 'n' symbolizes the index number of the slice's last element in addition to one (if the last element seems to have index number 2, after which n is 3, if the last element seems to have index number 4, after which n is 5, and so on).



When we want to choose the 1st or last ‘x’ elements (x represents a number), we may use even less complex shortcuts for syntax:

a_list[:x] when we need to choose the first x elements.

a_list[-x:] when we need to choose the last x elements.

script.py

```
row_3 = ['Clash of Clans', 0.0, 'USD', 2130805, 4.5]

first_3 = row_3[:3]
last_3 = row_3[-3:]

print(first_3)
print(last_3)
```

```
Output
['Clash of Clans', 0.0, 'USD']
['USD', 2130805, 4.5]
```

See how we retrieve from the first row the first four elements (with Facebook data):

first_4_fb = row_1[:4]

print(first_4_fb)

['Facebook', 0.0, 'USD', 2974676]

From the same row, the last three elements are:

last_3_fb = row_1[-3:]

print(last_3_fb)

['USD', 2974676, 3.5]

In the fifth row (data in the row for Pandora) with elements third and fourth are:

pandora_3_4 = row_5[2:4]

print(pandora_3_4)

['USD', 1126879]

Python List of Lists

Lists were previously introduced as a viable approach to using one variable per data point. Rather than having a different variable for any of the five 'Facebook' data points, 0.0, 'USD,' 2974676, 3.5, we can connect the data points into a list together and then save the list in a variable.

We have worked with a data set of five rows since then and have stored each row as a collection in each different variable (row 1, row 2, row 3, row 4, and row 5 variables). Even so, if we had a data set of 5,000 rows, we would probably have ended up with 5,000 variables that will create our code messy and nearly difficult to work with.

To fix this issue, we may store our five variables in a unified list:

script.py

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
row_2 = ['Instagram', 0.0, 'USD', 2161558, 4.5]
row_3 = ['Clash of Clans', 0.0, 'USD', 2130805, 4.5]
row_4 = ['Temple Run', 0.0, 'USD', 1724546, 4.5]
row_5 = ['Pandora - Music & Radio', 0.0, 'USD', 1126879, 4.0]

data_set = [row_1, row_2, row_3, row_4, row_5]
data_set
```

Output

Notice the double brackets

```
[['Facebook', 0.0, 'USD', 2974676, 3.5],
['Instagram', 0.0, 'USD', 2161558, 4.5],
['Clash of Clans', 0.0, 'USD', 2130805, 4.5],
['Temple Run', 0.0, 'USD', 1724546, 4.5],
['Pandora - Music & Radio', 0.0, 'USD', 1126879, 4.0]]
```

Notice the commas

As we're seeing, the data set is a list of five additional columns (row 1, row 2, row 3, row 4, and row 5). A list containing other lists is termed a set of lists.

The data set variable is already a list, which indicates that we can use the syntax we have learned to retrieve individual list elements and execute list slicing. Under, we have:

- Use datset[0] to locate the first list element (row 1).
- Use datset[-1] to locate the last list element (row 5).
- Obtain the first two list elements (row 1 and row 2) utilizing data set[:2] to execute a list slicing.

script.py

```
data_set = [row_1, row_2, row_3, row_4, row_5]
print(data_set[0])
print(data_set[-1])
print(data_set[:2])
```

Output

```
['Facebook', 0.0, 'USD', 2974676, 3.5]
['Pandora - Music & Radio', 0.0, 'USD', 1126879, 4.0]
[['Facebook', 0.0, 'USD', 2974676, 3.5],
['Instagram', 0.0, 'USD', 2161558, 4.5]]
```

Often, we will need to obtain individual elements from a list that is a portion of a list of lists — for example; we might need to obtain the rating of 3.5 from the data row ['FACEBOOK', 0.0, 'USD', 2974676, 3.5], which is a portion of the list of data sets. We retrieve 3.5 from data set below utilizing what we have learnt:

- Using data set[0], we locate row_1, and allocate the output to a variable named fb_row.
- fb_row ['Facebook', 0.0, 'USD', 2974676, 3.5] outputs, which we printed.
- Using fb_row[-1], we locate the final element from fb_row (because fb row is a list), and appoint the output to a variable called fb_rating.
- Print fb_rating, outputting 3.5

script.py

```
data_set = [row_1, row_2, row_3, row_4, row_5]
fb_row = data_set[0]
print(fb_row)

fb_rating = fb_row[-1]
print(fb_rating)
```

Output

```
['Facebook', 0.0, 'USD', 2974676, 3.5]
3.5
```

Earlier in this example, we obtained 3.5 in two steps: data_set[0] was first retrieved, and fb_row[-1] was then retrieved. There is also an easy way to get the same 3.5 output by attaching the two indices ([0] and [-1]); the code data_set[0][-1] gets 3.5.:

script.py

```
data_set = [row_1, row_2, row_3, row_4, row_5]
print(data_set[0][-1])
```


```
['Facebook', 0.0, 'USD', 2974676, 3.5]
```

Output

```
3.5
```

Earlier in this example, we have seen two ways to get the 3.5 value back. Both methods lead to the same performance (3.5), but the second approach requires fewer coding, as the steps we see from the example are elegantly integrated. As you can select an alternative, people generally prefer the latter.

Let's turn our five independent lists in to the list of lists:

app_data_set = [row_1, row_2, row_3, row_4, row_5]

then use:

print(app_data_set)

```
[
 ['FACEBOOK', 0.0, 'usd', 2974676, 3.5]

 ['INSTAGRAM', 0.0, 'usd', 2161558, 4.5

 ['CLASH OF CLANS', 0.0, 'usd', 2130805, 4.5]

 ['TEMPLE RUN', 0.0, 'usd', 1724546, 4.5]

['PANDORA', 0.0, 'usd', 1126879, 4.0]          ]
```

List Processes by Repetitive method

Earlier, we had an interest in measuring an app's average ranking in this project. It was a feasible task while we were attempting to work only for three rows, but the tougher it becomes, the further rows we add. Utilizing our tactic from the beginning, we will:

1. Obtain each individual rating.
2. Take the sum of the ratings.
3. Dividing by the total number of ratings.

script.py

```
row_1 = ['Facebook', 0.0, 'USD', 2974676, 3.5]
row_2 = ['Instagram', 0.0, 'USD', 2161558, 4.5]
row_3 = ['Clash of Clans', 0.0, 'USD', 2130805, 4.5]
row_4 = ['Temple Run', 0.0, 'USD', 1724546, 4.5]
row_5 = ['Pandora - Music & Radio', 0.0, 'USD', 1126879, 4.0]

app_data_set = [row_1, row_2, row_3, row_4, row_5]
avg_rating = (app_data_set[0][-1] + app_data_set[1][-1] +
              app_data_set[2][-1] + app_data_set[3][-1] +
              app_data_set[4][-1]) / 5

avg_rating
```

Output

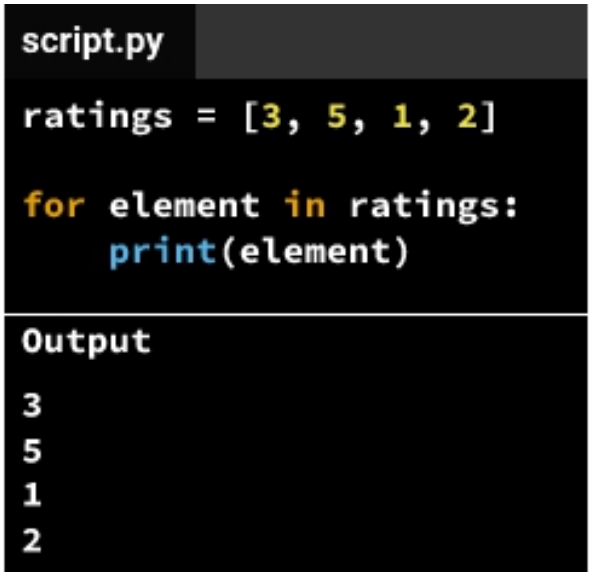
```
4.2
```

As you have seen that it becomes complicated with five ratings. Unless we were dealing with data that includes thousands of rows, an unimaginable amount of code would be needed! We ought to find a quick way to get lots of ratings back.

Taking a look at the code example earlier in this thread, we see that a procedure continues to reiterate: within app_data_set, we select the last list element for every list. What if we can just directly ask Python we would like to repeat this process in app_data_set for every list?

Luckily we can use it — Python gives us a simple route to repeat a plan that helps us tremendously when we have to reiterate a process tens of thousands or even millions of times.

Let’s assume we have a list [3, 5, 1, 2] allocated to a variable rating, and we need to replicate the following procedure: display the element for each element in the ratings. And this is how we can turn it into syntax with Python:



The procedure that we decided to replicate in our first example above was "generate the last item for each list in the app_data_set." Here's how we can transform that operation into syntax with Python:

script.py

```
app_data_set = [row_1, row_2, row_3, row_4, row_5]

for each_list in app_data_set:
    rating = each_list[-1]
    print(rating)
```

Output

```
3.5
4.5
4.5
4.5
4.0
```

Let's attempt and then get a good idea of what's going on above. Python differentiates each list item from app_data_set, each at a time, and assign it to each_list (which essentially becomes a vector that holds a list — we'll address this further):

script.py

```
app_data_set = [row_1, row_2, row_3, row_4, row_5]

for each_list in app_data_set:
    print(each_list)
```

Output

```
['Facebook', 0.0, 'USD', 2974676, 3.5]
['Instagram', 0.0, 'USD', 2161558, 4.5]
['Clash of Clans', 0.0, 'USD', 2130805, 4.5]
['Temple Run', 0.0, 'USD', 1724546, 4.5]
['Pandora - Music & Radio', 0.0, 'USD', 1126879, 4.0]
```

In the last figure earlier in this thread, the code is a much simpler and much more conceptual edition of the code below:

script.py

```
app_data_set = [row_1, row_2, row_3, row_4, row_5]

print(app_data_set[0])
print(app_data_set[1])
print(app_data_set[2])
print(app_data_set[3])
print(app_data_set[4])
```

```
for each_list in app_data_set:
    print(each_list)
```

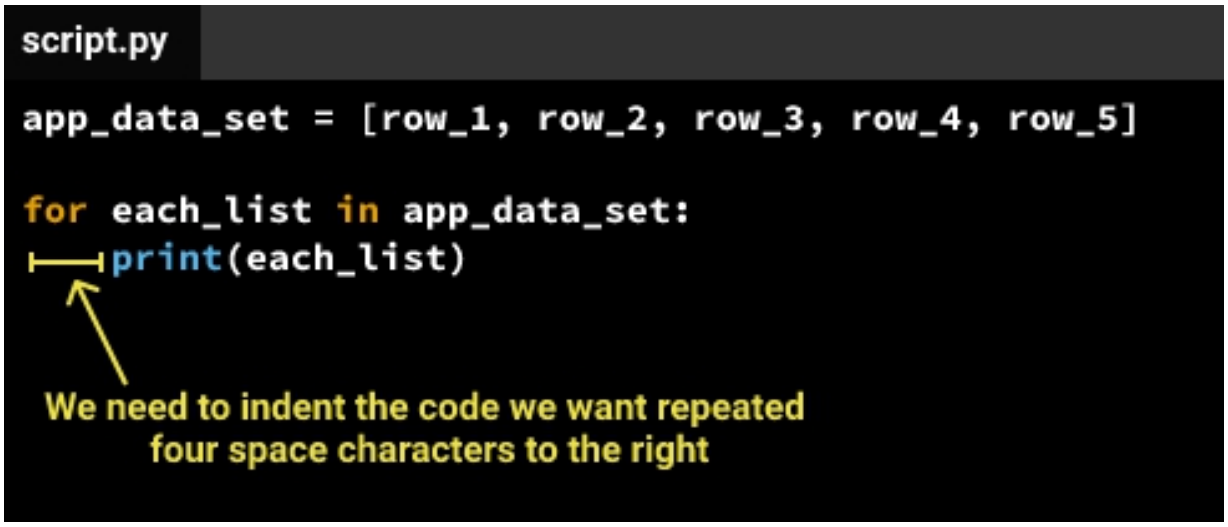
Output

```
['Facebook', 0.0, 'USD', 2974676, 3.5]
['Instagram', 0.0, 'USD', 2161558, 4.5]
['Clash of Clans', 0.0, 'USD', 2130805, 4.5]
['Temple Run', 0.0, 'USD', 1724546, 4.5]
['Pandora - Music & Radio', 0.0, 'USD', 1126879, 4.0]
```

Utilizing the above technique requires that we consider writing a line of code for each row in the data set. But by using the app_data_set methodology for each list involves that we write only two lines of code irrespective of the number of rows in the data set — the data set may have five rows or a hundred thousand.

Our transitional goal is to use this special method to calculate the average rating of our five rows above, in which our ultimate goal is to calculate the average rating of 7,197 rows for our data set. We 're going to get exactly that within the next few displays of this task, but we're going to concentrate for now on practicing this method to get a strong grasp of it.

We ought to indent the space characters four times to the right before we want to write the code:



Theoretically, we would only have to indent the code to the right with at least one space character, but in the Python language, the declaration is to use four space characters. This assists with readability — reading your code will be fairly easy for other individuals who watch this convention, and you will find it easier to follow theirs.

Now use this technique to print each app's name and rating:

foreach_list in app_data_set:

name = each_list[0]

rating = each_list[-1]

print(name, rating)

Facebook 3.5

Instagram 4.5

Clash of Clans 4.5

Temple Run 4.5

Pandora - Music & Radio 4.0

Loops

A loop is frequently used to iterate over a series of statements. We have two kinds of loops, ‘for loop’ and ‘while loop’ in Python. We will study ‘for loop’ and ‘while loop’ in the following scenario.

For Loop

Python's for loop is used to iterate over a sequence (list, tuple, string) or just about any iterate-able object. It is called traversal to iterate over a

sequence.

Syntax of For loop in Python

for<variable> in <sequence>:

body_of_loop that has set of statements

which requires repeated execution

In this case < variable > is often a variable used to iterate over a < sequence >. Around each iteration the next value is taken from < sequence > to approach the end of the sequence.

Python – For loop example

The example below illustrates the use of a loop to iterate over a list array. We calculate the square of each number present in the list and show the same with the body of for loop.

```
#Printing squares of all numbers program
# List of integer numbers
numbers = [1, 2, 4, 6, 11, 20]
#variable to store each number’s square temporary
sq = 0
#iterating over the given list
forval in numbers:
    # calculating square of each number
sq = val * val
    # displaying the squares
print(sq)
```

Output:

1

4

16

36

121

400

For loop with else block

Excluding Java, we can have the loop linked with an optional 'else' block in Python. The 'else' block only runs after all the iterations are finished by the loop. Let's see one example:

```
For val in range(5):
    print(val)
else:
    print("The loop has completed execution")
```

Output:

```
0
1
2
3
4
The loop has completed execution
```

Note: else block is executed when the loop is completed.

Nested For loop in Python

If there is a loop within another for loop, then it will be termed a nested for loop. Let's take a nested for loop example.

```
for num1 in range(3):
    for num2 in range(10, 14):
        print(num1, ",", num2)
```

Output:

```
0 , 10
0 , 11
0 , 12
0 , 13
1 , 10
1 , 11
1 , 12
1 , 13
2 , 10
```

2 , 11

2 , 12

2 , 13

While Loop

While loop is also used to continuously iterate over a block of code until a specified statement returns false, we have seen in many for loop in Python in the last guide, which is used for a similar intent. The biggest distinction is that we use for looping when we are not sure how many times the loop needs execution, yet on the other side when we realize exactly how many times we have to execute the loop, we need for a loop.

Syntax of while loop

while conditioning:

    #body_of_while

The body of the while is a series of statements from Python which require repetitive implementation. These claims are consistently executed until the specified condition returns false.

while loop flow

1. Firstly given condition is inspected, the loop is canceled if the condition returns false, and also the control moves towards the next statement in the compiler after the loop.

2. When the condition returns true, the set of statements within the loop will be performed, and the power will then switch to the loop start for the next execution.

Those two measures continuously occur as long as the condition defined in the loop stands true.

While loop example

This is an example of a while loop. We have a variable number in this case, and we show the value of the number in a loop, the loop will have an incremental operation where we increase the number value. It is a very crucial component, while the loop should have an operation of increase or decrease. Otherwise, the loop will operate indefinitely.

num = 1

#loop will repeat itself as long as it can

```
#num< 10 remains true
whilenum< 10:`
print(num)
    #incrementing the value of num
num = num + 3
```

Output:

1

4

7

Infinite while loop

Example 1:

This will endlessly print the word 'hello' since this situation will always be true.

```
while True:
print("hello")
```

Example 2:

```
num = 1
whilenum<5:
print(num)
```

This will endlessly print '1' since we do not update the number value inside the loop, so the number value would always remain one, and the condition number<5 would always give back true.

Nested while loop in Python

While inside another while loop a while loop is present, then it will be considered nested while loop. To understand this concept, let us take an example.

```
i = 1
j = 5
while i< 4:
while j < 8:
print(i, ",", j)
```

```
        j = j + 1
i = i + 1
```

Output:

1 , 5

2 , 6

3 , 7

Python – while loop with else block

We may add an 'else' block to a while loop. The section 'else' is possible. It executes only when the processing of the loop has ended.

```
num = 10
whilenum> 6:
print(num)
num = num-1
else:
print("loop is finished")
```

Output:

10

9

8

7

Loop is finished

ADDING MULTIPLE VALUED DATA IN PYTHON

Often the creator wants users to input multiple values or inputs in a line. In Python, users could use two techniques to take multiple values or inputs in one line.

1. Use of split() method
2. Use of List comprehension

Use of split() method :

This feature helps to receive many user inputs. It splits the defined separator to the given input. If no separator is given, then a separator is blank space. Users generally use a split() method to separate a Python string, but it can be used when multiple inputs are taken.

Syntax:

input().split(separator, maxsplit)

Example:

```
filter_none
edit
play_arrow
brightness_4
#Python program showing how to add
#multiple input using split
#taking two inputs each time
x, y = input("Enter a two value: ").split()
print("Number of boys: ", x)
print("Number of girls: ", y)
print()
# taking three inputs at a time
x, y, z = input("Enter a three value: ").split()
print("Total number of students: ", x)
print("Number of boys is : ", y)
print("Number of girls is : ", z)
print()
# taking two inputs at a time
a, b = input("Enter a two value: ").split()
print("First number is {} and second number is {}".format(a, b))
print()
# taking multiple inputs at a time
# and type casting using list() function
x = list(map(int, input("Enter a multiple value: ").split()))
print("List of students: ", x)
```

Output:

```
Enter a two value: 5 10
Number of boys:  5
Number of girls:  10

Enter a three value: 30 10 20
Total number of students:  30
Number of boys is :  10
Number of girls is :  20

Enter a four value: 20 30
First number is 20 and second number is 30

Enter a multiple value: 20 30 10 22 23 26
List of students:  [20, 30, 10, 22, 23, 26]
```

Using List comprehension:

Comprehension of lists is an easy way of describing and building a list in Python. Just like mathematical statements, we can generate lists within each line only. It is often used when collecting multiple device inputs.

Example:

```
filter_none
edit
play_arrow
brightness_4
# Python program showing
# how to take multiple input
# using List comprehension
# taking two input at a time
x, y = [int(x) for x in input("Enter two value: ").split()]
print("First Number is: ", x)
print("Second Number is: ", y)
print()
# taking three input at a time
x, y, z = [int(x) for x in input("Enter three value: ").split()]
print("First Number is: ", x)
print("Second Number is: ", y)
print("Third Number is: ", z)
print()
# taking two inputs at a time
x, y = [int(x) for x in input("Enter two value: ").split()]
print("First number is {} and second number is {}".format(x, y))
print()
# taking multiple inputs at a time
x = [int(x) for x in input("Enter multiple value: ").split()]
print("Number of list is: ", x)
```

Output:

```
Enter two value: 2 5
First Number is:  2
Second Number is:  5

Enter three value: 2 4 5
First Number is:  2
Second Number is:  4
Third Number is:  5

Enter two value: 2 10
First number is 2 and second number is 10

Enter multiple value: 1 2 3 4 5
Number of list is:  [1, 2, 3, 4, 5]
```

Note: The definitions above take inputs divided by spaces. If we prefer to pursue different input by comma (“,”), we can just use the below:

```
# taking multiple inputs divided by comma at a time
x = [int(x) for x in input("Enter multiple value: ").split(",")]
print("Number of list is: ", x)
```

Assign multiple values to multiple variables

By separating the variables and values with commas, you can allocate multiple values to different variables.

```
a, b = 100, 200
print(a)
# 100
print(b)
# 200
```

You have more than three variables to delegate. In addition, various types can be assigned, as well.

```
a, b, c = 0.1, 100, 'string'
print(a)
# 0.1
print(b)
```

```
# 100
print(c)
#string
```

Assign the same value to multiple variables

Using = consecutively, you could even appoint multiple variables with the same value. For instance, this is helpful when you initialize multiple variables to almost the same value.

```
a = b = 100
print(a)
# 100
print(b)
# 100
```

Upon defining the same value, another value may also be converted into one. As explained later, when allocating mutable objects such as lists or dictionaries, care should be taken.

```
a = 200
print(a)
# 200
print(b)
# 100
```

It can be written three or more in the same way.

```
a = b = c = 'string'
print(a)
# string
print(b)
# string
print(c)
# string
```

Instead of immutable objects like int, float, and str, be careful when appointing mutable objects like list and dict.

When you use = consecutively, all variables are assigned the same object, so if you modify the element value or create a new element, then the other object will also modify.

```
a = b = [0, 1, 2]
print(a is b)
# True
a[0] = 100
print(a)
# [100, 1, 2]
print(b)
# [100, 1, 2]
```

Same as below.

```
b = [0, 1, 2]
a = b
print(a is b)
# True
a[0] = 100
print(a)
# [100, 1, 2]
print(b)
# [100, 1, 2]
```

If you would like to independently manage them you need to allocate them separately.

after c = []; d = [], c and d are guaranteed to link to two unique, newly created empty,different lists. (Note that c = d = [] assigns the same object to both c and d.)

Here is another example:

```
a = [0, 1, 2]
b = [0, 1, 2]
print(a is b)
# False
```

```
a[0] = 100
print(a)
# [100, 1, 2]
print(b)
# [0, 1, 2]
```

ADDING STRING DATA IN PYTHON

What is String in Python?

A string is a Character set. A character is just a symbol. The English language, for instance, has 26 characters. Operating systems do not handle characters they handle the (binary) numbers. And if you may see characters on your computer, it is represented internally as a mixture of 0s and 1s and is manipulated. The transformation of character to a number is known as encoding, and probably decoding is the reverse process. ASCII and Unicode are two of the widely used encodings. A string in Python is a series of characters in Unicode. Unicode was incorporated to provide all characters in all languages and to carry encoding uniformity. Python Unicode allows you to learn regarding Unicode.

How to create a string in Python?

Strings may be formed by encapsulating characters or even double quotes inside a single quotation. In Python, even triple quotes may be used but commonly used to portray multiline strings and docstrings.

```
# defining strings in Python

# all of the following are equivalent

my_string = 'Hello'

print(my_string)

my_string = "Hello"

print(my_string)

my_string = '''Hello'''

print(my_string)

# triple quotes string can extend multiple lines

my_string = """Hello, welcome to the world of Python"""

print(my_string)
```

When the program is executed, the output becomes:

```
Hello

Hello

Hello

Hello, welcome to the world of Python
```

Accessing the characters in a string?

By indexing and using slicing, we can obtain individual characters and scope of characters. The index commences at 0. Attempting to obtain a character from index range will cause an IndexError to increase. The index

has to be integral. We cannot use floats or other types, and this will lead to TypeError. Python lets its sequences be indexed negatively. The -1 index corresponds to the last object, -2 to the second object, and so forth. Using the slicing operator ‘(colon),’ we can access a range of items within a string.

#Python string characters access:

```
str = 'programiz'

print('str = ', str)

#first character

print('str[0] = ', str[0])

#last character

print('str[-1] = ', str[-1])

#slicing 2nd to 5th character

print('str[1:5] = ', str[1:5])

#slicing 6th to 2nd last character

print('str[5:-2] = ', str[5:-2])
```

If we execute the code above we have the following results:

```
str =  programiz

str[0] =  p

str[-1] =  z

str[1:5] =  rogr

str[5:-2] =  am
```

When we attempt to access an index out of the range, or if we are using numbers other than an integer, errors will arise.

index must be in the range

```
>>>my_string[15]

...

IndexError: string index out of range

# index must be an integer

>>>my_string[1.5]

...
```

TypeError: Define string indices as integers only

By analyzing the index between the elements as seen below, slicing can best be visualized. Whenever we want to obtain a range, we need the index that slices the part of the string from it.

How to change or delete a string?

Strings are unchangeable. This means elements of a list cannot be modified until allocated. We will easily reassign various strings of the same term.

```
>>>my_string = 'programiz'

>>>my_string[5] = 'a'

...

TypeError: 'str' object does not support item assignment

>>>my_string = 'Python'

>>>my_string

'Python'
```

We cannot erase characters from a string, or remove them. But it's easy to erase the string completely by using del keyword.

```
>>>delmy_string[1]

...

TypeError: 'str' object doesn't support item deletion

>>>delmy_string

>>>my_string

...

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

Python String Operations

There are many methods that can be used with string making it one of the most commonly used Python data types. See Python Data Types for more information on the types of data used in Python coding

Concatenation of Two or More Strings

The combination of two or even more strings into one is termed concatenation. In Python, the + operator does that. They are likewise concatenated by actually typing two string literals together. For a specified number of times, the * operator could be used to reiterate the string.

```
# Python String Operations

str1 = 'Hello'

str2 ='World!'

# using +

print('str1 + str2 = ', str1 + str2)

# using *

print('str1 * 3 =', str1 * 3)

Once we execute the program above we get the following results:

str1 + str2 =  HelloWorld!

str1 * 3 = HelloHelloHello

Using two literal strings together would therefore concatenate them like + operator.

We might use parentheses if we wish to concatenate strings in various lines.
>>> # two string literals together

>>> 'Hello ''World!'

'Hello World!'

>>> # using parentheses

>>> s = ('Hello '

...      'World')

>>>s

'Hello World'
```

Iterating Through a string

With a for loop, we can iterate through a string. This is an example of counting the number of 'l's in a string function.

```
#Iterating through a string

count = 0

for letter in 'Hello World':

if(letter == 'l'):

count += 1

print(count,'letters found')
```

If we execute the code above, we have the following results:

‘3 letters found.’

String Membership Test

We can check whether or not there is a substring within a string by using keyword in.

>>> 'a' in 'program'

True

>>> 'at' not in 'battle'

False

Built-in functions to Work with Python

Different built-in functions which can also be work with strings in series. A few other commonly used types are len() and enumerate(). The function enumerate() returns an enumerate object. It includes the index and value as combinations of all elements in the string. This may be of use to iteration. Comparably, len() returns the string length (characters number).

```
str = 'cold'

# enumerate()

list_enumerate = list(enumerate(str))

print('list(enumerate(str) = ', list_enumerate)

#character count

print('len(str) = ', len(str))

Once we execute the code above we have the following results:

list(enumerate(str) =  [(0, 'c'), (1, 'o'), (2, 'l'), (3, 'd')]

len(str) =  4
```

Formats for Python String

Sequence for escaping

We can't use single quotes or double quotes if we want to print a text like He said, "What's there?" This would result in a SyntaxError because there are single and double quotations in the text alone.

>>>print("He said, "What's there?"")

...

SyntaxError: invalid syntax

>>>print('He said, "What's there?"')

...

SyntaxError: invalid syntax

Triple quotes are one way to get round the problem. We might use escape sequences as a solution. A series of escape starts with a backslash, which is represented differently. If we are using a single quote to describe a string, it is important to escape all single quotes within the string. The case with double quotes is closely related. This is how the above text can be represented.

```
# using triple quotes

print('''He said, "What's there?"''')

# escaping single quotes

print('He said, "What\'s there?"')

# escaping double quotes

print("He said, \"What's there?\"")
```

Once we execute the code above, we have the following results:

He said, "What's there?"

He said, "What's there?"

He said, "What's there?"

Raw String to ignore escape sequence

Quite often inside a string, we might want to reject the escape sequences. To use it, we can set r or R before the string. Which means it's a raw string, and it will neglect any escape sequence inside.

>>>print("This is \x61 \ngood example")

This is a

good example

>>> print(r"This is \x61 \ngood example")

This is \x61 \ngood example

The format() Method for Formatting Strings

The format() sources available and make with the string object is very flexible and potent in string formatting. Style strings contain curly braces{} as placeholders or fields of substitution, which are substituted.

To specify the sequence, we may use positional arguments or keyword arguments.

```
# Python string format() method
# default(implicit) order
default_order = "{}, {} and {}".format('John','Bill','Sean')
print('\n--- Default Order ---')
print(default_order)
# order using positional argument
positional_order = "{1}, {0} and {2}".format('John','Bill','Sean')
print('\n--- Positional Order ---')
print(positional_order)
# order using keyword argument
keyword_order = "{s}, {b} and {j}".format(j='John',b='Bill',s='Sean')
print('\n--- Keyword Order ---')
print(keyword_order)
Once we execute the code above we have the following results:
--- Default Order ---
John, Bill and Sean
--- Positional Order ---
Bill, John and Sean
--- Keyword Order ---
Sean, Bill and John
```

The format() technique can have requirements in optional format. Using colon, they are divided from the name of the field. For example, a string in the given space may be left-justified <, right-justified >, or based ^.

Even we can format integers as binary, hexadecimal, etc. and floats can be rounded or shown in the style of the exponent. You can use tons of compiling there. For all string formatting available using the format() method, see below example:

```
>>> # formatting integers

>>> "Binary representation of {0} is {0:b}".format(12)

'Binary representation of 12 is 1100'

>>> # formatting floats

>>> "Exponent representation: {0:e}".format(1566.345)

'Exponent representation: 1.566345e+03'

>>> # round off

>>> "One third is: {0:.3f}".format(1/3)

'One third is: 0.333'

>>> # string alignment

>>> "|{:<10}|{:^10}|{:>10}|".format('butter','bread','ham')

'|butter    |  bread   |       ham|'
```

Old style formatting

We even can code strings such as the old sprint() style in the programming language used in C. To accomplish this; we use the '%' operator.

```
>>> x = 12.3456789

>>>print('The value of x is %3.2f' %x)

The value of x is 12.35

>>>print('The value of x is %3.4f' %x)

The value of x is 12.3457
```

String common Methods for Python

The string object comes with various methods. One of them is the format() method we described above. A few other frequently used technique include lower(), upper(), join(), split(), find(), substitute() etc. Here is a wide-range list of several of the built-in methodologies in Python for working with strings.

```
>>> "PrOgRaMiZ".lower()

'programiz'

>>> "PrOgRaMiZ".upper()

'PROGRAMIZ'

>>> "This will split all words into a list".split()

['This', 'will', 'split', 'all', 'words', 'into', 'a', 'list']

>>> ' '.join(['This', 'will', 'join', 'all', 'words', 'into', 'a', 'string'])

'This will join all words into a string'

>>> 'Happy New Year'.find('ew')

7

>>> 'Happy New Year'.replace('Happy','Brilliant')

'Brilliant New Year'
```

Inserting values into strings

Method 1 - the string format method

The string method format method can be used to create new strings with the values inserted. That method works for all of Python's recent releases. That is where we put a string in another string:

```
>>>shepherd = "Mary"

>>>string_in_string = "Shepherd {} is on duty.".format(shepherd)

>>>print(string_in_string)
```

Shepherd Mary is on duty.

The curved braces indicate where the inserted value will be going.

You can insert a value greater than one. The values should not have to be strings; numbers and other Python entities may be strings.

```
>>>shepherd = "Mary"

>>>age = 32

>>>stuff_in_string = "Shepherd {} is {} years old.".format(shepherd, age)

>>>print(stuff_in_string)

Shepherd Mary is 32 years old.

>>> 'Here is a {} floating point number'.format(3.33333)

'Here is a 3.33333 floating point number'
```

Using the formatting options within curly brackets, you can do more complex formatting of numbers and strings — see the information on curly brace string layout.

This process allows us to give instructions for formatting things such as numbers, using either: inside the curly braces, led by guidance for formatting. Here we request you to print in integer (d) in which the number is 0 to cover the field size of 3:

```
>>>print("Number {:03d} is here.".format(11))

Number 011 is here.

This prints a floating point value (f) with exactly 4 digits after the decimal point:

>>> 'A formatted number - {:.4f}'.format(.2)

'A formatted number - 0.2000'
```

Method 2 - f-strings in Python >= 3.6

When you can rely on having Python > = version 3.6, you will have another appealing place to use the new literal (f-string) formatted string to input variable values. Just at the start of the string, an f informs Python to permit any presently valid variable names inside the string as column names. So here's an example such as the one above, for instance using the f-string syntax:

```
>>>shepherd = "Martha"

>>>age = 34

>>> # Note f before first quote of string

>>>stuff_in_string = f"Shepherd {shepherd} is {age} years old."

>>>print(stuff_in_string)
```

Shepherd Martha is 34 years old.

Method 3 - old school % formatting

There seems to be an older string formatting tool, which uses the percent operator. It is a touch less versatile than the other two choices, but you can still see it in use in older coding, where it is more straightforward to use ‘%’ formatting. For formatting the ‘%’ operator, you demonstrate where the encoded values should go using a ‘%’ character preceded by a format identifier to tell how to add the value.

So here's the example earlier in this thread, using formatting by ‘%.’ Note that ‘%s’ marker for a string to be inserted, and the ‘%d’ marker for an integer.

```
>>>stuff_in_string = "Shepherd %s is %d years old." % (shepherd, age)

>>>print(stuff_in_string)
```

Shepherd Martha is 34 years old.

MODULE DATA

What are the modules in Python?

Whenever you leave and re-enter the Python interpreter, the definitions you have created (functions and variables) will get lost. Consequently, if you'd like to develop a code a little longer, it's better to use a text editor to plan the input for the interpreter and execute it with that file as input conversely. This is defined as script formation. As the software gets bigger, you may want to break it into different files to make maintenance simpler. You might also like to use a handy function that you wrote in many other programs without having to replicate its definition inside each program. To assist this, Python has the option of putting definitions into a file and using them in the interpreter's code or interactive instances. This very file is considered a module; module descriptions can be loaded into certain modules or into the main module (the list of variables you have exposure to in a high-level script and in converter mode).

A module is a file that contains definitions and statements from Python. The name of the file is the name of the module with the .py suffix attached. The name of the module (only as string) inside a module is available as the value, including its global variable __name__. For example, use your preferred text editor to build a file named fibo.py with the following contents in the current working directory:

```
# Python Module example

def add(a, b):

   """This program adds two

numbers and return the result"""

result = a + b

return result
```

In this, we defined an add() function within an example titled “ module.” The function requires two numbers and returns a total of them.

How to import modules in Python?

Within a module, we can import the definitions to some other module or even to the interactive Python interpreter. To do something like this, we use the keyword import. To load our recently specified example module, please enter in the Python prompt.

>>> import example

This should not import the identities of the functions directly in the existing symbol table, as defined in the example. It just imports an example of the module name there.

Using the name of the module, we can use the dot(.) operator to access the function. For instance:

>>>example.add(4,5.5)

9.5

Python comes with lots of regular modules. Check out the complete list of regular Python modules and their usage scenarios. These directories are within the destination where you've installed Python in the Lib directory. Normal modules could be imported just the same as our user-defined modules are imported.

There are different ways of importing the modules. You'll find them below:

Python import statement

Using the import statement, we can extract a module by using the dot operator, as explained in the previous section and access the definitions within it. Here is another example.

```
# import statement example

# to import standard module math

import math

print("The value of pi is", math.pi)
```

Once we execute the code above, we have the following results:

The value of pi is 3.141592653589793

Import with renaming

We can load a module in the following way by changing the name of it:

```
# import module by renaming it

import math as m

print("The value of pi is", m.pi)
```

We called the module Math as m. In certain instances, this will save us time to type. Remember that in our scope, the name math is not identified. Therefore math.pi is incorrect, and m.pi is correctly implemented.

Python from...import statement

We can import individual names from such a module without having to import the entire module. Here is another example.

```
# import only pi from math module

from math import pi

print("The value of pi is", pi)
```

In this, only the pi parameter was imported from the math module. We don't utilize the dot operator in certain cases. We can likewise import different modules:

>>>from math import pi, e

>>>pi

3.141592653589793

>>>e

2.718281828459045

Import all names

With the following form, we can import all terms (definitions) from a module:

import all names from standard module math

from math import *

print("The value of pi is," pi)

Above, we have added all of the math module descriptions. This covers all names that are available in our scope except those that start with an underscore. It is not a good programming technique to import something with the asterisk (*) key. This will lead to a replication of an attribute's meaning. This also restricts our code's readability.

Python Module Search Path

Python looks at many locations when importing a module. Interpreter searches for a built-in module instead. So if not included in the built-in module, Python searches at a collection of directories specified in sys.path. The exploration is in this sequence:

PYTHONPATH (list of directories environment variable)

The installation-dependent default directory

```
>>> import sys

>>>sys.path

['',

'C:\\Python33\\Lib\\idlelib',

'C:\\Windows\\system32\\python33.zip',

'C:\\Python33\\DLLs',

'C:\\Python33\\lib',

'C:\\Python33',

'C:\\Python33\\lib\\site-packages']
```

We can insert that list and customize it to insert our own location.

Reloading a module

During a session, the Python interpreter needs to import one module only once. This makes matters more productive. Here is an example showing how that operates.

Assume we get the code below in a module called my_module:

```
# This module shows the effect of

#  multiple imports and reload

print("This code got executed")
```

Now we suspect that multiple imports have an impact.

>>> import my_module

This code was executed:

>>> import my_module

>>> import my_module

We have seen our code was only executed once. This means that our module has only been imported once.

Also, if during the process of the test our module modified, we will have to restart it. The way to do so is to reload the interpreter. But that doesn't massively help. Python offers an effective way to do so. Within the imp module, we may use the reload() function to restart a module. Here are some ways to do it:

```
>>> import imp
>>> import my_module
```

This code executes

```
>>> import my_module
>>>imp.reload(my_module)
```

This code executes

```
<module 'my_module' from '.\\my_module.py'>
```

The dir() built-in function

We may use the dir() function to locate names specified within a module. For such cases, in the example of the module that we had in the early part, we described a function add().

In example module, we can use dir in the following scenario:

```
>>>dir(example)

['__builtins__',

'__cached__',

'__doc__',

'__file__',

'__initializing__',

'__loader__',

'__name__',

'__package__',

'add']
```

Now we'll see a list of the names sorted (alongside add). Many other names that start with an underscore are module-associated (not user-defined) default Python attributes. For instance, the attribute name contains module __name__.

>>> import example

>>>example.__name__

'example'

You can find out all names identified in our existing namespace by using dir() function with no arguments.

```
>>> a = 1

>>> b = "hello"

>>> import math

>>>dir()

['__name__', '__doc__','__builtins__ ', 'a', 'b', 'math', 'pyscripter']
```

Executing modules as scripts

Python module running with python fibo.py <arguments>the program will be running in such a way, just like it was being imported, but including the __name__ set to "__main__." That implies this program is inserted at the end of the module:

If __name__ == "__main__": import sys fib(int(sys.argv[1]))

You could even create the file usable both as a script and as an importable module since this code parsing the command - line interface runs only when the module is performed as the "main" file:

$ python fibo.py 50

0 1 1 2 3 5 8 13

When the module is imported, the code will not be executed:

>>>

>>> import fibo

>>>

It is most often used whether to get an efficient user interface to a module or for test purposes (the module runs a test suite as a script).

“Compiled” Python files

To speed up loading modules, Python caches the compiled version of each module in the __pycache__ directory with the name module.version.pyc, in which the version encapsulates the assembled file format; it normally includes the firmware version of Python. For instance, the compiled edition of spam.py in CPython launch 3.3 will be cached as __pycache__/spam.cpython-33.pyc. This naming convention enables the coexistence of compiled modules from various updates and separate versions of Python.

Python tests the source change schedule against the compiled edition to see if it is out-of-date and needs recompilation. That's a fully automated system. Even the assembled modules become platform-independent, so different algorithms will use the same library between systems. In two situations Pythoniswill not check the cache:

- ➢ First, it often recompiles the output for the module, which is loaded explicitly from the command line but does not store it.
- ➢ Second, when there is no root module, it will not search the cache. The compiled module must be in the source directory to facilitate a non-source (compiled only) release, and a source module should not be installed.

Some tips for users:

- To minimize the size of a compiled file, you can use the -O or -OO switches in the Python order. The -O switch erases statements of assert, the -OO switch removes statements of assert as well as strings of doc. Although some codes may support getting these options available, this method should only be used if you are aware of what you are doing. "Optimized" modules usually have such an opt-tag and are tinier. Future releases may modify the optimal control implications.
- A project run no faster once it is read from a.pyc file than how it was read from a.py file; just one thing about.pyc files that are faster in the speed with which they will be loaded.
- A compile all modules can generate .pyc files in a directory for all of the other modules.
- More details on this process are given in PEP 3147, along with a flow chart of the decision making.

Standard Modules

Python has a standard modules library, mentioned in a separate section, the Python Library allusion (hereafter "Library Reference"). A few modules are incorporated into the interpreter; that provide direct exposure to processes that are not component of the language's base but are nonetheless built-in, whether for effectiveness or to supply access to primitive operating systems such as source code calls. The collection of these modules is an

alternative to customize and also relies on the framework underlying it. The winreg module, for instance, is only available on Microsoft windows. One particular module is worthy of certain interest: sys, which is integrated into every Python interpreter. The sys.ps1 and sys.ps2 variables classify strings which are used as primary and secondary instructions:

```
>>>
>>> import sys
>>> sys.ps1
'>>> '
>>> sys.ps2
'... '
>>> sys.ps1 = 'C> '
C>print('Yuck!')
Yuck!
C>
```

Only when the interpreter is in interactive mode are those two variables defined. The sys.path variable is a collection of strings that defines the search path for modules used by the interpreter. When PYTHONPATH is not a part of the set, then it will be defined to a predefined path taken from either the PYTHONPATH environment variable or through a built-in default. You can change it with regular list procedures:

```
>>>
>>> import sys
>>>sys.path.append('/python/ufs/guido/lib/')
```

Packages

Packages are indeed a way to construct the namespace of the Python module by using "pointed names of the module." For instance, in a package called A., the module title A.B specifies a submodule named B. Even as the use of modules prevents the writers of various modules from stopping to know about the global variable names of one another, any use of dotted module names prevents the developers of multi-module bundles like NumPy or Pillow from needing to worry more about module names of one

another. Consider making a series of lists of modules (a "package") to handle sound files and sound data in an even manner.

There are several various programs of sound files usually familiar with their extension, for example: ‘wav,.aiff,.au,’ though you'll need to build and maintain a massive collection of modules to convert between some of the multiple formats of files. There are several other different operations that you may like to run on sound data (such as blending, adding echo, implementing an equalizer function, producing an optical stereo effect), and you'll just be writing an infinite series of modules to execute those interventions. Here is another feasible package layout (described in terms of a hierarchical file system):

```
sound/                          Top level package
      __init__.py               sound package initialization
formats/                  Subpackage for conversions of file format
              __init__.py
              wavread.py
              wavwrite.py
              aiffread.py
              aiffwrite.py
              auread.py
              auwrite.py
              ...
effects/                  Sound effectssubpackage
              __init__.py
              echo.py
              surround.py
              reverse.py
              ...
filters/                  Filterssubpackage
              __init__.py
              equalizer.py
              vocoder.py
              karaoke.py
              ...
```

While loading the bundle, Python checks for the packet subdirectory via the folders on sys.path. To allow Python view directories that hold the file as packages, the __init__.py files are needed. This protects directories with a common name, including string, from accidentally hiding valid modules, which later appear mostly on the search path of the module. In the correct

order; __init__.py can only be a blank file, but it could also implement the package preprocessing code or establish the variable __all__ described below

- Package users could even upload individual modules from the package, such as: ‘import sound.effects.echo’
- This loads the ‘sound.effects.echo’ sub-module. Its full name must be mentioned: ‘sound.effects.echo.echofilter(input, output, atten=4, delay=0.7)’
- Another way to import the submodule is: ‘fromsound.effects import echo’
- It, therefore, launches the sub-module echo and provides access but without package prefix: ‘echo.echofilter(input, output, atten=4, delay=0.7)’
- And just another option is to explicitly import the desired function or attribute: ‘fromsound.effects.echo import echofilter’
- This again activates the echo sub-module however this enables its echofilter() feature explicitly accessible: ‘echofilter(input, output, delay=0.7, atten=4)’

So it heaps the sub-module echo; however this tends to make its function; remember that the object will either be a sub-module (or sub-package)of the package or any other name described in the package, such as a function, class or variable while using from package import object. Initially, the import statement analyses if the object is characterized in the package; otherwise, it supposes that it is a module and makes an attempt to load it. Once it fails to reach it, an exception to ‘ImportError’ will be promoted.

Referring to this, while using syntax such as import ‘item.subitem.subsubitem’, each item has to be a package, but the last one; the last item could be a module or package, but this cannot be a class or function or variable identified in the previous item.

CONCLUSION

Research across almost all fields has become more data-oriented, impacting both the job opportunities and the required skills. While more data and methods of evaluating them are becoming obtainable, more data-dependent aspects of the economy, society, and daily life are becoming. Whenever it comes to data science, Python is a tool necessary with all sorts of advantages. It is flexible and continually improving because it is open-source. Python already has a number of valuable libraries, and it cannot be ignored that it can be combined with other languages (like Java) and current frameworks. Long story short -Python is an amazing method for data science.

PYTHON
CRASH COURSE

Beginner guide to computer programming, web coding and data mining. Learn in 7 days machine learning, artificial intelligence, NumPy and Pandas packages with exercises for data analysis.

JASON TEST

DAY 1

What Is Python?

Python is a high-level, object-oriented, construed programming language with complex semblance. Combined with dynamic typing and dynamic binding, its high-level data structures make it very attractive for Rapid Application Development as well as for use as a scripting or glue language for connecting existing components. Python's quick, easy to understand syntax, stresses readability, and hence reduces the expense of running the software. Python connects modules and packages that promote the modularity of the software and reuse of code. For all major platforms, the Python interpreter and the comprehensive standard library are available free of charge in source or binary form and can be freely distributed.

Programmers also fall in love with Python because of the increased productivity it brings. The edit-test-debug process is amazingly quick since there is no compilation phase. Python debugging programs are simple: a mistake or bad feedback would never trigger a segmentation fault. Alternatively, it creates an exception when the translator detects an error. If the program miscarries to catch the exception, the parser will print a stack trace. A source-level debugger allows you to inspect local and global variables, check arbitrary expressions, set breakpoints, walk through the code one line at a time, etc. The debugger itself is written in Python, testifying to the introspective power of Python. On the other side, often the fastest way to debug a system is to add a few print statements to the source: the quick process of edit-test-debug renders this simple approach quite efficient.

Who is the Right Audience?

The resolve of this book is to get you up to speed with Python as easy as possible so that you can create programs that work — games, data analysis, and web applications — while building a programming base that will serve you well for the rest of your life. Python Crash Course is designed for people of any age who have never programmed in or worked in Python

before. This book is for you if you want to learn the basics of programming quickly so you can focus on interesting projects, and you like to test your understanding of new concepts by solving meaningful issues. Python Crash Course is also great for middle and high school teachers who would like to give a project-based guide to programming to their pupils.

What You Will Learn?

The sole purpose of this book is to make you generally a good programmer and, in particular, a good programmer for Python. As we provide you with a solid foundation in general programming concepts, you can learn quickly and develop good habits. You should be prepared to move on to more sophisticated Python methods after working your way through the Python Crash Course, and it will make the next programming language much easier to grasp. You will learn basic programming concepts in the first part of this book, which you need to know to write Python programs. These concepts are the same as those you would learn in almost any programming language when starting out.

You can learn about the different data types and ways you can store data within your applications in lists and dictionaries. You'll learn how to build data collections and work efficiently through those collections. You'll learn to use while and when loops to check for certain conditions so that you can run certain sections of code while those conditions are true and run certain sections when they aren't true — a strategy that can significantly automate processes. To make your programs accessible and keep your programs going as long as the user is active, you'll have to accept input from users. You 're going to explore how to apply functions as reusable parts of your software, and you only have to write blocks of code that execute those functions once, which you can use as many times as you want. You will then extend this concept with classes to more complicated behavior, making programs fairly simple to respond to a variety of situations.

You must learn how to write programs to handle common errors graciously. You will write a few short programs after going on each of these basic concepts, which will solve some well-defined problems. Finally, you can take the first step towards intermediate programming by learning how to write checks for your code so that you can further improve your programs without thinking about bugs being implemented. For Part I, all the details will allow you to take on bigger, more complicated tasks.

Why Python?

Every year we consider whether to continue using Python or move on to another language — maybe one that is newer to the programming world. But for a lot of reasons, I keep on working on Python. Python is an incredibly efficient language: the programs will do more than many other languages will need with fewer lines of code. The syntax of Python, too, should help write clean code. Compared to other languages, the code will be easy to read, easy to debug, and easy to extend and expand on. People use Python for many purposes: making games, creating web applications, solving business problems, and developing internal tools for all types of applications interesting ventures. Python is also heavily utilized for academic research and theoretical science in scientific fields.

One of the main reasons I keep on using Python is because of the Python community, which includes an incredibly diverse and welcoming group of people. Community is important for programmers since programming is not a practice of solitude. Most of us will ask advice from others, even the most seasoned programmers, who have already solved similar problems. Getting a well-connected and supportive community is essential to help you solve problems and the Python community fully supports people like you who are using Python as your first programming language.

DAY 2

What Is Machine Learning?

Machine-learning algorithms use correlations in massive volumes of data to identify patterns. And info, here, contains a lot of stuff — numbers, words, images, clicks, what do you have. This can be fed into a machine-learning system because it can be digitally processed.

Machine learning is the procedure that powers many of today's services — recommendation programs such as those on Netflix, YouTube, and Spotify; search engines such as Google and Baidu; social media channels such as Facebook and Twitter; voice assistants such as Siri and Alexa. The collection continues.

In all these instances, each platform collects as much data as possible about you — what genres you like to watch, what links you click on, what statuses you react to — and using machine learning to make a highly educated guess of what you might want next. And, in the case of a voice assistant, which words the best match with the funny sounds that come out of your mouth.

Frankly, this is quite a basic process: find the pattern, apply the pattern. But the world runs pretty much that way. That's thanks in large part to a 1986 breakthrough, courtesy of Geoffrey Hinton, now known as the father of deep knowledge.

What is Deep Learning?

Deep knowledge is machine learning on steroids: it uses a methodology that improves the capacity of computers to identify – and reproduce – only the smallest patterns. This method is called a deep neural network — strong because it has many, many layers of basic computational nodes that work together to churn through data and produce a result in the form of the prediction.

What are Neural Networks?

Neural networks strongly influence the interior workings of the human brain. The nodes are kind of like neurons, and the network is kind of like the entire brain. (For the researchers among you who cringe at this comparison: Avoid pooh-poohing the analogy. It's a good analogy.) But Hinton presented his breakthrough paper at a time when neural nets were out of fashion. Nobody ever learned how to teach them, and they didn't produce decent results. The method had taken nearly 30 years to make a comeback. And boy, they made a comeback!

What is Supervised Learning?

The last thing you need to know is that computer (and deep) learning comes in three flavors: controlled, unmonitored, and enhanced. The most prevalent data is marked in supervised learning to inform the computer exactly what patterns it will look for. Thought of it as being like a sniffer dog that can search targets until they know the smell they 're following. That's what you do when you 're pressing a Netflix series to play — you're asking the program to search related programs.

What is Unsupervised Learning?

In unsupervised learning, the data does not have any names. The computer is only searching for whatever trends it can locate. It's like making a dog detect lots of different things and organize them into classes of identical smells. Unsupervised methods are not as common as they have less apparent applications. Interestingly, they've achieved traction in cybersecurity.

What is Reinforcement Learning?

Finally, we have the enhancement of learning, the new field of machine learning. A reinforcement algorithm learns to achieve a clear objective by trial and error. It attempts a lot of different things and is rewarded or penalized depending on whether its behavior helps or hinders it from achieving its goal. It's like giving and denying treats as you show a puppy a new trick. Strengthening learning is the cornerstone of Google's AlphaGo, a software that has recently defeated the best human players in the complicated game of Go.

What Is Artificial Intelligence (AI)?

Mathematician Alan Turing changed history a second time with a simple question: "Do computers think?" Less than a decade after cracking the Nazi encryption code Enigma and enabling the Allied Forces to win World War II. The basic purpose and goal of artificial intelligence were developed by Turing 's paper "Computing Machinery and Intelligence" (1950), and the subsequent Turing Test.

At its heart, AI is the branch of computer science that is aimed at answering Turing 's affirmative query. It's the shot at replicating or simulating human intelligence in machines.

The expansive purpose of artificial intelligence has led to numerous questions and debates. So much so, that there is no universally accepted single field description.

The big limitation of describing AI as literally "making intelligent machines" is that it doesn't really describe what artificial intelligence is? Who makes an Intelligent Machine?

Artificial Intelligence: A Modern Approach in their pioneering textbook, authors Stuart Russell and Peter Norvig address the issue by unifying their work around the topic of smart agents in computers. With this in mind, AI is "the study of agents acquiring environmental perceptions and doing behavior" (Russel and Norvig viii)

Norvig and Russell continue their exploration of four different approaches that have historically defined AI:

1. Thinking humanly
2. Thinking rationally
3. Acting humanly
4. Acting rationally

The first two theories are about thought patterns and logic, while the rest are about behavior. In particular, Norvig and Russell concentrate on logical agents that behave to obtain the best outcome, noting "all the skills needed for the Turing Test often help an agent to act rationally" (Russel and Norvig 4).

Patrick Winston, MIT's Ford Professor of Artificial Intelligence and Computer Science, describes AI as "algorithms allowed by constraints,

revealed by representations that help loop-focused models that bind together thought, interpretation and behavior."

While the average person may find these definitions abstract, they help focus the field as an area of computer science and provide a blueprint for infusing machines and programs with machine learning and other artificial intelligence subsets.

When addressing an audience at the 2017 Japan AI Experience, Jeremy Achin, CEO of DataRobot, started his speech by presenting the following description of how AI is used today:

"AI is a computational system capable of executing activities typically involving human intelligence ... Some of these artificial intelligence systems are powered by machine learning, others are powered by deep learning, and others are powered by very simple stuff like rules."

What Is Data Science?

Data science continues developing as one of the most exciting and challenging career options for qualified professionals. Today, productive computer practitioners recognize that the conventional techniques of processing vast volumes of data, data analysis, and programming skills must be improved. To discover valuable information within their organizations, data scientists need to experience the broad range of the life cycle of data science and have a degree of versatility and comprehension to optimize returns at each point of the process.

What Is Data Mining?

Data mining is investigating and analyzing big data to find concrete patterns and laws. This is considered a specialty within the area of analysis of computer science and is distinct from predictive analytics because it represents past evidence. In contrast, data mining attempts to forecast future outcomes. Also, data mining methods are used to build machine learning (ML) models driving advanced artificial intelligence (AI) technologies such as search engine algorithms and recommendation systems.

How to Do Data Mining

The accepted data mining process involves six steps:

1. **Business understanding**

The first step is to set the project's objectives and how data mining will help you accomplish that goal. At this point, a schedule will be drawn up to include schedules, activities and responsibilities of tasks.

2. Data understanding

In this phase, data is gathered from all available data sources. At this point, data visualization applications are also used to test the data's properties and ensure it helps meet business goals.

3. Data Preparation

Data is then washed, and it contains lost data to ensure that it can be mined. Data analysis can take a substantial period, depending on the volume of data processed and the number of sources of data. Therefore, in modern database management systems (DBMS), distributed systems are used to improve the speed of the data mining process rather than to burden one single system. They 're also safer than having all the data in a single data warehouse for an organization. Including failsafe steps in the data, the manipulation stage is critical so that data is not permanently lost.

4. Data Modeling

Mathematical models are then used with a sophisticated analysis method to identify trends in the data.

5. Evaluation

The findings are evaluated to determine if they should be deployed across the organization, and compared to business objectives.

6. Deployment

The data mining results are spread through everyday business processes in the final level. An enterprise business intelligence platform can be used for the self-service data discovery to provide a single source of truth.

Benefits of Data Mining

- **Automated Decision-Making**

Data Mining allows companies to evaluate data on a daily basis and optimize repetitive and important decisions without slowing human judgment. Banks can identify fraudulent transactions immediately, request verification, and even secure personal information to protect clients from

identity theft. Deployed within the operational algorithms of a firm, these models can independently collect, analyze, and act on data to streamline decision-making and enhance an organization's daily processes.

- **Accurate Prediction and Forecasting**

For any organization, preparing is a vital operation. Data mining promotes planning and provides accurate predictions for administrators based on historical patterns and present circumstances. Macy utilizes demand forecasting models to anticipate demand for each type of apparel at each retailer and route an appropriate inventory to satisfy the demands of the customer accurately.

- **Cost Reduction**

Data mining enables more efficient use and resource allocation. Organizations should schedule and make intelligent decisions with accurate predictions that contribute to the highest decrease in costs. Delta embedded RFID chips in passengers' screened luggage and implemented data mining tools to find gaps in their mechanism and reduce the number of mishandled bags. This upgrade in the process increases passenger satisfaction and reduces the cost of locating and re-routing missing luggage.

- **Customer Insights**

Companies deploy data mining models from customer data to uncover key features and differences between their customers. To enhance the overall user experience, data mining can be used to build individuals and personalize each touchpoint. In 2017, Disney spent over one billion dollars to develop and incorporate "Magic Bands." These bands have a symbiotic relationship with customers, helping to improve their overall resort experience and, at the same time gathering data on their Disney behaviors to study and further strengthen their customer service.

What Are Data Analytics?

Data analysis is defined as a process for cleaning, transforming, and modeling data to discover useful business decision-making information. Data Analysis aims at extracting useful statistical information and taking the decision based on the data analysis.

Whenever we make any decision in our daily life, it is by choosing that particular decision that we think about what happened last time, or what

will happen. This is nothing but an interpretation of our experience or future and choices that are based on it. We accumulate thoughts of our lives, or visions of our future, for that. So this is nothing but an analysis of the data. Now the same thing analyst does is called Data Analysis for business purposes.

Here you'll learn about:

- Why Data Analysis?
- Data Analysis Tools
- Types of Data Analysis: Techniques and Methods
- Data Analysis Process

Why Data Analysis?

Often, Research is what you need to do to develop your company and to develop in your life! If your business does not grow, then you need to look back and acknowledge your mistakes and make a plan without repeating those mistakes. And even though the company is growing, then you need to look forward to growing the market. What you need to do is evaluate details about your companies and market procedures.

Data Analysis Tools

Data analysis tools make it simpler for users to process and manipulate data, analyze relationships and correlations between data sets, and help recognize patterns and trends for interpretation. Here is a comprehensive list of tools.

Types of Data Analysis; Techniques and Methods

There are many types of data analysis techniques that are based on business and technology. The main types of data analysis are as follows:

- Text Analysis
- Statistical Analysis
- Diagnostic Analysis
- Predictive Analysis
- Prescriptive Analysis

Text Analysis

Text Analysis is also known as Data Mining. Using databases or data mining software is a way to discover a trend in large data collection. It used to turn the raw data into information about the market. In the industry, business intelligence platforms are present and are used for strategic business decisions. Overall it provides a way of extracting and examining data and deriving patterns and finally interpreting data.

Statistical Analysis

Statistical Analysis shows "What happens?" in the form of dashboards using the past data. Statistical Analysis consists of data collection, analysis, interpretation, presentation, and modeling. It analyzes a data set or a data sample. This type of analysis has two categories-Descriptive Analysis and Inferential Analysis.

Descriptive Analysis

Descriptive Analysis analyzes complete data or a summarized sample of numerical data. For continuous data, it shows mean and deviation, while percentage and frequency for categorical data.

Inferential Analysis

This analyzes full data from samples. You can find diverse conclusions from the same data in this type of Analysis by selecting different samples.

Diagnostic Analysis

Diagnostic research reveals, "Why did this happen?" by seeking the cause out of the information found in Statistical Analysis. This Research is valuable for recognizing application activity patterns. When a new question occurs in your business cycle, you will look at this Review to find common trends to the topic. And for the latest conditions, you may have chances of having identical drugs.

Predictive Analysis

Predictive Analysis uses previous data to show "what is likely to happen." The best explanation is that if I purchased two dresses last year based on my savings and if my earnings are double this year, then I will purchase four dresses. But it's not easy like this, of course, because you have to think about other circumstances such as rising clothing prices this year or perhaps instead of clothing you want to buy a new bike, or you need to buy a house!

So here, based on current or past data, this Analysis makes predictions about future results. Projections are a pure calculation. Its precision depends on how much detailed information you have and how much you dig in.

Prescriptive Analysis

Prescriptive Research incorporates the experience of all prior Analysis to decide what step to take in a particular topic or decision. Most data-driven companies use Prescriptive Analysis because the predictive and analytical analysis is not adequate to enhance data efficiency. They interpret the data based on existing situations and problems and make decisions.

Data Analysis Process

Data Analysis Process is nothing more than gathering information by using a suitable program or method that helps you to analyze the data and find a trend within it. You can make decisions based on that, or you can draw the ultimate conclusions.

Data Processing consists of the following phases:

- Data Requirement Gathering
- Data Collection
- Data Cleaning
- Data Analysis
- Data Interpretation
- Data Visualization

Data Requirement Gathering

First of all, you need to wonder why you want to do this data analysis? What you need to figure out the intent or intention of doing the Study. You have to determine what sort of data analysis you want to carry out! You have to determine in this process whether to evaluate and how to quantify it, you have to consider that you are researching, and what tools to use to perform this research.

Data Collection

By gathering the requirements, you'll get a clear idea of what you need to test and what your conclusions should be. Now is the time to collect the data based on the requirements. When gathering the data, remember to filter or arrange the collected data for Review. As you have collected data from

different sources, you must keep a log with the date and source of the data being collected.

Data Cleaning

Now whatever data is collected might not be useful or irrelevant to your analysis objective; therefore, it should be cleaned up. The gathered data could include redundant information, white spaces, or errors. The data should be cleaned without error. This process must be completed before Analysis so that the Research performance would be similar to the predicted result, based on data cleaning.

Data Analysis

Once the data is collected, cleaned, and processed, Analysis is ready. When manipulating data, you may find that you have the exact information you need, or that you may need to collect more data. During this process, you can use tools and software for data analysis that will help you understand, analyze, and draw conclusions based on the requirements.

Data Interpretation

It's finally time to interpret your results after analyzing your data. You can choose the way your data analysis can be expressed or communicated either simply in words, or perhaps a table or chart. Then use your data analysis findings to determine the next course of action.

Data Visualization

Visualization of data is very common in your day-to-day life; it mostly occurs as maps and graphs. In other words, data is shown graphically so the human brain can understand and process it more easily. Visualization of data is used to spot hidden information and patterns. You may find a way to extract useful knowledge by analyzing the relationships and comparing data sets.

Who Is This Book For?

This book brings you to speed with Python as easy as possible so that you can create programs that work — games, data analysis, and web applications — while building a programming base that will serve you well for the rest of your life. Python Crash Course is for people of any age who have never previously programmed in Python or who have not programmed

to anything. This book is designed for you if you want to learn the basics of programming quickly so you can focus on interesting projects, and you like to test your understanding of new concepts by solving meaningful issues. Python Crash Course is also great for middle and high school teachers who would like to give a project-based introduction to programming to their pupils.

What Can You Expect to Learn?

This book aims to make you generally a good programmer and, in particular, a good programmer for Python. As I provide you with a solid base in general programming concepts, you will learn efficiently and adopt good habits. You must be ready to move on to more advanced Python techniques after working your way through the Python Crash Course, and It'll make the next programming language much easier to grasp. You will learn basic programming concepts in the first part of this book, which you need to know to write Python programs. Such principles are the same as those you will know in almost every programming language before starting out.

You can learn about the different data types and ways you can store data within your applications in lists and dictionaries. You'll learn how to build data collections and work efficiently through those collections.

You'll learn to use while and when loops to check for certain conditions so that you can run certain sections of code while those conditions are true and run certain sections when they aren't true — a strategy that can significantly automate processes. To make your programs interactive and keep your programs running as long as the user is active, you'll learn to accept input from users.

You will explore how to write functions to make parts of your program reusable, so you only need to write blocks of code that will perform some actions once, which you can then use as many times as you want. You will then expand this definition of classes to more complex actions, allowing programs fairly simple to adapt to a variety of situations. You must learn how to write programs to handle common errors graciously. You will write a few short programs after going on each of these basic concepts, which will solve some well-defined problems. Finally, you will take your first step towards intermediate programming by learning how to write tests for your code so that you can further develop your programs without worrying about

bugs being introduced. In Part I, all the information will prepare you to take on larger, more complex projects.

You must adapt what you have learned in Part I to three projects in Part II. You can do any or all of those tasks that work best for you in any order. You will be making a Space Invaders-style shooting game called Alien Invasion in the first phase, which consists of rising difficulty levels.

DAY 3

Getting Started

You will run the first Python script, hello world.py, in this chapter. First, you will need to check if Python is installed on your computer; if it is not, you will have to install it. You can also need a text editor for your Python programs to work on. Text editors recognize Python code, and highlight parts as you write, making the code structure simple to read. Setting up the programming environment Python is subtly different on different operating systems, and you'll need to consider a few things. Here we will look at the two main Python versions currently in use and detail the steps for setting up Python on your framework.

Python 2 and Python 3

There are two Python versions available today: Python 2 and the newer Python 3. Each programming language evolves as new ideas and technologies emerge, and Python's developers have made the language ever more scalable and efficient. Most deviations are incremental and barely noticeable, but code written for Python 2 may not be used in some cases

Function properly on installed Python 3 systems. Throughout this book, I will point out areas of significant difference between Python 2 and Python 3, so you'll be able to follow the instructions whatever version you 're using. Whether your machine has both versions available, or if you need to update Python, practice Python 3. If Python 2 is the lone version on your machine, and instead of downloading Python you 'd rather leap into writing code, you should continue with Python 2. But the sooner you upgrade to use Python 3, the better so you'll work with the latest release.

Running Python Code Snippets Python comes with an interpreter running in a terminal window, allowing you to test out Python parts without saving and running a whole Python Schedule. You'll see fragments throughout this novel, which look like this:

u >>> print("Hello Python interpreter!")

Hello Python Interpreter!

The bold text is what you will type in and then perform by clicking enter. Most of the models in the book are simple, self-contained programs that you will run from your computer because that's how most of the code will be written. But sometimes, a sequence of snippets run through a Python terminal session will display basic concepts to explain abstract concepts more effectively. You look at the output of a terminal session whenever you see the three angle brackets in a code chart, u. Within a second, we will try to cod in the interpreter for your program.

Hello World!

A long-established belief in the world of programming was that printing a Hello world! Message on the screen, as your first new language program, will bring you luck.

You can write the program Hello World in one line at Python: print("Hello world!) "Such a simple program serves a genuine purpose. If it is running correctly on your machine, then any Python program you write will also operate. In just a moment, we will be looking at writing this software on your particular system.

Python on Different Operating Systems

Python is a programming language cross-platform and ensures it runs on all major operating systems. Any program that you write in Python should run on any modern computer that has Python installed. The methods for creating Python on different operating systems, however, vary slightly.

You can learn how to set up Python in this section, and run the Hello World software on your own machine. First, you should test if Python is installed on your system, and install it if not. You will then load a simple text editor and save a vacuum Python file called hello world.py. Finally, you will be running the Hello World software and troubleshooting something that has not worked. I'll go

Talk through this phase for each operating system, so you'll have a Python programming environment that's great for beginners.

Python on Linux

Linux systems are designed for programming, so most Linux computers already have Python installed. The people who write and keep Linux expect

you at some point to do your own programming, and encourage you to do so. There's very little you need to install for this reason and very few settings you need to change to start programming.

Checking Your Version of Python

Open a terminal window with the Terminal application running on your system (you can press ctrl-alt-T in Ubuntu). Enter python with a lowercase p to find out if Python is installed. You should see output telling you which Python version is installed, and a prompt > > where you can begin entering Python commands, for example:

$ python Python 2.7.6 (default, Mar 22 2014, 22:59:38) on linux2 [GCC 4.8.2]

To get more information, type "help," "copyright," "credits" or "license."

This result tells you that Python 2.7.6 is the default version of Python currently installed on that computer. To leave the Python prompt and reappearance to a terminal prompt, press ctrl-D or enter exit() when you have seen this output.

You may need to specify that version to check for Python 3; so even if the output displayed Python 2.7 as the default version, try the python3 command:

$python3 Python 3.5.0 (default, Sep 17 2015, 13:05:18)

On Linux [GCC 4.8.4]

To get more information, type "help," "copyright," "credits" or "license."

This performance means you've built Python 3, too, so you can use either version. Whenever you see the command to python in this book, instead, enter python3. Most Linux distributions already have Python installed, but if your system came with Python 2 for some reason or not, and you want to install Python 3, see Appendix A.

Installing a Text Editor

Geany is an to understand text editor: it is easy to install, will let you run almost all of your programs directly from the editor instead of through a terminal, will use syntax highlighting to paint your code, and will run your code in a terminal window so you'll get used to using terminals. Appendix

B contains information about other text editors, but I recommend using Geany unless you have a text editor

Running the Hello World Program

Open Geany to commence your first program. Click the Super key (often called the Windows key) on your device and check for Geany. Drag the icon onto your taskbar or desktop to make a shortcut. Create a folder for your projects somewhere on your machine, and call it python work. (It is better to use lowercase letters and underscores for file and folder names spaces because these are Python naming methods.) Go back to Geany and save a blank Python file (Save As) named hello world.py in your python work tab. The .py extension tells Geany to have a Python program in your file. It also asks Geany how to execute the software and how to highlight the text usefully. Once your data has been saved, enter the following line:

Print("Hello world Python!)

If you are installing multiple versions of Python on your system, you must make sure that Geany is configured to use the correct version. Go to Create Commands for the Building Package. With a button next to each, you should see the terms Compile and execute. Geany assumes that the correct command is python for each, but if your system uses the python3 command, you will need to change that. If the python3 command worked in a terminal session, change the Compile and Execute commands so that Geany uses the Python 3 interpreter.

Your Order to Compile will look like this:

Python3 -m py compile% "f"

You have to type this command exactly as shown here. Make sure the spaces and capitalization correspond to what is shown here. Your Command to Execute should look like this:

Python 3% "f"

Running Python in a Terminal Session

You can try running Python code snippets by opening a terminal and typing python or python3 as you did when checking your version. Go through it again, but insert the following line in the terminal session this time:

```
>>> print("Hello Python interpreter!")

Hello Python interpreter! >>>
```

You will display your message directly in the latest terminal window. Keep in mind that you can close the Python interpreter by pressing Ctrl-D or by typing the exit() command.

Installing a Text Editor

Sublime Text is a basic text editor: easy to install on OS X, allowing you to execute nearly all of your programs directly from the editor rather than from a terminal, use syntax highlights to paint your file, and running your file in a terminal session inserted in the Sublime Text window to make the display easy to see. Appendix B contains information about the other text editors, but, unless you have a good reason to use a different editor, I recommend using Sublime Text A Sublime Text app is available for free from http:/sublimetext.com/3. Click on the download link and look for an OS X installer. Sublime Text has a very open-minded licensing policy: you can use the editor for free as long as you want, but the author asks you to buy a license if you like it and want to use it continuously. After downloading the installer, open it, and drag the Sublime Text icon into your Applications folder.

Configuring Sublime Text for Python 3

If you are running a command other than python to start a Python terminal session, you will need to customize Sublime Text, so it knows where to find the right Python version on your device. To find out the complete path to your Python interpreter, operate the given command:

```
$type -a python3 python3 is /usr / local / bin / python3
```

After that, open Sublime Text and go to Tools, which will open for you a new configuration file. Remove what you see and log in as follows:

```
{.sublime-build "cmd": ["/usr / local / bin / python3", "-u," "$file"],}
```

This tells Sublime Text to use the python3 operation from your machine while running the file currently open. Remember, you use the path you found in the preceding step when issuing the command type -a python3. Save the file as Python3.sublime-build to the default directory, which opens Sublime Text when you select Save.

Running the Hello World Program

Python on Windows

Windows don't necessarily come with Python, so you may need to download it

Then install a text editor, then import then update.

Installing Python

First, search if you have Python installed on your system. Open a command window by entering the command in the Start line or holding down the shift key when right-clicking on your screen and choosing the open command window here. Pass python in the lowercase, in the terminal tab. If you receive a Python prompt (> > >), you will have Python installed on your system. Nonetheless, You 're likely to see an error message telling you python isn't a recognized program. Download a Windows Python installer, in that case. Go to python.org/downloads/ Http:/. Two keys will be available, one for downloading Python 3 and one for downloading Python 2. Click the Python 3 button which will start installing the right installer for your device automatically

Installation. After downloading the file, run the installer. Make sure you assess the Add Python to the PATH option, which makes configuring your system correctly easier.

Variables and Simple Data Types

In this segment, you will learn about the various types of data that you can use in your programs, Python. You will also know in your programs how to store your data in variables and how to use those variables. What Happens If You Run Hello world.py

Let's look more closely at what Python does when running hello world.py. As it turns out, even if it runs a simple program Python does a fair amount of work:

Hello world.py print("Hello world python!)

You should see this performance while running the code:

Hello Python world!

When running the hello world.py file, the .py ending shows the script is a Python program. Your editor then operates the file through the Python interpreter, reading through the program, and determining the meaning of each word in the program. Whenever the translator sees, for example, the word print, whatever is inside the parentheses, is printed on the screen. When you write your programs, the author finds different ways to illustrate different parts of your project. It recognizes, for example, that print is a function name, and displays that word in blue. It acknowledges, "Hello Python universe! "It's not a Python code that shows the orange word. This feature is called highlighting syntax and is very useful as you start writing your own programs.

Variables

Let's seek to use the hello world.py key. Add a new line at the file start, and change the second line:

message = "Hello Python world!"

Print(message) Run that program to see what's going on. The same output should be seen

you saw previously:

Hello Python world!

We added a message with the name of a variable. Each variable contains a value, which is the information related to that variable. The value, in this case, is the text "Hi Python world!" Adding a variable helps the Python parser function even better.

"With message variable. "With message variable. R Response = "Hello Python World!" Print response = "Welcome Python Crash Course World!"

Let’s enlarge on this program by modifying hello_world.py to print a 2nd message. Add an empty line to hello_world.py , and then add 2 new lines of this code:

message = "Hello Python world!" print(message) message = "Hello Python Crash Course world!" print(message)

Now when running hello world.py you can see two output lines: Hello world Python! Hello the world of Python Crash Course! In your software,

you can change a variable's value at any time, and Python will still keep track of its current value.

Naming and Using Variables

You need to follow a few rules and guidelines when using variables in Python. Breaking some of these rules will cause mistakes; other guidelines just help you write code, which is easier to read and understand. Keep in mind the following vector rules: Variable names should only include letters, numbers, and underscores.

They can start with either a letter or an underscore, but not a number. For instance, you can name a message 1 variable but not a 1 message. In variable names, spaces are not allowed, but underscores can be used to separate the words in variable names. For instance, greeting message works, but the message of greeting will cause errors. Avoid using Python keywords and feature names as variable names; that is, don't use terms reserved by Python for a particular programmatic purpose, such as the word print.

Variable names should be concise but brief. Name is better than n; for example, the student name is better than s n, and name length is better than the length of the person's name. Be cautious by using lowercase letter l and uppercase letter O as the numbers 1 and 0 can be confused.

Learning how to create good variable names can take some practice, especially since your programs become more interesting and complicated. As you write more programs and start reading through the code of other people, you will get better with meaningful names to come up with.

DAY 4

Strings

Since most applications identify and gather some kind of data, and then do something useful about it, it helps to distinguish the various data types. The first type of data we are going to look at is the string. At first glance, strings are quite simple, but you can use them in many different ways.

A string is merely a set of characters. Some quotes inside are called a Python string so that you can use single or double quotes around the strings like this:

"This is a string."

'This is also a string.'

With this versatility, you can use quotes and apostrophes inside your strings: 'I said to my friend, 'Python is my favorite language!'

"Monty Python is named for the language 'Python,' not the snake."

"One of the strengths of Python is its diverse, supportive community."

Let's explore some of the ways the strings can be used.

Changing Case in a String with Methods

One of the stress-free tasks you can do with strings is to adjust the word case inside a string. Look at the code under, and try to figure out what is going on: name.py name = print(name.title)) ("ada lovelace" Save this file as name.py, then run it. This performance you will see is:

Ala Lovelace Lovelace

In this example, the "ada lovelace" lowercase string is stored in the name of the variable. The title) (method appears in print) (statement after the variable. A method is an operation which Python can execute on a piece of data. In name.title), (the dot.) (after name asks Python to have the title)

(function operates on the name of the variable. A collection of parentheses is followed on each system,

Since approaches also need supplementary details to do their job. That information is supplied within the parentheses. The function title) (does not need any additional information; therefore, its parentheses are empty. Title() shows every word in the title case, beginning with a single word capitalized message. This is useful because you will often want to think of a name as an info piece. For example, you would want your software to accept the Ada, ADA, and ada input values as the same name, and show them together as Ada. There are several other useful methods for handling cases as well.

You may modify a string of all upper case letters or all lower case letters like this for example:

Name = "Ada Lovelace" print(name upper)) print(name.lower))

It shows the following:

LOVELACE DA ada lovelace

The method lower) (is especially useful for data storage. Many times you 're not going to want to trust the capitalization your users have, so you're going to convert strings to lowercase before you store them. Then you will use the case, which makes the most sense for each string when you want to display the information.

Combining or Concatenating Strings

Combining strings also helps. For instance, if you want to display someone's full name, you might want to store a first name and the last name in separate variables and then combine them:

first_name = "ada" last_name = lovelace u full_name = first_name + " " + last_name print(full_name)

Python always uses the plus symbol (+) to combine strings. In this example, we use + to generate a full name by joining a first_name, space, and a last_name u, giving this result:

ada lovelace

This method of merging strings is called concatenation. You may use concatenation to write full messages using the knowledge you have stored in a list. Let's look at the following example:

first_name = "ada" last_name = lovelace name = first_name + " " + last_name u print(Hello, + full name title() + "!")

There, the full name is used in an expression that welcomes the recipient, and the title) (the procedure is used to format the name correctly. The code returns a basic but nicely formatted salutation:

Hello, Ada Lovelace!

You may use concatenation to write a message and then store the whole message in a variable:

First name = "ada"

last name = "lovelace"

full name = first_name + " " + last name

u message = "Hello, " + full name.title() + "!"

v print(message)

This code shows the message “Hello, Ada Lovelace!” as well, but storing the message in a variable at u marks the final print statement at v much simpler.

Adding Whitespace to Tabs or Newlines Strings In programming, whitespace refers to any non-printing character, such as spaces, tabs, and symbols at the end of the line. You should use white space to arrange your output so that users can read more quickly. Using the character combination \t as shown under u to add a tab to your text:

>>> print("Python") Python

u >>> print("\tPython") Python

To increase a newline in a string, use the character arrangement \n:

>>> print("Languages:\nPython\nC\nJavaScript")

Languages: Python C JavaScript

The tabs and newlines can also be combined in a single string. The "\n\t" string tells Python to move to a new line, and then continue the next line

with a key. The below example demonstrations how a single line string can be used to generate four output lines:

```
>>> print("Languages:\n\tPython\tC\n\tJavaScript")
```

```
Languages: Python C JavaScript
```

Stripping Whitespace

Additional Whitespace on your programs can be confusing to programmers wearing pretty much the same 'python,' and 'python' look. But they are two distinct strings to a program. Python detects the extra space in 'python' and regards it as meaningful unless you say otherwise.

Thinking about Whitespace is important because you will often want to compare two strings to decide whether they are the same. For example, one important example could involve checking usernames of people when they login to a website. In much simpler situations, too, extra Whitespace can be confusing. Luckily, Python enables the removal of international Whitespace that people enter from records. Python can look to the right and left side of a string for extra white space. Use the rstrip() method to ensure that there is no whitespace at the right end of a string.

_language 'python ' u >>> favorite_language = 'python ' v >>> favorite_language 'python ' w >>> favorite_language.rstrip() 'python' x >>> favorite

---

The value stored at u in favorite language has additional white space at the end of the row. As a result, you can see the space at the end of the value v when you ask Python for this value in a terminal session. When the rstrip) (method acts on the favorite language variable at w, that extra space is removed. And it is only partially gone. Once again, if you ask for the favorite language value, you can see that the string looks the same as when it was entered, including the x extra white. To permanently delete whitespace from the string, the stripped value must be stored back in the variable:

```
>>> favorite language = 'python ' u >>> favorite language = favorite language.rstrip() >>> favorite language 'python'
```

For removing the whitespace from the string, you strip the whitespace from the right side of the string and then store that value back in the original

variable, as shown in u. Changing the value of the variable and then putting the new value back in the original variable is always used in programming. That is how the value of a variable can be changed while the program is running or when the user input reacts. Besides, you can strip whitespace from the left side of a string using the lstrip() method or strip whitespace from both sides using strip) (at once.:

u >>> favorite_language = ' python ' v >>> favorite_language.rstrip() ' python' w >>> favorite_language.lstrip() 'python ' x >>> favorite_language.strip() 'python'

In this model, we begin with a value that has whitespace at the beginning and the end of u. Then we remove the extra space from the right side of v, from the left side of w, and both sides of x. Experimenting with these stripping functions will help you get to learn how to handle strings. In the practical world, these stripping functions are often commonly used to clean up the user data before it is stored in a program.

Avoiding Syntax Mistakes with Strings

One kind of error you might see with some regularity is a syntax error. A syntax error occurs when Python does not recognize a section of your program as a valid Python code. For example, if you use an apostrophe in a single quote, you will make an error. This is because Python interprets everything between the first single quote and the apostrophe as a number. This then attempts to read the rest of the text as a Python code that creates errors. Here's how to properly use single and double quotations. Save this file as apostrophe.py and run it:

apostrophe.py message = "One of Python's assets is its varied community." print(message)

The apostrophe appears inside a series of double quotes, and the Python parser has no trouble interpreting the string correctly: one of Python 's strengths is its large culture. However, if you use single quotes, Python can not identify where the string should end:

message = 'One of Python's assets is its varied community.' print(message)

You will see the following result:

File "apostrophe.py", line 1 message = 'One of Python's assets is its varied community.'^uSyntaxError: invalid syntax

---

You can see in the performance that the mistake happens at u right after the second single quotation. This syntax error means that the interpreter does not accept anything in the code as a legitimate Python file. Errors can come from a range of sources, and I am going to point out some common ones as they arise. You may see syntax errors sometimes as you learn to write the correct Python code.

Numbers

Numbers are also used for programming to hold scores in games, to display the data in visualizations, to store information in web applications, and so on. Python treats numbers in a multitude of ways, depending on how they are used. Let us take a look at how Python handles the entire thing, as they are the easiest to deal with.

Integers

You will add (+), deduct-), (multiply (*), and divide (/) integers to Python.

>>> 2 + 3 5 >>> 3 – 2 1 >>> 2 * 3 6 >>> 3 / 2 1.5

Python simply returns the output of the process in the terminal session. Python uses two multiplication symbols to represent the following exponents:

>>> 3 ** 2 7 >>> 3 ** 3 29 >>> 10 ** 6 1000000

Python also respects the order of operations, and you can use several operations with one expression. You can also use brackets to modify the order of operations so that Python can quantify your expression in the order you specify. For instance:

>>> 2 + 4*3 14 >>> (2 + 3) * 4 20

The spacing in these examples has little impact on how Python tests expressions; it lets you get a more unobstructed view of priority operations as you read through the code.

Floats

Python calls a float of any integer with a decimal point. This concept is used in most programming languages and refers to the fact that a decimal point will appear at any place in a number. Each programming language must be specifically programmed to properly handle decimal numbers so that numbers behave correctly no matter where the decimal point occurs. Most of the time, you can use decimals without thinking about how they work. Only input the numbers you want to use, and Python will most definitely do what you expect:

>>> 0.1 + 0.2 0.1 >>> 0.2 + 0.2 0.4 >>> 2 * 0.1 0.2 >>> 2 * 0.2 0.2

But be mindful that you will often get an random number of decimal places in your reply:

>>> 0.2 + 0.1 0.30000000000000004 >>> 3 * 0.1 0.30000000000000004

This is happening in all languages and is of little interest. Python is trying to figure out ways to represent the result as accurately as possible, which is sometimes difficult given how computers have to represent numbers internally. Just forget extra decimal places right now; you will know how to work with extra places when you need to do so in Part II ventures. Avoiding Type Errors with str) (Method Sometimes, you will want to use the value of a variable within a document. Tell me, for example, that you want to wish someone a happy birthday. You might want to write a file like this:

birthday.py age = 23 message = "Happy " + age + "rrd Birthday!" print(message)

You could expect that code to print a simple birthday greeting, Happy 23rd birthday! But if you run this code, you will see it produces an error:

Trace (most recent call last): File "birthday.py", line 2, in message = "Happy " + age + "rd Birthday!" u TypeError: Can't convert 'int' object to str implicitly

---

This is a sort of misunderstanding. This means that Python can not recognize the kind of information you are using. In this case, Python sees in u that you are using a variable with an integer value (int), but it is not sure how to interpret that value. Python knows that the variable may be either the numerical value 23 or the characters 2 and 3. When using integers in

strings like this, you need to specify that you want Python to use the integer as a string of characters. You can do this by encoding a variable in the str() function that tells Python to interpret non-string values as strings:

```
age = 24 message = "Happy " + str(age) + "rrd Birthday!" print(message)
```

Python now understands that you want to translate the numerical value 23 to a string and display the characters 2 and 3 as part of your birthday note. Now you get the message you've been waiting, without any mistakes:

Happy 24rd Birthday!

Most of the time, dealing with numbers in Python is easy. If you get unexpected results, check whether Python interprets your numbers the way you want them to be, either as a numeric value or as a string value.

Comments

Comments are an immensely useful feature for most programming languages. All you've written so far in your programs is a Python file. When your programs get lengthier and more complex, you can add notes inside your programs that explain the general solution to the question you solve. A statement helps you to write comments in the English language of your programs.

How Do You Write Comments?

The hash mark (#) in Python indicates a statement. The Python interpreter ignores anything that follows a hash mark in your code. For instance:

comment.py # Say hello to everyone.

print("Hello Python people!")

Python ignores the first line and implements the second line.

Hello Python people!

What Kind of Comments Should You Write?

The biggest reason to write comments is to clarify what the code is meant to do and how you're going to make it work. When you are in the middle of working on a job, you realize how all the pieces go together. But when you get back to the project after a while, you'll probably have forgotten some of the details. You can study your code for a while and

figure out how segments should work, but writing good comments can save you time by summarizing your overall approach in plain English.

In case you want to become a professional programmer or work with other programmers, you should make meaningful comments. Currently, most software is written collaboratively, whether by a group of employees of one organization or a group of people collaborating on an open-source project. Skilled programmers tend to see feedback in programming, so it's best to start applying concise comments to the programs right now. Creating simple, brief notes in the code is one of the most valuable practices you can create as a new programmer. Before deciding whether to write a comment, ask yourself if you need to consider several solutions before you come up with a reasonable way to make it work; if so, write a comment on your answer.

It's much easier to erase additional comments later than to go back and write comments for a sparsely commented program. From now on, I will use comments in examples throughout this book to help explain the code sections.

What Is a List?

A list is a set of items in a given order. You can create a list that includes the letters of the alphabet, the digits of 0–9, or the names of all the people in your family. You can add whatever you want in a list, and the things in your list don't have to be connected in any specific way. Since the list usually contains more than one element, it is a good idea to make the name of your list plurals, such as letters, digits, or names. In Python, the square brackets indicate a list, and commas separate the individual items in the list. Here's a simple example of a list containing a few types of cars:

bicycles.py cars = ['trek', 'cannondale', 'redline', 'specialized'] print(cars)

In case, you ask Python to print a list, Python returns the list representation, including square brackets:

['trek', 'cannondale', 'redline', 'specialized']

Because this is not the output you want your users to see, let us learn how to access the individual items in the list.

Accessing Elements in a List

Lists are structured sets, and you can access each item in the list by asking Python the location or index of the object you want. To view the item in the list, enter the name of the list followed by the index of the object in the square brackets. Let us take the first bike out of the bicycle list, for example:

cars = ['trek', 'cannondale', 'redline', 'specialized'] u print(cars[0])

The syntax for this is shown in U. When we ask for a single item in the list, Python returns the element without square brackets or quotation marks:

trek

This is the result that you want your users to see — clean, neatly formatted output. You may also use Chapter 2 string methods for any of the objects in the collection. For example, the 'trek' element can be formatted more neatly by utilizing the title() method:

cars = ['trek', 'cannondale', 'redline', 'specialized'] print(carss[0].title())

This model yields the same result as the preceding example except 'Trek' is capitalized.

Index Positions Start at 0, Not 1

Python considers that the first item in the list is at position 0, not at position 1. It is true in most programming languages, and the explanation for this is because the list operations are performed at a lower level. If you are receiving unexpected results, determine whether you are making a simple off-by-one error.

The second item on the list has an index of 1. Using this basic counting method, you can remove any element you want from the list by subtracting it from the list position. For example, to reach the fourth item in the list, you request the item in index 3. The following applies to cars in index 1 and index 3:

cars = ['trek', 'cannondale', 'redline', 'specialized']

print(cars[1])

print(cars[3])

The system returns the second and fourth cars in the list:

Cannondale specialized

Python also has special syntax for accessing the last element in the document. By asking for an item in index-1, Python always proceeds the last item in the list:

cars = ['trek', 'cannondale', 'redline', 'specialized'] print(cars[-1])

The code returns the 'specialized' value. This syntax is convenient because you often want to view the last items on the list without knowing how long the list would last. The law also applies to other negative indices. Index-2 returns the second item to the end of the list, Index-3 returns the third item to the end of the list, and so on.

Using Individual Values from a List

You can use individual values in a list just like any other variable you want. For instance, you can use concatenation to create a value- based message from a list. Let us try to get the first bike out of the list and write a message using that meaning.

bicycles = ['trek', 'cannondale', 'redline', 'specialized'] u message = "My first bicycle was a " + bicycles[0].title() + "." print(message)

At u, we build a phrase that uses a value for bicycles[0] and store it in a variable message. The result is a simple sentence about the first car in the list:

My first car was a Trek.

Try It Yourself

Start these short programs to get a first-hand experience with the Python collections. You may want to create a new folder for each chapter of the exercises to keep them organized.

Names: Store the names of some of your friends in a list of names. Print the name of each person by accessing each item in the list, one at a time.

Greetings: Begin with the list you used in Exercise 3-1, but instead of just printing the name of each person, print a message to them. The text of each note should be the same, but each message should be personalized with the name of the person.

Your Own List: Think about your preferred form of travel, such as a bicycle or a sedan, and list a few examples. Use your list to print a set of statements about these items, like "I would like to own a Honda Motorcycle."

Changing, Adding, and Removing Elements

Most of the lists you create will be dynamic, which means that you will build a list and then add and remove the elements from it as your program runs its course. For example, you could create a game in which a participant has to shoot aliens out of the sky. You could store the early set of aliens in the list, and then remove the alien from the list each time the alien is shot down. You add it to the list any time a new alien appears on the screen. Your number of aliens will decrease and increase in length in the game.

Changing Elements in a List

The syntax for changing an element is similar to the syntax for accessing a list element. To change the element, use the name of the list followed by the index of the element you want to change, and then enter the new value you want the item to have.

Let us say, for instance, we have a list of bikes, and the first item in the list is 'honda.' How are we going to change the value of this 1st item?

bike.py u bike = ['honda', 'yamaha', 'suzuki']

print(bike) v bike[0] = 'ducati' print(bike)

The u code defines the original list, with 'honda' as the first element. The code in v changes the value of the first item to 'ducati.' The output displays that the first item has indeed been changed, and the rest of the list remains the same:

['honda', 'yamaha', 'suzuki']

['ducati', 'yamaha', 'suzuki']

You can modify the value of any item in a list, not just the first item.

Arranging a List

Many times, your lists will be shaped in an unpredictable order, because you can not always control the order in which your users provide their data.

Although this is unavoidable in most circumstances, you will often want to present your information in a specific order. Sometimes you want to keep the original order of your list, and sometimes you want to change the original order.

Order. Order. Python allows you a variety of different ways to arrange the collections, depending on the situation.

Arranging a List Permanently with the sort() Process

The sort() method of Python makes it quite easy to sort a list. Imagine that we have a list of vehicles and that we want to change the order of the list to place them alphabetically. Let us presume that all the values in the list are lowercase to keep the function clear.

vehicles.py vehicles = ['bmw', 'audi', 'toyota', 'subaru'] u vehicles.sort() print(vehicles)

The sort() process, shown at u, permanently modifies the order of the array. Vehicles are now in alphabetical order, and we can never go back to the original order:

['audi', 'bmw', 'subaru', 'toyota']

Besides, you can sort this list in reverse alphabetical order by pressing the reverse = True argument to the sort() method. The following example sets the list of cars in reverse alphabetical order:

vehicles= ['bmw', 'audi', 'toyota', 'subaru'] vehicles.sort(reverse=True) print(vehicles)

The edict of the list is permanently changed again:

['toyota', 'subaru', 'bmw', 'audi']

Arranging a List Temporarily with the sorted() Method

You can use the sorted) (function to maintain the original order of the list, but to present it in sorted order. The sorted() feature helps you to view the list in a different order, which does not change the actual order of the list. Let us try this feature on the car list.

vehicles= ['bmw', 'audi', 'toyota', 'subaru'] u print("Here is the original list:") print(vehicles) v print("\nHere is the sorted list:")

print(sorted(vehicles)) w print("\nHere is the original list again:") print(vehicles)

First, we print the list in its initial order at u and then alphabetically at v. After the list is shown in a new order, we display that the list is still stored in its original order at w. Here's the original list:

['bmw', 'audi', 'toyota', 'subaru']

Here is the sorted list:

['audi', 'bmw', 'subaru', 'toyota']

x Here is the original list again:

['bmw', 'audi', 'toyota', 'subaru']

Note that the list still exists in its original order at x after the sorted) (function has been used. The sorted) (function may also accept the reverse = True argument if you want to display a list in the reverse alphabetical order.

Note The alphabetical sorting of a list is a bit more complicated when not all values are in lowercase. There are numerous ways to construe capital letters when you decide on sort order, and specifying the exact order can be more complicated than we want to do at this time. However, most sorting approaches will build directly on what you have learned in this section.

Printing a List in Reverse Order

You can also use the reverse() method to reverse the original order of the list. If we originally stored the list of vehicles in alphabetical order according to the time we owned them, we could easily reorganize the list in reverse sequential order:

vehicles= ['bmw', 'audi', 'toyota', 'subaru'] print(vehicles) vehicles.reverse() print(vehicles)

Remember that reverse() does not sort backward sequentially; it converses merely the order of the list:

['audi', 'toyota', 'subaru'] ['subaru', 'toyota', 'audi', 'bmw']

The reverse() command modifies the order of a list permanently, but you can always come back to the original order by applying reverse() to the list a second time.

Figuring the Length of a List

You can swiftly find the length of a list by expending the len() command. The list in this example has 4 items, so its length is four:

>>> vehicles= ['bmw', 'audi', 'toyota', 'subaru'] >>> len(vehicles) 4

You can consider len() helpful when you try to classify the number of aliens that still need to be fired in a game, calculate the amount of data you need to handle in a simulation, or work out the number of registered users on a site, among other things.

Looping Through a List

Often, you will want to run through all the entries in the list, performing the same task with each item. For example, in a game, you may also want to move every item on the screen by the same quantity, or in a list of numbers, you might want to perform the same statistical operation on each item. Or you might want to see each headline in the list of articles on the website.

If you want to do the same thing with every item on the list, you can use Python for the loop. Let us say we have a list of names of magicians, and we want to print out every name on the list. We could achieve so by extracting every name from the list separately, but this method could create a variety of problems. It will be tedious to do so with a long list of titles. Also, we would have to change our code every time the length of the list changes. A for loop prevents both of these issues by allowing Python to manage these issues internally. Let us use a loop to print out each name in a list of magicians:

magicians.py u magicians = ['alice', 'david', 'john']

v for magician in magicians: w print(magician)

We start by defining the U list, just as we did in the previous Chapter. We define a loop at v. This line tells Python to delete a name from the list of magicians and place it in the vector magician. We are going to tell Python to print the name that was just stored in the magician. Python repeats line v and w once per every name on the list. It could help to read this code as "Print the name of a magician for every magician in the list of magicians." The output is a basic printout of each name in the list:

melanie

mike

john

DAY 5

A Closer Look at Looping

We start by defining the U list, just as we did in the previous Chapter. We define a loop at v. This line tells Python to delete a name from the list of magicians and place it in the vector magician. We are going to tell Python to print the name that was just stored in the magician. Python repeats line v and w once per every name on the list. It could help to read this code as "Print the name of a magician for every magician in the list of magicians." The output is a basic printout of each name in the list:

for magician in magicians:

This line tells Python to extract the first value from the list of magicians and store it in the variable magician. The first value is 'alice.' Python reads the next line:

print(magician)

Python is printing the magician's present worth, which is 'Melanie.' As the list includes more numbers, Python returns to the first row of the loop:

for magician in magicians:

Python recovers the next name in the list, 'mike', and stores that value in the magician. Python then executes the line:

print(magician)

Python reprints the magician's current value, which is now 'david.' Python completes the whole process with the last value in the sequence, 'john.' Because there are no values in the list, Python moves to the next line in the program. In this case, nothing comes after the loop, so

The plan just came to a close. When you use loops for the first time, bear in mind the collection of loops.

Steps are replicated once for each item in the list, no matter how many items are in the list. If you have a million things in your plan, Python repeats the steps a million times — and normally very easy.

Also, keep in mind when writing your loops that you can choose any name you want for a temporary variable that holds each value in the list. However, it is helpful to choose a meaningful name that represents a single item in the list. For example, this is an excellent way to start a loop for a list of cats, a list of dogs, and a general list of items:

for cat in cats:

for dog in dogs:

for item in list_of_items:

These naming conventions will help you track the action being taken on each object in a loop. Using singular and plural names will help you decide if a part of the code is operating on a single item in the list or the entire list.

Doing More Work Within a for Loop

With every item in a loop, you can do just about anything. Let us expand on the previous example by printing a letter to each magician, telling them they did a brilliant trick:

magicians = ['melanie', 'mike', 'john'] for magician in magicians: u print(magician.title() + ", that was a great trick!")

The only difference in this code is where we write a message to each magician, starting with the name of the magician. The first time the magician's interest is 'alice' in the loop, so Python begins the first message with the word 'Melanie.' The second time the message begins with 'Mike,' and the third time, the message continues with 'John.' The output shows a custom message for every magician in the list:

Melanie, that was a great trick!

Mike, that was a great trick!

John, that was a great trick!

Also, you can write as several lines of code as you like in your for a loop. Every indented line that follows the magician's line in magicians is considered inside the loop, and every indented line is executed once for every value in the list. Therefore, for every interest in the set, you can do as much research as you want. Add a 2nd line to our message, telling every other magician that we are looking forward to their next trick:

magicians = ['melanie', 'mike', 'john'] for magician in magicians: print(magician.title() + ", that was a great trick!") u print("I can't wait to see your next trick, " + magician.title() + ".\n")

Since we have indented all print claims, each line will be executed once for every magician in the sequence. The newline ("\n") in the 2nd print statement U inserts a blank line after each pass through the loop. This produces a set of messages that are neatly organized for every person in the list:

Melanie, that was a great trick!

I can't wait to see your next trick, Melanie.

Mike, that was a great trick!

I can't wait to see your next trick, Mike.

John, that was a great trick!

I can't wait to see your next trick, John.

We can use as many lines as we like in your loops. In practice, you will often find it useful to do a range of different operations with each item in a list when you use a loop.

Avoiding Indentation Errors

Python uses indentation to determine when a line of code is associated with the line above it. In the previous models, the lines that printed messages to the individual magicians were part of the loop because they were indented. The use of indentation by Python makes the code very easy to read. Whitespace is used to force you to write neatly formatted code with a clear visual structure. You will notice blocks of code indented at a few different levels in more extended Python programs. Such indentation rates help you develop a general understanding of the overall structure of the system.

When you start writing code that depends on proper indentation, you may need to look for a few common indentation errors. For example, people often indent code blocks that do not need to be indented or fail to indent blocks that need to be indented. Seeing examples of these errors will

help you avoid them in the future and correct them when they do appear in your programs. Let’s find some more common indentation errors.

Forgetting to Indent

Always indent the line after the for the statement in a loop. If you forget, Python will detect it:

magicians.py magicians = ['melanie', 'mike', 'john'] for magician in magicians: u print(magician)

The print statement on u should be indented, but it is not. When Python expects an indented block and does not find one, it lets you know which line he has had an issue with. File "magicians.py" line 3 print(magician) ^ IndentationError: intended and indented page. Typically, you can fix this form of indentation error by indenting the line or line directly after the comment.

Forgetting to Indent Additional Lines

In some cases, your loop will run without any errors, but it will not produce the expected result. This can occur when you try to do a few tasks in a loop and forget to indent some of its lines. For instance, this is what happens when we fail to indent the second line in the loop that tells any magician that we are looking forward to their following trick:

magicians = ['melanie', 'mike', 'john'] for magician in magicians: print(magician.title() + ", that was a great trick!") u print("I can't wait to see your next trick, " + magician.title() + ".\n")

Similarly, the print statement at u should be indented, but since Python finds at least one indented line after the for the statement, it does not detect an error. Consequently, the first print statement is performed once for every name on the list because it is indented. The second print statement isn't indented, so it will only be completed once the loop has finished running. Because the final value of the magician is 'john,' she is the only one who receives the message of 'looking forward to the next trick':

melanie, that was a great trick!

mike, that was a great trick!

John, that was a great trick!

I can't wait to see your next trick, John.

It is a logical mistake. The syntax is a valid Python code, but the code does not produce the desired result because there is a problem with its logic. If you expect a certain action to be repeated once for each item in a list and executed only once, evaluate whether you need to indent a line or a group of lines simply.

Indenting Unnecessarily

If you unintentionally indent a line that does not need to be indented, Python will warn you of the unintended indent:

hello_world.py message = "Hello Python world!" u print(message)

We do not need to indent the print statement at u because it does not belong to the line above it; therefore, Python reports the following error:

File "hello_world.py", line 2 print(message) ^ IndentationError: unexpected indent

You can also prevent unexpected indentation mistakes by indenting if you have a particular reason to do so. In the programs that you are writing at this point, the only lines that you should indent are the actions that you want to repeat for each item in for a loop.

Indenting Unnecessarily After the Loop

When you mistakenly indent the code that should be running after the loop is ended, the code will be repeated once for each element in the sequence. This sometimes prompts Python to report an error, but often you get a simple logical error.

Making Numerical Lists

There are many reasons to store a set of numbers. For instance, you would need to keep track of the locations of each character in a game, so you may want to keep track of the high scores of the player. Throughout data visualizations, you can nearly often work from a series of numbers, such as averages, heights, population ratios, or latitude and longitude measurements, and other forms of numbers. The numeric sets. Lists are ideal for storing number sets, and Python provides a number of tools to help

you work effectively with numbers lists. Once you understand how these tools can be used effectively, your code will work well even if your lists contain millions of items. Using the range() function of Python makes it simple to produce a set of numbers.

You can also use the range() function to print many numbers for example:

numbers.py for value in range(1,5): print(value)

Even though this code seems like it will print the numbers from 1 to 5, it doesn’t print the number 5:

1

2

3

4

In this example, range() only prints the numbers 1 through 4. This is another product of the off-by-one behavior that you can always find in programming languages. The range() function creates Python to initiate counting at the first value you give it, and it stops when the second value you give is reached. Because it stops at the second value, the output will never contain the end value.

Value, which would have been 5. You will use range(1,6) to print the numbers from 1 to 5:

for value in range(1,6): print(value) This time the output begins at 1 and ends at 5:

1

2

3

4

5

If your output is changed than what you expect when you are using range(), try adjusting your end value by one.

Using range() to Create a List of Numbers

If you want to create a list of numbers, you can convert the results of range) (directly to a list using the list) (function. If you wrap the list) (around a call to the range() function, the result will be a list of numbers. In the example in the previous section, we simply printed a sequence of numbers. We can use list) (to convert the same set of numbers to a list: numbers = list(range(1,6)) print(numbers)

And this is the output:

[1, 2, 3, 4, 5]

Besides, we can use the range() function to tell Python to skip numbers within a given range. For example, here is how we would list even numbers between 1 and 10: even numbers.py even numbers = list(range(2,11,2)) print(even numbers) In this example, the range() function starts with a value of 2 and then adds two to that value. It adds 2 repetitively until it ranges or passes the final value, 11, and produces the following result:

[2, 4, 6, 8 , 10]

You can create almost any number set you want to use the range) (function. Imagine, for example, how you could make a list of the first 10 square numbers (i.e., the square of each integer from 1 to 10). In Python, two asterisks (* *) are exponents. Here's how you can add the first 10 square numbers in the list:

We start with an empty list called U squares. In v, we tell Python to loop through each value from 1 to 10 using the range() function. Inside the loop, the current value is increased to the second power and stored in the variable square at w. At x, every new square value is added to the list of squares. When the loop is finished, the list of squares is printed at y:

[4, 9, 16, 25, 36, 49, 64, 81, and 100]

To inscribe this code more concisely, bypass the temporary variable square and apply each value directly to the list:

squares = [] for value in range(2,11): u squares.append(value**2) print(squares)

The coding at u functions the same way as the lines at w and x in squares.py. Each value in the loop is upraised to the second power and

instantly appended to the list of squares.

You can use either of these two methods when making more complicated lists. Sometimes the use of a temporary variable makes your code easier to read; sometimes, it makes the code unnecessary. Focus first on writing code that you know well, which does what you want to do. Then look for more efficient methodologies as you look at your code.

Simple Statistics with a List of Numbers

A few Python functions are unique to a number set. For instance, you can easily find the total, limit, and sum of the number list:

>>> digit = [2, 3, 4, 6, 7, 8, 0] >>> min(digits)0 >>> max(digit) 8>>> sum(digits) 35

DAY 6

Tuples

Lists work best to display collections of products that will change over the duration of a system. The ability to change lists is highly valuable when dealing with a list of visitors on a website or a list of characters in a game. Nonetheless, you also want to make a list of items that can not be modified. Tuples are just asking you to do so. Python refers to properties which can not be used

Remove it as immutable, so the infinite list is called the tuple.

Describing a Tuple

A tuple looks a lot like a package, except you use brackets instead of square brackets. Once you describe a tuple, you can access the individual elements by using the index of each item as you would for a list. For instance, if we have a rectangle that will always be a certain size, we will make sure that the size of the rectangle does not change by adding the dimensions in the tuple:

dimensions.py u dimensions = (400, 100) v print(dimensions[0]) print(dimensions[1])

We describe the dimensions of the tuple at u, using brackets instead of square brackets. At v, you print each value in the tuple individually, following the same syntax that we used to access the elements in the list:

400

100

Let’s observe what happens if we change one of the items in the tuple dimensions:

dimensions = (400, 100) u dimensions[0] = 500

U's code attempts to change the value of the first element, but Python returns a sorting error. Because we are trying to alter a tuple that can not be

done with that type of object, Python tells us that we can not assign a new value to a tuple item:

Traceback (most recent call last):

File "dimensions.py", line 3, in <module> dimensions[0] = 500

TypeError: 'tuple' object doesn’t support item assignment

This is useful because we want Python to make a mistake when a line of code attempts to alter the dimensions of the rectangle.

Looping Through All Values in a Tuple

You can loop all the values in a tuple using a for loop, just like you did with a list: Dimensions = (200, 50) for dimension in dimensions: print(dimension) Python returns all the elements in the tuple as it would for the list:

400

100

Writing over a Tuple

Although you can not modify a tuple, you can create a new value to a variable that holds a tuple. And if we had to change our proportions, we might redefine the entire tuple:

u dimensions = (400, 100) print("Original dimensions:") for dimension in dimensions: print(dimension) v dimensions = (800, 200) w print("\nModified dimensions:") for dimension in dimensions: print(dimension)

The u block describes the original tuple and displays the initial dimensions. At v, a new tuple is placed in the unit dimensions. Then we are going to print the new dimensions at w. Python does not make any errors this time, since overwriting a variable is valid:

Original dimensions:

400

100

Modified dimensions:

800

200

When compared to lists, tuples are easy data constructions. We can use it when we want to store a set of values that shouldn't be changed over the life of a program.

Indentation

PEP 8 recommends using four spaces per indentation level. Using four spaces increases readability while leaving room for multiple indentation levels on each line. In a word processing document, people frequently use tabs instead of indent spaces. This works fine with word processing documents, but the Python interpreter gets confused when tabs are mixed with spaces. Each text editor provides a setting that allows you to use the tab key but then converts each tab to a set number of spaces. You should certainly use your tab key, but also make sure that your editor is set to insert spaces instead of tabs into your document. Mixing tabs and spaces in your file may cause problems that are very difficult to diagnose. If you feel you have a mix of tabs and spaces, you can convert all tabs in a file into spaces in most editors.

Line Length

Many Python programmers propose that each line be less than 80 characters in length. Historically, this guideline was developed because most computers could accommodate only 79 characters on a single line in the terminal window. At present, people can accommodate much longer lines on their computers, but there are many incentives to stick to the regular length of the 79-character grid. Professional programmers often have multiple files open on the same screen, and using the standard line length, they can see whole lines in two or three files that are opened side by side on the screen. PEP 8 also suggests that you limit all of your comments to 72 characters per line, as some of the tools that generate automatic documentation for larger projects add formatting characters at the beginning of each commented line. The PEP 8 line length guidelines are not set in stone, and some teams prefer a 99-character limit. Do not worry too much about the length of the line in your code as you learn, but be aware that people who work collaboratively almost always follow the PEP 8

guidelines. Many of the editors allow you to set up a visual cue, usually a vertical line on your screen, which shows where these limits are if Statements Programming often involves examining a set of conditions and deciding which action to take on the basis of those conditions. Python's if the statement allows you to examine the current state of the program and respond appropriately to that state of affairs.

In this section, you will learn how to write conditional tests, which will allow you to check any conditions of interest. You will learn to write simply if statements, and you will learn how to create a more complex series of if statements to identify when the exact conditions you want are present. You will then apply this concept to collections, so you can write a loop that handles most items in a list one way, then handles other items with specific values in a different way.

A Simple Example

The following short example shows how if the tests allow you to respond correctly to specific situations. Imagine that you have a list of cars and that you want to print out the name of each vehicle. Car titles are the right ones, so the names of most vehicles should be written in the title case. But, the value 'BMW' should be printed in all cases. The following code loops through the car list

Names and looks for the 'BMW' value. Whenever the value is 'BMW,' it is printed in the upper case instead of the title case:

vehicles.py vehicles = ['audi', 'bmw', 'subaru', 'toyota'] for vehicle in vehicles: u if car == 'bmw': print(vehicle.upper()) else: print(vehicle.title())

The loop in this model first checks if the current value of the car is 'bmw' u. If it is, the element is printed in uppercase. If the value of the vehicle is other than 'bmw', it is printed in title case:

Audi

BMW

Subaru

Toyota

Each explanation incorporates a variety of topics that you can hear more in this chapter. Let us continue by looking at the types of measures you

might use to analyze the conditions in your system.

Conditional Tests

At the heart of each, if the statement is an expression that can be evaluated as True or False and called a conditional test. Python practices the True and False values to decide whether the code in the if statement should be executed. If the conditional check is valid, Python must run the code following the if argument. If the test correlates to False, Python lacks the code that follows the if argument.

Checking for Equality

Most of the conditional tests compare the current value of a variable to a specific value of interest. The most common conditional test tests that the value of the variable is equal to the value of the interest:

u >>> vehicle = 'bmw' v >>> vehicle == 'bmw' True

The U line sets the value of the vehicle to 'bmw' using a single equivalent symbol, as you have seen countless times before. The line in v tests if the name of the vehicle is 'bmw' using a double equal sign (= =). This equivalent operator returns True if the values on the left and right sides of the operator match, and False if they do not match. The values in this example will suit, so Python will return Real. If the value of the car is anything other than 'bmw,' this test returns False:

u >>> vehicle = 'audi' v >>> vehicle == 'bmw' False

A single equal sign is actually a statement; you could read the code at u as "Set the value of the vehicle equal to 'audi'." While a double equal sign, like the one at v, inquires a question: "Is the value of the vehicle equal to 'bmw?' "Most programming languages use the same sign in this way.

Ignoring Case When Checking for Equality

Testing for equality is a sensitive case in Python. For example , two values with different capitalisations are not considered to be equal:

>>> vehicle = 'Audi' >>> vehicle == 'audi' False

This conduct is beneficial if the situation matters. But if the case does not matter and instead you just want to test the value of the variable, you can

convert the value of the variable to the lowercase before you make the comparison:

>>> vehicle = 'Audi' >>> vehicle.lower() == 'audi' True

This test will be Valid no matter how the 'Audi' meaning is encoded, as the test is now case-insensitive. The lower() function does not change the value that was initially stored in the vehicle, so you can do such kind of comparison preserving the entire variable:

u >>> vehicle = 'Audi' v >>> vehicle.lower() == 'audi' True w >>> vehicle 'Audi'

U stores the capitalized string 'Audi' in the variable engine. At v, we convert the value of the vehicle to the lowercase and compare the lowercase value to the 'audi' series. The two strings are paired, so Python returns Real. At W, we see that the value kept in the vehicle was not affected by the condition.

Testing. Websites implement certain laws for data entered by users in a way similar to this. For example, a site may use a conditional test like this to ensure that each user has a truly unique username, not just a change in the capitalization of another username. When someone else is

Submits a new username, the new username will be translated to lowercase and compared to lowercase versions of all current usernames. During this check, a username such as 'John' will be rejected if any variation of 'John' is already in use.

Checking for Inequality

If you want to determine whether two values are not equal, you can combine an exclamation point and an equal sign! (=). The exclamation mark is not as it is in other programming languages. Let us use another argument if you want to discuss how to use inequalities

Director. Director. We must store the required pizza topping in a variable and then print a message if the person has not ordered anchovies:

toppings.py requested_topping = 'mushrooms' u if requested_topping != 'anchovies': print("Hold the anchovies!")

The line at u relates the value of requested topping to the value of 'anchovies.' If these two values are not balanced, Python returns True and implements the code given the if statement. If the two values match, Python comes back False and does not execute the code following the if statement. Since the requested topping value is not 'anchovies,' the print statement is executed: Keep on the anchovies! Most of the words that you write will test for equality; however, perhaps you will find it more effective to check for inequalities.

Numerical Comparisons

Checking numerical values is very easy. For instance , the given code checks whether a person is 20 years of age:

>>> age = 20 >>> age == 20 True

Also, You can check to see if two numbers are not the same. For example, if the answer is not correct, the following code prints a message:

magic_ answer = 19 number.py u if answer != 46: print("That is not the correct answer. Please try again!")

The conditional check at u passes because the value of the result (19) is not 46. The indented code block is executed because the test passes:

That is not the correct answer. Please try again!

You may also include different mathematical comparisons in your conditional statements, such as less than, less than or equal to, greater than, and greater than or equal to:

>>> age = 19 >>> age < 21 True

>>> age <= 21 True

>>> age > 21 False

>>> age >= 21 False

Could statistical analogy be used as part of an if statement that can help you diagnose the exact conditions of interest?

Checking Multiple Conditions

You may want to test different conditions at the same time. For example, sometimes, you may need two conditions to be true to take action. Other times, you might be satisfied with only one condition being True. Keywords and or can help you in these situations.

Using and to Check Multiple Conditions

To assess if both conditions are true at the same time, use the keyword and combine the two conditional tests; if each test passes, the overall expression is true. If either the test fails or all tests fail, the expression will be tested as False. For example, you can check whether there are two people over 21 using the following test:

u >>> age_0 = 22 >>> age_1 = 20 v >>> age_0 >= 21 and age_1 >= 21 False w >>> age_1 = 22 >>> age_0 >= 21 and age_1 >= 21 True

At u we describe two ages, age 0 and age 1. At v, we check whether the two ages are 21 or not. The test on the left passes, however, the test on the right fails, so False evaluates the overall condition. We are going to change the age 1 to 22. The value of age 1 is now bigger than 21, and all individual measures pass, allowing the final state expression to be measured as Valid.

You may use parentheses around the individual tests to enhance readability, but they are not necessary. If you were using parentheses, the exam should look like this:

(age_0 >= 21) and (age_1 >= 21)

Using or to Check Multiple Conditions

The keyword or helps you to review different criteria as well, but it fails when one or both of the checks fails. An object or function can only fail if all separate measures fail.

Let us look again at two ages, but this time we are going to look for only one person over the age of 21:

u >>> age_0 = 22 >>> age_1 = 10 v >>> age_0 >= 21 or age_1 >= 21 True

w >>> age_0 = 20 >>> age_0 >= 21 or age_1 >= 21 False

We start at u again with two age variables. If the age 0 check in v passes, the overall expression value is Valid. We are going to lower the age of 0 to

10. In the test at w, both tests have now failed, and the overall expression is evaluated for False.

DAY 7

If you understand the conditional tests, you can start writing the statements. Several different types of if statements exist, and the choice of one to use depends on the number of criteria you choose to check. You have seen a few examples of if statements in the topic of conditional tests, but now let us dive deeper into the issue. The simplest kind of argument that has one test and one action. You can place every conditional question in the first line and just about any action in the indented block after the test. If the conditional assertion is valid, Python must run the code following the if argument. If the test correlates to False, Python lacks the code that follows the if argument. Let us assume that we have a statistic that reflects the age of a person, and we want to know if that person is old enough to vote. The following code checks whether a person can vote:

voting.py age = 21 u if age >= 20: v print("You are old enough to vote!")

U Python checks whether the age value is greater than or equal to 18. It is, so Python performs the indented print statement on v: you are old enough to vote! Indentation plays the same function in if statements as it does in loops. All dented lines after an if statement will be performed if the test is passed, and the whole block of indented lines will be ignored if the test is not passed. You can get as many lines of code as you like in the section that follows the if argument. Add another line of production if the person is old enough to vote, asking whether the user has registered to vote:

age = 21 if age >= 20: print("You are old enough to vote!") print("Have you registered to vote yet?")

Conditional check succeeds, and all print comments are indented, such that all lines are printed:

You are old enough to vote!

Have you registered to vote yet?

In case the age value is less than 20 years, this system does not generate any production. If-else Statements Often, you are going to want to take one

action when the conditional test passes, and you are going to take another action in all other cases. The if-else syntax of Python makes this possible. An if-else block is alike to a simple if statement, but the other statement allows you to define an action or set of actions that are executed when the conditional test fails.

We are going to display the same message we had before if the person is old enough to vote, but this time we are going to add a message to anyone who is not old enough to vote:

age = 19 u if age >= 20: print("You are old enough to vote!") print("Have you registered to vote yet?") v else: print("Sorry, you are too young to vote.") print("Please register to vote as soon as you turn 20!")

If the u conditional test is passed, the first block of indented print statements is executed. If the test evaluates to False, the next block on v is executed. Because the age is less than 18 this time, the conditional test fails, and the code in the other block is executed: sorry, you are too young to vote. Please register for the ballot as soon as you turn 20! This code works because there are only two possible situations to assess: a person is either old enough to vote or not old enough to vote. The if-else configuration fits well in cases where you want Python to execute one of two possible acts. In a easy if-else chain like this, one of the actions is always executed.

The if-elif-else Chain

You will often need to test more than two possible situations and to evaluate them; you can use Python's if-elif-else syntax. Python executes only one block of the if-elif-else sequence. It will run each conditional check in order for one to pass. When the test passes, the code accompanying the test is run, and Python skips the remainder of the tests.

Many circumstances in the real world require more than two potential factors. Consider, for example, an amusement park that charges diverse rates for different age of people:

Admission for anyone under age 5 is free.

Admission for anyone between the ages of 5 and 20 is $5.

Admission for anyone age 20 or older is $10.

How do we use an if statement to decide the admission rate of a person? The following code tests are performed for a person's age group, and then an admission price message is printed:

amusement_ age = 12 park.py u if age < 5: print("Your admission cost is $0.") if Statements 85 v elif age < 20: print("Your admission cost is $5.") w else: print("Your admission cost is $10.")

If the test at u measures whether a person is under 4 years of age. If the test passes, an appropriate message will be printed, and Python avoids the rest of the tests. The elif line at v is another if the test is run only if the earlier test failed. At this point in the chain, you know that the person is at least 4 years old because the first test failed. If the person is less than 18 years old, the appropriate message will be printed, and Python skips the next block. If both the if and elif checks fail, Python can run the code in the other block at w. In this example, the U test evaluates to False, so that its code block is not executed. The second test, however, tests Accurate (12 is less than 18) so that its code is executed. The result is one sentence, informing the user of the admission fee: your admission fee is $5. Any age greater than 17 would have caused the first two tests to fail. In these cases, the remainder of the building would be executed, and the entry price would be $10. Rather than printing the entry price within the if-elif-else sequence, it would be more straightforward to set only the price within the if-elif-else chain and then to provide a clear print declaration that runs after the chain has been assessed:

age = 12 if age < 5: u price = 0

elif age < 20: v price = 5 else: w price = 10

x print("Your admission cost is $" + str(price) + ".")

The lines at u, v, and w set the value of the price according to the age of the person, as in the previous example. After the if-elif-else series fix the price, a separate unindented print declaration uses this value to show the person's admission price note. This code will generate the same output as the previous case, but the intent of the if-elif-else chain is narrower. Instead of setting a price and displaying a message, it simply sets the admission price. This revised code is simpler to change than the original approach. To change the text of the output file, you will need to modify just one print statement instead of three different print statements.

Using Multiple elif Blocks

We can use as many elif blocks in our code as we want. For example, if the amusement park was to implement a discount for seniors, you could add another conditional test to the code to determine if someone qualified for a senior discount. Let us assume that someone 65 or older charges half of the normal fee, or $5:

age = 12 if age < 5: price = 0

elif age < 20: price = 5 u elif age < 65: price = 10

v else: price = 5 print("Your admission cost is $" + str(price) + ".")

Any of this code remains unchanged. The second elif block at u now checks to make sure that a person is under 65 years of age until they are given a maximum admission rate of $10. Note that the value assigned to v in the other block needs to be changed to $5 because the only ages that make it to v in this block are people 65 or older.

Omitting the else Block

Python does not require another block at the end of the if-elif chain. Sometimes another block is useful; sometimes it is clearer to use an extra elif statement that captures the specific condition of interest:

age = 12 if age < 5: price = 0

elif age < 20: price = 5

elif age < 65: price = 10

u elif age >= 65: price = 5

print("Your admission cost is $" + str(price) + ".")

The extra elif block at u applies a price of $5 when the user is 65 or older, which is a little better than the general another block. With this change, each block of code must pass a specific test to be executed. The other section is the catchall argument. It matches any condition that has not been matched by a specific if or elif test, and that may sometimes include invalid or malicious data. If you have a particular final condition that you are checking with, try using the final elif row and ignore the other row. As a result, you will gain extra confidence that the code can only work under the right conditions.

Testing Multiple Conditions

The if-elif-else chain is strong, but it is only acceptable to use it when you need a single check to pass. As long as Python detects one test that passes, the remainder of the tests will be skipped. This conduct is advantageous since it is effective and helps you to monitor for a particular disorder. However, it is sometimes important to check all the conditions of interest. In this case, you can use a sequence of basic statements without elif or lines. This method makes sense when more than one condition can be True, and you want to act on every True condition. Let us take a look at the burger example. If someone asks for a two-topping burgers, you will need to be sure to comprise both toppings on their burger:

toppings.py u requested_toppings = [coconut, 'extra cream']

v if 'coconut' in requested_toppings: print("Adding coconut.")

w if ' sausage ' in requested_toppings: print("Adding sausage .")

x if 'extra cream' in requested_toppings: print("Adding extra cream.")

print("\nFinished making your burger!")

We start with a list of the requested toppings. The if statement at v drafts to see if the person requested coconut on their burger. If this is the case, a message confirming that topping is printed. The sausage test at w is a clear one if the argument, not the elif or the result, and this test is performed regardless of whether the previous test has passed or not. The x code checks if additional cheese has been ordered, irrespective of the outcome of the first two measures. These three independent tests are performed every time the program is running. Because each condition in this example is assessed, both coconut and extra cream are added to the burger:

Adding coconut.

Adding extra cream.

Finished making your burger!

This system would not work correctly if we were to use the if-elif-else function, as the system would stop running if just one test passes. Here's what it should feel like:

requested_toppings = ['coconut ', 'extra cream'] if 'coconut' in requested_toppings:

print("Adding coconut.") elif 'sausage' in requested_toppings:

print("Adding sausage.") elif 'extra cream in requested_toppings:

print("Adding extra cream.") print("\nFinished making your burger!")

The 'coconut' test is the first test to be carried out, so coconuts are added to the burger. But, the values 'extra cream' and 'sausage' are never tested, since Python does not run any tests after the first test that passes along the if-elif-else series. The first topping of the customer will be added, but all of their other toppings will be missed:

Adding coconuts.

Finished making your burger!

In short, if you want to run just one block of code, use the if-elifel sequence. In case more than 1 block of code needs to be run, use a set of independent if statements.

A Simple Dictionary

Consider a game featuring aliens that may have different colors and point values. This basic dictionary stores details about an alien:

alien.py alien_0 = {'color': 'red', 'points': 5}

print(alien_0['colour']) print(alien_0['points'])

The alien 0 dictionary stores the color and meaning of the alien. The two print statements access and display the information as shown here:

red 3

Like most new programming concepts, dictionaries are used to practice. Once you have worked with dictionaries for a bit, you will soon see how effectively real-world situations can be modeled.

Working with Dictionaries

The Python dictionary is a list of key-value pairs. -- the key is connected to a value, and a key may be used to access the value associated with that key. The value of a key can be a number, a string, a list, or even a different

dictionary. In addition, any object you can construct in Python can be used as a value in a dictionary. In Python, the dictionary is wrapped in bracelets,}, {with a sequence of key-value pairs within bracelets, as seen in the previous example:

alien_0 = {'colour': 'red', 'points': 3}

A key-value duo is a set of values that are connected. When you enter a key, Python returns the value associated with that key. Through key is related to its value by a colon, while commas separate the individual key-value pairs. You can save as many key-value pairs as you like in a dictionary. The easiest dictionary has exactly one key-value pair, as shown in the modified version of the alien_0_dictionary:

alien_0 = {'colour': 'red'}

This dictionary stores one piece of info about alien 0, the color of the alien. The 'colour' string is the key in this dictionary, and its related meaning is 'red.'

Accessing Values in a Dictionary

To obtain the value connected with the key, enter the name of the dictionary and then place the key inside the square bracket set, as shown here:

alien_0 = {'color': 'red'} print(alien_0['colour'])

This reverts the value connected with the key 'colour' from the dictionary alien_0:

red

You can have an infinite amount of key-value pairs in your dictionary. For example, here is the original alien 0 dictionary with two key-value pairs:

alien_0 = {'colour': 'red', 'points': 3}

You can now access either the color or the point value of alien 0. If a player shoots this alien down, you can see how many points they are supposed to earn using code like this:

alien_0 = {'colour': 'red', 'points': 3} u new_points = alien_0['points']

v print("You just got " + str(new_points) + " points!")

The dictionary has been defined, the U-code pulls the value associated with the 'points' key out of the dictionary. This value is then stored in the new point variable. The v line transforms this integer value to a string and prints a declaration of how many points the player has just earned:

You just earned 3 points!

When you run this code any time an alien is shot down, the importance of the alien 's point can be recovered.

Adding New Key-Value Pairs

The dictionaries are dynamic structures, and you can add new key-value pairs to your dictionary at any time. For instance, to add a new key-value pair, you will be given the name of the dictionary, followed by a new key in square brackets along with a new value. Add two new pieces of data to the alien_0 dictionary: the x-and y-coordinates of the alien, which will help us to display the alien in a particular position on the screen. Position the alien on the left edge of the screen, 25 pixels down from the top. Since the screen coordinates normally start at the top left corner of the screen, we can position the alien at the left edge of the screen by setting the x-coordinate to 0 and 25 pixels from the top by setting the y-coordinate to positive 25, as seen here:

alien_0 = {'colour': 'red', 'points': 3} print(alien_0)

u alien_0['x_position'] = 0 v alien_0['y_position'] = 15 print(alien_0)

We define the same dictionary that we worked with. Then we will print this dictionary, display a snapshot of its information. U adds a new key-value pair to the dictionary: key 'x position' and value 0. We do the same for the 'y position' key in v. When we print the revised dictionary, we see 2 additional key-value pairs:

{'colour': 'red', 'points': 3}

{'colour': 'red', 'points': 3, 'y_position': 15, 'x_position': 0}

The final version of the dictionary consists of four key-value pairs. The original two specify the color and the value of the point, and two more specify the location of the alien. Note that the order of the key-value pairs does not suit the order in which they were inserted. Python doesn’t care

about the rhythm in which you place each key-value pair; it just cares about the relationship between each key and its value.

Starting with an Empty Dictionary

In most cases, it is useful, or even essential, to start with an empty dictionary and then add each new element to it. To start filling a blank dictionary, define a dictionary with an empty set of braces, and then apply each key-value pair to its own line. For example, below is how to construct the alien 0 dictionaries using the following approach:

alien_0 = {} alien_0['colour'] = 'red'

alien_0['points'] = 5 print(alien_0)

We define a blank alien_0 dictionary, and then add colour and value to it. The result is the dictionary that we used in previous examples:

{'colour': 'red', 'points': 3}

Typically, empty dictionaries are used when storing user-supplied data in a dictionary or when writing code that automatically generates a large number of key-value pairs.

Modifying Values in a Dictionary

To change the value in the dictionary, enter the name of the dictionary with the key in square brackets, and then the new value you want to associate with that key. Consider, for example, an alien who changes from green to yellow as the game progresses:

alien_0 = {'colour': 'red'} print("The alien is " + alien_0['colour'] + ".")

alien_0['colour'] = 'yellow' print("The alien is now " + alien_0['colour'] + ".")

First, we describe a dictionary for alien 0 that includes only the color of the alien; then, we change the meaning associated with the 'colour' key to 'black.' The performance reveals that the alien actually shifted from green to yellow:

The alien is red.

The alien is now yellow.

For a more interesting example, let us take a look at the position of an alien who can move at different speeds. We will store a value that represents the current speed of the alien and then use it to determine how far the alien should move to the right:

```
alien_0 = {'x_position': 0, 'y_position': 15, 'speed': 'medium'}
print("Original x-position: " + str(alien_0['x_position']))
# Change the alien to your right.
#Identify how far to move the alien based on its current speed.
u if alien_0['speed'] == 'slow': x_increment = 1
elif alien_0['speed'] == 'medium': x_increment = 2 else:
# This must be a fast alien. x_increment = 3
# The fresh position is the previous position plus the increment.
v alien_0['x_position'] = alien_0['x_position'] + x_increment
print("New x-position: " + str(alien_0['x_position']))
```

We begin by defining an alien with an initial position of x and y and a speed of 'medium.' We have omitted color and point values for simplicity, but this example would work the same way when you include those key-value pairs as well. We also print the real value of x position to see how far the alien is moving to the right. At u, the if-elif-else string determines how far the alien should move to the right and stores this value in the x increment variable. If the speed of the alien is 'slow,' it moves one unit to the right; if the speed is 'medium,' it moves two units to the right; and if it is 'fast,' it moves three units to the right. If the calculation has been calculated, the value of x position is added to v, and the sum is stored in the x position dictionary. Since this is a medium-speed alien, its position shifts two units to the right:

Original x-position: 0 New x-position: 2

This approach is pretty cool: by modifying one meaning in the alien's vocabulary, you can alter the alien 's overall actions. For example, to

transform this medium-speed alien into a fast alien, you should add the following line:

```
alien_0['speed'] = fast
```

The if-elif-else block will then add greater value to x increment the next time the code is running.

CONCLUSION

Python is one of the several open-source, object-oriented programming applications available on the market. Some of the other uses of Python are application development, the introduction of the automated testing process, multiple programming build, fully developed programming library, all major operating systems, and platforms, database system usability, quick and readable code, easy to add to complicated software development processes, test-driven software application support.

Python is a programming language that assists you to work easily and implement your programs more effectively. Python is a versatile programming language used in a wide variety of application domains. Python is also compared with Perl, Ruby, or Java. Some of the main features are as follows:

Python enthusiasts use the term "batteries included" to describe the main library, which includes anything from asynchronous processing to zip files. The language itself is a versatile engine that can manage nearly every issue area. Create your own web server with three lines of javascript. Create modular data-driven code using Python 's efficient, dynamic introspection capabilities, and advanced language functionality such as meta-classes, duck typing, and decorators. Python lets you easily write the code you need. And, due to a highly optimized byte compiler and library support, Python code is running more than fast enough for most programs. Python also comes with full documentation, both embedded into the language and as separate web pages. Online tutorials are targeted at both the experienced programmer and the beginner. They are all built to make you successful quickly. The inclusion of an excellent book complements the learning kit.