Follows TIY
(Teach it Yourself)
Approach

FOUNDATION OF PYTHON

SECONDARY SCHOOL & COLLEGE

In Line with CBSE & in Tune with
Child's Aspirations
& their Technological Future

LIST OF CONTENT

Part 1 Python Basics

Part 2 Understanding Python Tools

Part 3 Using Python Tools

CHAPTER

01 Let us Get Started

1 Lesson Overview

This lesson shall cover, What is Python, Downloading of IDLE, Our First few Codes & an introduction to a few key terms to get you started with more ahead.

2 Layout of the Book

The book has been divided into three parts:

Part 1. Python basics. (introduction, cmd prompt, rules and converntions).

Part 2. Understanding python tools (var, conditionals, loops, data types, operators and functions).

Part 3. using python tools (sequences, strings, lists, tuples, dict, sets and errors).

3 What is Python

Python is an **interpreted, object-oriented, high-level programming language,** with **dynamic semantics.** Very confusing. Let us make it simpler.

Look at a **one-line code:**

```
>>> activity = 'We are learning Python from BDS Education'
```

- **Interpreted** means, once the code is written, Python stores & executes its green parts, without first compiling them into machine language.
- Whenever called, it returns the green pars for conversion to machine language. These green parts are stored as Objects. This makes it an **Object-oriented** language.
- Coding is a means of communication between humans who understand English & machines that understand Binary. Closer our code is to English, Higher its Level. In Python we code in English (green parts). This makes it a **High-Level Programming Language.**
- **Dynamic Semantics** is its internal framework of logic & words, that is very English like.

Isn't it now simple.

4 Programming Environments

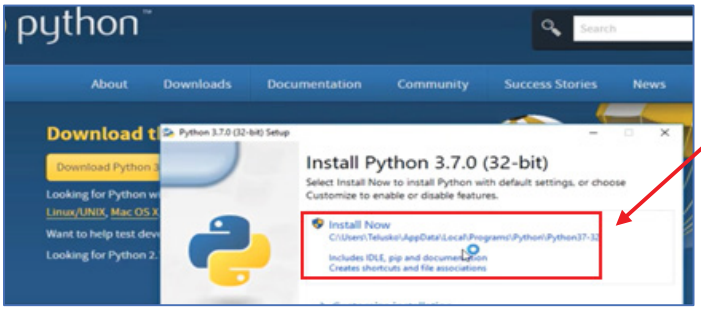
- Python, offers two download options to code:
  - IDLE (Integrated Development & Learning Environment). It offers line by line coding & is designed for learning the use of Python coding tools.
  - IDE (Integrated Development Environment). These are open-source platforms inspired by Python & developed by different companies for professional coding.
- We shall start with an IDLE, & later Migration from IDLE to IDE.

5 Downloading IDLE

To learn Python, the first step is to download Python IDLE.
To do this:

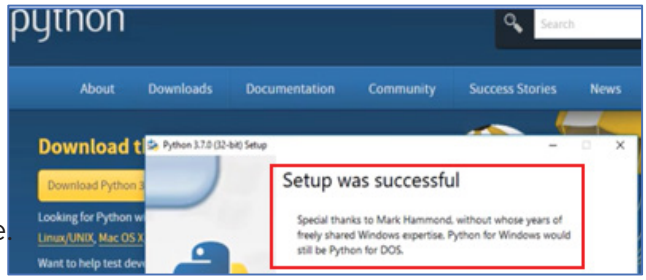
- Google **Python download.**

- Click on **Download Python.**
- Select **Download 3.7.0 or the latest version that appears.**



Select **install Now.** Installing will start but will take some time.

At the end, **Set Up Successful** window appears. Click on **Close.** Download is complete.



To use: Go to bottom left of menu bar. Click search icon.

- Type **Python** against search icon.

Window on the right opens.
In search result, select IDLE.

- Python **IDLE Shell** appears.

Most important part of the shell is this Symbol >>>
It indicates start of a line. Entire Python coding is done against this. It is called **Cmd (command) Prompt.**

Note: Shell background is selectable between **white or black.** We have chosen white.

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER 02

Command Prompt, Syntax & Comments

1 Lesson Overview

In has lesson we shall examine its five **interlinked** ingredients:

- **Story line.** Statements that summarise what we want to code & its outline plan.
- **Command Prompt.** Where & how to enter the code.
- **Running options.** Internal functioning of our interface with the machine – cmd prompt.
- **Python Syntax.** Machine grammar that ensures code is readable & processable.
- **Comments.** Notes that record thinking process to help other coders understand.

Story is a **written or mental declaration** of what we **want to code**, in what **sequence,** & the **result** we desire to achieve. Ex make an app for a shopping list reminder. Its storyline will be:

> Shopping app should enable me to make a list containing item name & qty. It should also allow me to recall the list & make a new list after use.

Its simple code is:

```
>>> shoppingList1 = ['Apples', 12, 'Surf', 3, 'Oil', 1]
>>> print(shoppingList1)
['Apples', 12, 'Surf', 3, 'Oil', 1]
>>> shoppinglist2 = {'Milk: 2', 'Butter: 4'}
```

- 1st line captures data defining the **code.**
- 2nd defines the cmd on which it will be **executed** & method for **return** of output.
- 3rd captures data to make a new list after use of the first & so on.

There are no formal rules for writing a story. It is a Tool to streamline thinking process. It is part of Pythons **Good Practices.** It breaks task into smaller, logical & manageable chunks.

2 Understanding Command Prompt

In Python IDLE, code moves one **line at a time.** Each line of code once entered & accepted is **error free.** In this ex, Name in capital & small are two separate names. We defined it with small g (line 1), & were calling in capital G (Line 2). It thus returned an error. Once removed (line 3), code is accepted, & prompt moves ahead for next cmd.

```
>>> greet = 'Hello World'
>>> print(Greet)
Traceback (most recent call last):
>>> print(greet)
Hello World
>>>
```

3 Running Command Prompt

It means Executing the Code or sending a **return.** Basic ways of return for **display** (print) are:

```
>>> fruit = 'Apple'
>>> print(fruit)
Apple
>>> fruit
'Apple'
```

- **print(fruit)** or just **fruit** (first box().
- **print(x)** or just **x** (second box).

Note: 1. Commands like print are case sensitive. Use of P in capitals will give error. Try it.

```
>>> x = 22
>>> print(x)
22
>>> x
22
```

At cmd prompt the **last value** of any input or variable prevails (44). (first box)Once changed, the **changed value** prevails (66). Previous value is overwritten.

```
>>> x = 44
>>> print(x)
44
```

```
>>> x = 66
>>> print(x)
66
```

4 Options for Running the Command Prompt

We have two options:

- **Interactive Interpreter mode.**
- **Script file mode.**

Option 1 - Interactive Interpreter Prompt Mode

It is the preferred mode of use by **Beginners,** for learning the basics line by line. It is also used by **Programmers,** when they are concerned about the output of each line.
This is called **interactive** because it is executed through interaction with the interpreter:

- Once the object is defined **(x = 4+5)**, the **data is saved** (1st line).
- Thereafter, the shell waits for action **cmd** print**(X)** from the user.

```
>>> x = (4 + 5)
>>> print(x)
9
```

This cmd can be given soon after def the var (as in this case) or at time of its use later.

Interactive mode is best to run single-line statements of a code. Its disadvantages are:

- It is not suitable to write long codes having multiple lines.
- It is not suited for off line coding which is very frequently the case with professionals.

Option 2 - Script File or Script Mode

Using **script mode,** we can write codes **off line.** We can write multiple lines of code into a file which can be saved & **executed later.** For this, we need to open an editor like notepad, create a file, name it & save it with .**py** extension (**py** stands for Python). **In lesson on migration to IDE, in the next volume we shall be introduced to this.**

Advantages of Script Mode. It has a strong prompting mechanism to assist data entry. Its debugging is easy & is the preferred mode for professional coding. Python IDE is based on this mode.
Disadvantages of Script Mode. We have to save the code every time we make a change. This procedure can be tedious when as beginners we need to run single or few lines of code.

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

03 Rules & Conventions

1 Lesson Overview

In has lesson we shall understand the conventions & rules for use of Colour, Names, Parentheses,White Space & Indentation.

2 Python Rules & Conventions

Machines only have a binary logic. **A logic of yes or no, true or false, go or no go, zero or one.** Instructions given to them have to be **PERFECT.** This is possible if stringent rules & conventions are laid down & followed.

Python rules & conventions include conventions for comments, naming, code layout, use of whitespaces, indentations & separations. They can broadly be divided into two types:

- Rules & conventions for **style.**
- Rules & conventions for **naming.**

We have covered the rules for comments. Let us now see the others.

PEP (Python Enhancement Proposal) has several proposals written in 2001 by Guido van Rossum, Barry Warsaw, & Nick Coghlan. **PEP 8**, is a document that provides guidelines & best practices on writing Python code. These are contained in a chapter – **Style guide for python code.** https://www.python.org/dev/peps/pep-0008/

3 Colour Conventions

As a gen rule, Python objects like names, nums, operators etc are in black. Return commands in purple. String inputs in green. Outputs & function names in blue.

```
>>> My_list = ['Apples', 10, 'Oranges',12]
>>> print(My_list)
['Apples', 10, 'Oranges', 12]
```

Func def, loop statements, logic elements etc in light red.

```
>>> def myfunction():
```

Comments in red or green. These give one look access to programmers to different segments, help debugging & reduce input entry errors.

4 Naming Conventions

While coding, we have to name things like Variables (Var), Function (func) etc. A sensible name tells what a var, func or class represents. They reduce errors & help debugging. **Recommended Conventions** are:

Type	Convention	Example
Functions	Use lower case words. Separate using underscore	my_func, greet
Variables	Use lower case letter, words or words. Separate using underscore	x, fruits, my_var
Constants	Use upper case letter, word or words. Separate using underscore	Y, CONSTANT
Class	Start word or words with capital letter. No underscore. Camel style.	MyVehicles, Phase1
Methods	Use lowercase word or words. Separate using underscore	my_methods
Modules	Use short word or words in lower case. Separate using underscore	module.py
Package	Use short word or words in lower case. No underscore	mypack

4.1. Naming Var – What is Permitted?

- **Rule-1:** Var name with alphabet or underscore (_) character.
- **Rule-2:** Var name can contain A to Z, a to z, 0 to 9 & underscore (_).
- **Rule-3:** Var names are case sensitive. Ex – My & my, are different var.

```
>>> x = 44
>>> School = 'Bal Bharati'
>>> mySchool = 'Bal Bharati'
>>> my_school = 'Bal Bharati'
>>> _My_School = 'Bal Bharati'
```

4.2. What is Not Permitted?

- **Rule-1:** You cannot start var name with a num or - .
- **Rule-2:** You cannot use special ch such as $,%,#,&,@ etc.
- **Rule-3:** You cannot use a keyword as a var name. In ex 4, global is a keyword.

```
>>> 8_Branch = 'Sarojni Nagar'
SyntaxError: invalid decimal literal
>>> -school = 'Bal Bharti'
SyntaxError: cannot assign to operator
>>> branch@ = 'Sarojni Nagar'
SyntaxError: invalid syntax
>>> global = 'Sectoe 4, Noida'
SyntaxError: invalid syntax
```

5 Use of brackets

These are of three types. They are reserved for:

- **Round brackets ()** for callable functions & classes or for creating tuples.
- **Square brackets []** for requesting individual items (ex index value) & creating lists.
- **Curley brackets {}** for creating dictionaries & sets.

6 White Spaces

Whitespace is a method of **creating spaces around characters** & lines in code using space, tab & CRLF (carriage returns & line feeds). These allow us to **format our code** in a way that **makes them readable** in a single glance. In most cases **Python provides for default** white spacing. In case like assigning values, these have to be provided by us.

6.1. Whitespace Around Operators.

- Provide a **single space** around assignment operators, `>>> x = 22` comparison operators `>>> print(x == y)` & boolean operators. `>>> x>4 and y>4`
- When more than one operator is involved, add space only after operator **with lowest priority.**

```
>>> y = x**2 + 22
>>> z = (x+y) - (x-y)
```

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

04 Variables & Constants

1 Lesson Overview

In has lesson we shall understand **What are** variables (vars). **Making** vars. **Naming** Vars. **Assigning** values to Vars. Understanding **data types** in vars. **Local & Global** vars. Making of **Constants.**

2 What are Variables

In python Variables (Var) are a means to **store data values.**
Think of a var as an Empty box. It becomes usable, ONLY when we put **something inside it** or **assign it a value.**
For ex, In our real life an empty box becomes:

- A **jewellery box** if we put jewellery inside it.
- A **masala box,** if we put masala inside it.
- Or a **python variable** if we put data value **(age = 22)** inside it.



3 Making a Variable

A var is made in three steps:

- **Step 1.** Process of giving a **Name (age)** is called **Declaring a variable.**
- **Step 2.** Process of giving name to a **Value (16)** is called **Assigning a var.**
- **Step 3.** Process of assigning value for the **first time** is called **Initialising the Var.**

```
>>> age
```

```
>>> age = 16
```

3.1. Storing a Variable. Python automatically stores it for us, at a fixed location in its memory. It remembers this location & will return it to us when required. To know this location, use function **id()** & put the name inside the round bracket. A 13-digit num is printed.
Term var is **symbolic & temporary.**
De-facto var is a name or a **label** allotted to a memory loc for storing the var as an **Object.**

```
>>> age = 16
>>> id(age)
1863593388816
```

3.2. Calling a Variable. Whenever, we need this Var or Object, it must be **Called.** It has two choices for the **value it can return:**
Straight forward return for actions like printing.
Return after taking cognisance of additional, or new data, or instruction given at the time of call **(age of student is:).**

```
>>> age = 22
>>> print(age)
22
```

```
>>> print('Age of student is:', age)
Age of student is: 22
```

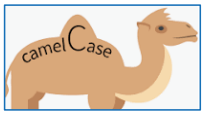
Green text inside single or double quotes **(called string)** is the data added at the time of call. The string can have any content, & is changeable. This is an **important concept.**

3.3. Changing a Variable Value. Variable are **immutable.** This means, we **can not change** its value till its lifetime. When value of any var is changed, its life time commitment to that value (lines 1) is over. It becomes a **new variable** with same name (var), but new value (lines 4). Now since value is related to loc, if the value changes, its loc ID must also change (line 5 & 6).

```
>>> var = 'Python'
>>> id(var)
1786671305968
>>> var = 'Learning Python'
>>> id(var)
1784564703152
```

3.4. Good Naming Practices.

- Name & data contained in it, should have a logical link.
- Clarity is important. Do not loose it in an attempt to be concise.
- Camel Casing. Mix of small with Capitalisation.




```
>>> tempCelsius = 55
```

- Pothole_case_. Use lowercase words separated by one or more underscores.

```
>>> cus_preference = 'Movie Godfather'
```

- Snake case.

```
>>> num_pass_with_distinction = 4
```

- Pascal case. Also called double hump camel case.

```
>>> NumPassWithDistinction = 3
```

4 Assignment of Values to Variables

- **Simple Assignment.** Python syntax for making a variable is **name = Value.**
- **Lvalue & Rvalue.** In python = is called an **assignment operator**. It assigns values to the **operands** on its either side. Value assigned to LHS, is the Lvalue. Value assigned to RHS, is the Rvalue.Trying to assign the opposite, returns an error. We cannot assign:

```
>>> x = 22 # Ex of Lvalue & Rvalue
```

```
>>> 22 = x # Wrong ex that will return error.
SyntaxError: cannot assign to literal
```

4.1. Multiple Assignments. This is basically of two types:

- **Assigning same value to multiple var.** This ex has assigned value **22** to all var **x, y & z.** This is often done at the time of coding, as **default values** in cases where these values can be changed at that time of use.

```
>>> x = y = z = 22
>>> print(y)
22
>>> z
22
```

- **Assigning Multiple Values.** Python allows us to do multiple assignments in one assignment. Syntax is **var1, var2, varN = exp1, exp2, expN.** It has three cases:

```
>>> x, y, z = 11, 22, 33
>>> print(y)
```

  - **Case 1. Values of same data type.** In line 1 python evaluates **Expressions** on RHS & then assigns values to corresponding var on LHS **(11 assigned to x).** Desired var can now be accessed **(y = 22).**

```
>>> x, y, z = 11, 22, 33
>>> print(y)
```

  - **Case 2. Values of different data types.**

```
>>> x, y, z, = 'Apple', 22.2, True
>>> print(z)
True
```

  - **Case3. Assignment after performing an operation.**

```
>>> x, y = 5, 8
>>> add, multi = (x+y), (x*y)
>>> print(multi)
40
```

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

05 Decision Making Statements

1 Lesson Overview

In has lesson students will get a clear idea of the decision-making apparatus of Python. They shall learn about Iteration. Thereafter, they will learn about the IF statement, the if-else statement & the Elif statements. In addition, they will learn about Nested statements , one liners & short hand statements.

2 Decision Making In Python

Decisions must be taken when a code block has **conditional choices.** Example, while approaching a traffic light, if the light **(condition)** is red, we must stop. Humans do this by seeing & deciding. Autonomous cars would do it through code. Python provides decision-making statements that help machines create logical conditions for evaluation. We shall study four decision making tools of Python – **Iteration & Statements** in this lesson & **Loops & Loop control** in next.

3 Iteration

Iterate, Iterable, Iterator & Iteration. Iterate means going over something repeatedly. Ex in a library bookshelf containing search items **(books)** is the **iterable.** Finding a book in the bookshelf is result of iteration. Person who implements the search is the **Iterator.** Going to the library, doing the search & getting the book is process of **iteration.** In this example of python, **Apple** is the **object** or result of iteration. Data type **tuple** is the **iterable.** Syntax **(myit)** is the **iterator**, & the **entire process** is **iteration.**

```
>>> tuple = ('Apple', 'Orange')
>>> myit = iter(tuple)
>>> print(next(myit))
Apple
>>> print(next(myit))
Orange
```

How Iterations Works? To understand, see code lines along with line remarks:

```
>>> s = 'CAT'
>>> t=iter(s)
>>> next(t)
'C'
>>> next(t)
'A'
>>> next(t)
'T'
>>>
>>>
```

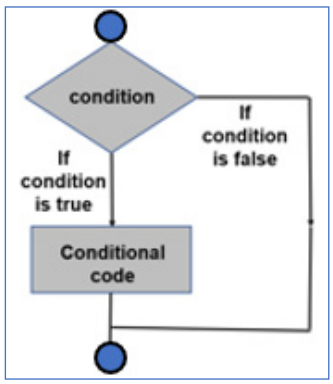
- **s = 'CAT'** is a string var. It is an **iterable.**
- t = **iter(s)** is an **iterator** using **iter()** to **iterate** the string var.
- **next()** points at C in CAT to **start** iteration. It returns value C & advances the state of iteration to letter A.
- Again **next()** returns A & advances state of iteration to T.
- Now since we are at the end of the iterable, **next**() returns T & raises **stop iteration** to signal that iteration is complete

Note: Iteration starts from top, goes till the bottom, prints the result of iteration & then goes back to iterate again, till all iterables have been iterated upon one by one.

4 Statements

Decision making comes natural to Humans. A month-old child soon understands **if** I cry **then** I get attention. Statements **if** & **then** are his decision-making apparatus. In case of machines, we have to figure this process & convert it into code so that machines can then take decisions on our behalf. Ex: A drone can be told that **if** battery level falls below a certain level, **then** it should return home.

This is the typical **Decision-making Structure**. To understand, let us take the example of traffic lights. Machines using binary (language of zero & one) assume, **IF** any condition (say light red) is **True**, it will be represented by a one, & will execute the conditional code (stop car) following the condition. On the other hand, **IF** it is **False,** it will by-pass condition to stop & keep moving. This is how machines take decisions.



Python has three decision-making statements - **IF** statement, **IF – ELSE** Statement, & **Elif** statement.

5 IF Statement

This box represents structure of IF Statement. In this, 1st line is a **header** that defines (def) the condition (expression). Lines 2 & 3 make the **body** that define its execution. Use of **(:)** colon at end of line 1 is mandatory. This is a typical example. It has a header of one condition (hungry) & a body of two actions. I will do nothing if condition is false (not hungry), But will eat if it is true. Python behaves the same way. This is the code for temperature control.

```
>>> if expression:
        statement(1)
        statement(n)
```

```
>>> if hungry:
        'Opem fridge'
        'Eat Pizza'
```

```
>>> temp = 20
>>> if temp < 15:
        print("Switch heater")
```

Automation industry works on simple decision-making logic.

5.1. Daily life ex using Boolean Var. In above example we used a nonBoolean var having a discreet value. Boolean var returns true or false. In this ex we are checking value of arrival (have guests arrived). If value is True, it means guests have arrived. In that case we print the three **statements** (red box) one by one.

```
>>> arrival = True
>>> if arrival==True:
        print('Welcome')
        print('To')
        print('India')
Welcome
To
India
```

5.2. Making a One Liner. In above example, condition was def as a two-line code. We could have done it in one line as shown in line 1 of this example. The code would run just fine. **Note this logic.**

```
>>> if arrival:
        print('Welcome')
        print('To')
        print('India')
```

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

06 Loops & Loop Control

1 Lesson Overview

In has lesson we shall understand the various types of loops and how they can be used in coding.
We shall also learn about the three-loop control statements & how they are used.

2 Terms Related to Loops

Loop is a coding mechanism, that repeats a block of code, until a condition is met. They are used to execute a code multiple times without having to write the same line of code repeatedly.

Example: `>>> for letters in password: print (letters)`
This is a bank code where a loop has been set to iterate three times. If the letters in password you enter, does not match the one it has in its data base, it reurn an error message or will lock you after three failed attempts.

2.1. Types of Loops. Python provides three types of loops:

- **For Loop.** Executes a **sequence of statements** multiple times.
- **While Loop.** Repeats a statement or group of statements **while a condition is TRUE.**
- **Nested Loop** is a derivative of the above two.

2.2 Range Function. This **Generates values** in a defined range. `>>> for i in range(1,7):`
In this, use of **range(1,7)** generates a list of values from index 1 to 7 [1,2,3,4,5,6]
where 1 is start value, 7 is up to value, & 1 is default Step Value. Same applies to While loop.
Note: Letter i is called **iteration Var.** Instead of using letter we can use any word (say temp) or alphabet (say d).

2.3. Changing Default Step Value. This needs a third parameter. **range(1:8:2)** where 1 is start value, 8 is up to value, & 2 is the new step value. This will generate [1,3,5,7].
Note : Negative step value gives a list in **reverse order.**

2.4. Meaning of Inside & Outside the Loop.

- **Inside the loop.** Loop starts after colon of line 2. Lines below it, in a prescribed ident are referred to as being inside the loop. Staying in is essential to add more statements between lines 2 & 3 in the same loop.

```
>>> n = 5
>>> for i in range(1,3):
        print(i)


1
2
>>>
```

- **Outside the loop.** Once code is written, to exit we must press enter twice after **print(i).** It removes indent & executes the cmd. Single click keeps us inside the loop. **Try This.**

3 For Loop

This is used for **sequential movements of statements in a code.** Syntax is:

```
>>> for iterator_var in sequence:
        statements(s)
```

Moment, we type **for** in the IDLE, python will indent itself for the coming **for** loop.

3.1 Working of for loop: inbox below, line 1 is var named **num.** The loop has three segments:

```
>>> num = [11,22,33,44]
>>> sum = 0
>>> for i in num:
        sum = sum + i

>>> print(sum)
110
```

- **Segment 1. Initialization (line 2).** We have taken a var named **sum** initialised with value 0 (since at this point sum is 0). Later when the loop runs, **nums** will be stored in it & then **summed** up and displayed.
- **Segment 2. Def condition for** iteration where i is the iterator. (condition is – iterate num, & get value till nums last, & pass on to line 2).
- **Segment 3. Iteration Tracking (line 4).** It tracks each iteration, & updates value of each iteration in var **sum** (line2). At end of iteration, output is returned when called (5th line).

3.2. Application of for Loop 2

Accessing Data Base. Say we have a data base named Score containing names of students & marks (line 2). We want to find the **marks** of a student named **Ram.** For this we create a var **searchName** (1st line) & use it as a condition in **if** branch (4th line).

```
>>> searchName = 'Ram'
>>> Score = {'Ram':95, 'Rohan':88}
>>> for student in Score:
        if student == searchName:
            print(Score[student])


95
```

In 3rd line, **for** loop Iterates & checks presence of a **student** as well as his **marks** in data base named **score.** It can now be used to find the **score of any student** (in the data base), by entering his name in line 1. Note the indenting. It happens automatically.

3.3. Application of for Loop - Accessing Data 2.

What if the **name does not exist?** How will the loop respond? Its coding is similar to the above except that here we **add an else branch** in the **for** loop. Adding this **dictates** the **action**, in case the name does not exist.

```
>>> searchName = 'Tom'
>>> Score = {'Ram':95, 'Rohan':88}
>>> for student  in Score:
        if student == searchName:
            print(marks[student])
else:
        print('Name not found.')
```

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

07 Operators – Basics & Types

1 Lesson Overview

In has lesson we shall understand the working of all types of operators other than logical operators. These are very different & will be learnt in the next lesson.

2 What are Operators

Operators are special symbols, keywords, or combinations of symbols that perform operations on values and variables.

```
>>> Operators = '+ - * / ** //etc'
```

They are used for everything from simple operations like counting to making complex security encryption algorithms. Operator carry out three basic operations:

- **First** is **action.** Its ex is (+) that represents an operation **(Add).**
- **Second** is **assignment.** Its ex is (=) that **Assigns** a value.
- **Second** is **comparison.** Its ex is **x>y** that **Compares** their values.

3 Types of Operators

Pythons 34 basic operators are divided into seven groups. In addition, we have a few miscellaneous or other operators. These are:

- **Arithmetic** operators (7 of them).
- **Comparison** or rational operators (6).
- **Assignment** operators (8).
- **Logical** operators (3).
- **Membership** operators (2).
- **Bitwise** operators (6).
- **Identity** operators (2).
- **Other Operators** (11).

4 Arithmetic Operators

They are most used, operation they are **a mathematical function that takes two operands & performs basic mathematical calculation (add, subs etc) on them.** We have seven arithmetic operators. These are – Addition, Subtraction, Multiplication, Division, Modulus, Exponentiation & Floor division.

4.1. Addition Operator (+). Adds the values on either side. Salient points are:

- Nums can have single or multi digits `>>> 3 + 33`
- No restriction of num of additions we can execute `>>> 3 + 33 + 44 + 723`
- No restriction on length of a sequence of nums `>>> 3333333 + 444444`

4.2. Subtraction (-), Multiplication (*) & Division (/). In much the same way, we can do subtraction, multiplication & division operations. Here again there are no restrictions on the num of digits in a num or in a sequence.

```
>>> 8-4
4
>>> 8*4
32
>>> 8/4
2.0
```

4.3. Multiple Arithmetic Operations.

Say we want to add & subtract. It is easy & has no problems.

```
>>> 7 + 8 - 9
6
```

Say we want to add & multiply. `>>> 6+3*6` It could return 24, which is wrong, or it could return 54, which is correct. This is because in such cases, rule of BODMAS will apply. Thus, we must first put 6+3 in brackets, execute the bracket to get 9, & multiply it by 6 to get 54.

```
>>> (6+3)*6
54
```

BODMAS in python is called **PODMAS.** In this, P **(parentheses)** has **highest priority** & S the lowest.

4.4. Exponent (**). This raises first num to the power of the second. Say we want value of 3 to power of 3. We could multiple 3, three times. This may not appear of significance for small values. However say we need to multiply it 16 times. Doing it by above method will be bad. In such multiplications use the ** **operator**

```
>>> 3*3*3
27
```

```
>>> 3 ** 16
43046721
```

4.5. Modulus (%). Another useful operator. It divides & returns **value of the remainder** called modulus or modulo, or mod. Thus, mod of 10/3 is 1. Mod of 11/3 is 2.

```
>>> 10%3
1
>>> 11%3
2
```

4.6. Floor Division (//). In Python, // is used to conduct floor division. It is used to find the floor of the quotient when first operand is divided by the second.

```
>>> x= 11
>>> y = 3
>>> z = x//y
>>> z
3
```

```
>>> x = 444
>>> y = 321
>>> z = x//y
>>> z
1
```

5 Comparison or Rational Operators

Comparison operators **compare LHS values with RHS values** `>>> 6 > 7` to determine, whether the **two values** are **equal,** or if one is **higher,** or **lower** than the other, & then return a decision. Return of comparison is always a boolean value, either **True** or **False.**

Say the nums to be compared are:

```
>>> a = 9
>>> b = 5
```

Comparison operators check:

- **>.** Value of left operand is **greater** than right.
- **<.** Value of left operand is **less** than right.

```
>>> print('a>b is',a>b)
a>b is True
>>> print('a<b is',a<b)
a<b is False
```

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

08 Working with Logical Operators

1 Lesson Overview

In has lesson we shall understand the magic that coding creates in electronics, the correct method of viewing Boolean expressions, Types & real-life applications of logical operators, working with logical comparison operators, making of chained comparison codes & one line codes.

2 Magic of Coding in Electronics

Conventional way of allowing or not allowing current to flow is based on hardware switches & gates. However, we can achieve better result using **Code with very little hardware.** This approach holds the key to **miniaturisation** & pumping in **limitless switching capabilities** in devices. It holds the key behind making devices like software defined radio or software defined juke box or **software defined** movie player. **Logical operators** are instrumental in making this happen, & thus the importance of this lesson.

3 Viewing a Boolean Expression

Logical operators relate to **world of Booleans** & not arithmetic's. **They enable us to make decisions based on multiple conditions. Operands are conditions (x>18)** & **(y==25)** placed on either side of the **logical operator (and).**

```
>>> if (x>18 and y==25):
```

The outcome of such an operation is either **true** or **false** & not any discreet value. This true or false is then **used to switch on or switch off a circuit** or execute other operations.

3.1. Arguments in Code: Value to a var is given at the time of writing the code. This is its default value. However, when working with sensors or for interactivity, we may not know the value when the code is written. Such values are given at the time of use. To distinguish them from default, these are given as **Arguments** (args).

4 Role & Types of Logical Operators

Logical Operators have **two important** roles:

- As a **code enabler** - to allow data to pass.
- As a **code inhibitor** - to stop data from passing.

This ability of enabling & inhibiting, comes from its evaluation of the two conditions. This capability enables us to make **software defined** switches etc.

Logical Operators are of three types - **AND** operator, **OR** operator & **NOT** operator.

4.1. AND Operator. It is represented by **&&** but is written as **and.** This takes two conditions as arguments (**x>8** & **Y==25**).
It **returns** True when **both** evaluate to True, & False otherwise.

```
>>> x = 20
>>> y = 25
>>> if (x>18 and y==25):
        print('True')
```

4.2. OR Operator. It is written as **||** or as **OR.** This operator also takes two args. It returns true when either or both args are true.

```
>>> x = 22
>>> print(x > 11 or x < 33)
True
```

In an **industrial plant** if one or more parameter exceeds safe value, then its **protective code,** that uses **OR** logic, will put its safety measures into motion.

4.3. NOT Operator. NOT operator is written as **|** or as **NOT**. It does **double evaluation**
To understand, let us take a **var:** `>>> x = 22` Condition to be evacuated is: `>>> x > 11`
To evaluate under **NOT,** focus on evaluation code: `>>> print(not(x > 11 and x < 33))`
Its evaluation under **NOT** takes place in in two stages:
In 1st stage, the first part of the expression (x > 11) is evaluated by **NOT** & in this case 22 is > 11 so it is True. However, this is reversed by **NOT** to **False .**
In 2nd stage, the entire expression (**x>11 and x<33**)) is evaluated as one by **AND.** To enable **AND** to return **true,** both parts should be true. Since 1st evaluation is **False,** then even through in reality it is **true, AND** will also return False.
Thus, **NOT** operator **inverts the result of actual evaluation**.

4.4. Understanding logical operators. Let us take a daily life example of **Eating a Burger.** In this we evaluate two **conditions:**

- **Condition 1.** Has the burger come.
- **Condition 2.** Am I hungry.

The **case of AND operator is easy.** As per our burger example. Evaluating under **AND**, if the outcome of both conditions, namely, has the burger come & am I hungry is True, then I will get to eat. Not otherwise.

The **case of OR is also easy.** As per Burger example. Evaluating under OR implies:

- If the burger does not arrive, **and** I is not hungry i.e., both the conditions become false; I will not eat.
- If **either** the burger comes, **or** I am hungry, or **both,** I will get to eat.

The case of **NOT** operator is a bit tricky. As per burger example:

- It first evaluates the first condition - arrival of the burger as a **NOT** operator. Say it arrives. Itevaluates it as **true,** but **reverse** the outcome true to false (implying it has not arrived).
- It will then evaluate this **false** (in this case), with second condition am I feeling hungry (true) using an **AND** operator. Now since evaluating operator is **AND,** both conditions should be true to return true. But since one is false & one true, it returns **false.**

Thus, even if actually the **burger has come**, & actually **I am hungry, I cannot eat.**

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

09 Python Functions & Arguments

1 Lesson Overview

In has lesson we shall understand what are functions, arguments & their types. Making, storing & naming user-defined func. Meaning of print, return & pass. Understanding different types of functions & args.

2 What is a Function

Function (Func) is a **block of code,** having a name & a specified format, to perform specific tasks. Once a func is declared, it is stored. It **runs only** when **called.** At that time, it has the option to print or if specified, to **return a value.**

In a program, a func can be called anywhere & any num of times. At time of call, it can be infused with new data values **(arguments).** With each return, input values could be changed. This gives it the capability of being re-used with added flexibility, as well as being shared across programmes.

3 Types of Funcs

They are of two types, **Built-in or User defined.**

```
>>> def learning():
        print('I am learning Python')
```

print() is its simplest & most used example.

It is a built-in function that comes embedded in python.

3.1. Built-in functions. We have 69 that perform operations we specify. They are listed at: https://docs.python.org/3/library/functions.html

print ()	type()	input()	abs()	pow()
divmod()	sorted()	ord()	len()	sum()
help()	max()	dir()	round()	id()

These are built into the python interpreter. These are some of the most used func.

3.2. User Defined Functions. These are defined by us to do certain specific tasks. Its syntax is:

```
>>> def FuncName():
        statement1 (optional)
        statementn (optional)
        print() # These lines make a func block


>>> FuncName() # Calling the func
```

As programmers it is important to have a clear understanding of how Funcs are **created,** how they are **called,** how they are used, what they **can do** & what they **cannot do**

3.3. Making a Func. 1st line **defines the func** using keyword def. Its syntax is func_name(). Its name is **"my_func"**. The line must end with a colon : 2nd & subsequent lines **declares func body.** They spell statements & action we want it to perform.

This case has no statements but action Print with data (greenstring) entered inside the bracket. We need to remove indent (blank space) to get out of the func. This is done by **pressing enter twice.** 3rd line indicates you are **out of the func.** In addition, it **calls** the func as & when required.

```
>>> def my_func():
        print('This is a demo func')


>>> my_func()
This is a demo func
```

3.4. Storing the Func. Like var, func is stored at a **unique ID** loc. They can now be used again & again. In addition, green text can be **re-defined** at time of each re-use. This provides modularity & high degree of code re-use.

3.5. Guidelines for Naming Funcs. Every func is a **call for action.**
Give meaningful Names:

- **write(to:** "filename.txt"): `>>> def write(to: ' '):`
- **reloadTableData():** means to reload table data.
- **isBirthDay():** `>>> def isbirthday():`

- **getAddress():** or **setAddress(**"Myaddress"): `>>> def setAddress():`
- **increase(by** increaseAmount: integer): `>>> def increase(by:3):`

3.6. Difference between Print & Return. These are **completely different** functions:

- **Print is a built-in func call.** Its calling **writes out** the text for you to see.
- **Return is a keyword.** When python sees return, it stops the execution of the current func, sending a value to where the func was or will be. The value it returns is called **return value.** Used to send value from one point in code to another, like from inside the func to outside.

3.7. Statement Pass in Funcs. Empty code is not allowed in loops, funcs, class, or in if statements. **Pass** is used as a **placeholder** for **future code.** When pass is executed, nothing happens & error is avoided .

```
>>> def my_function():
        pass

>>>
```

4 Understanding Working of Different Types of Funcs & Arguments

Catering to real-life requirements, we have four options to def a func:

- With parameters (def in black in the bracket). `>>> def adding(x , y):`
- Without parameters (def as bracket empty). `>>> def adding():`
- With print cmd (appears in black). `print('Answer:', Add)`
- With return cmd. `return Add`

Based on the above, python supports four types of func:

- Func with **no** argument & **no** return.
- Func with **no** argument, but **with a Return value.**
- Func **with argument,** but **no** Return value.
- Func **with argument** as well as **with a return** value.

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER 10 Python Literals

1 Lesson Overview

In has lesson we shall understand what are python literals & see the Literals tree. We shall see the character literals & understand the ASCII tables where they are housed. We shall then see String literals & Numeric literals. Thereafter we shall see examples of numeric literals Int, float & complex including the Angad Diagram. Finally, we shall see Boolean literals, special literal none & collection or sequence literals.

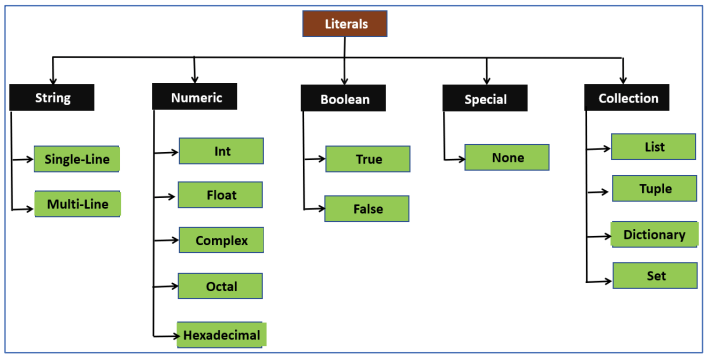
2 What are Python Literals

Python literals are fixed values that are encoded directly into a program's source code and are used to represent data types. They are the building blocks of a program and have values concisely. They are used to provide initial values for parameters, variables, & data structures in a code. They also designate data that can't be modified, such as data specifying the software's operating parameters.

Mathematically, **literals constitute a set of strings** defined by finite set of rules, called **programming grammar.** Two separate passes in the front end, by the **scanner** & the **parser,** determine whether or not the type of input code complies with the **Syntactic grammar of python.** This allows creation of **multiple variable possibilities called literals.**

2.1. Literals Family Tree. We were introduced to some of them in the basics.

In this lesson we shall have another look before going into details in separate lessons for each.



2.2. Mutable & Immutable. These are two very important terms while dealing with literals. Mutable means the values can be changed. Immutable means values are fixed.

3 Character Literals

Character (Char) is not a typical literal or a data type, but a String which is of one character length. It has interesting uses, functions & concepts:

- **Firstly,** it considers all character or set of characters placed inside quotes as a string.

```
>>> s = 'M'
>>> s1 = 'Man'
```

```
>>> s2='$ 45'
>>> s3= '=>$ 7'
```

- **Secondly,** computers only understand binary nums. Now if we want to represent characters like A, @, $ etc we need to map them to discreet nums. For this, Python uses ASCII tables. In these, each char has a unique value allotted to it (ex **65** represents **A** & **97** represents **a**).

3.1. Number representation in ASCII tables. These are as follows:

- Nums from 1 to 32 are for **non-printing characters** like null, start of text, end of text, backspace, escape, device control 1, space etc.
- Nums 33 to 127 are for **printable characters:**
- Nums 33 to 47 are for delimiters like “, #, %, +, (,).
- Nums 48 to 57 for integers 0 to 9.
- Nums 58 to 64 for :, :, <, =, @ etc.
- Nums 65 to 90 for upper case char A to Z.
- Nums 91 to 96 for [,\, * etc.
- Nums 97 to 123 for lower case char a to z.
- Nums 124 to 127 for (, |,), -, DEL.

Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr	Dec	Chr
0	NUL	26	SUB	52	4	78	N	104	h
1	SOH	27	ESC	53	5	79	O	105	i
2	STX	28	FS	54	6	80	P	106	j
3	ETX	29	GS	55	7	81	Q	107	k
4	EOT	30	RS	56	8	82	R	108	l
5	ENQ	31	US	57	9	83	S	109	m
6	ACK	32		58	:	84	T	110	n

3.2. Extended ASCII. These are contained in tables from 128 to 255. These:

- Give **Octal & Hex equivalents** of above chars. Ex: Octal equivalent of A is 41 & its Hex equivalent is 101.
- Give **binary** equivalents, **html** equivalents & **HTML names** for ASCII Ch.

ASCII Code	Char	Description
128	Ç	Latin Capital Letter C With Cedilla
129	ü	Latin Small Letter U With Diaeresis
130	é	Latin Small Letter E With Acute
131	â	Latin Small Letter A With Circumflex
132	ä	Latin Small Letter A With Diaeresis
133	à	Latin Small Letter A With Grave

See Table Site www.LookupTable.com. It provides access to:

1. ALT code chars

ALT Code	Char	Description
ALT 1	☺	White Smiley Face
ALT 2	☻	Black Smiley Face
ALT 3	♥	Black Heart Suit
ALT 4	♦	Black Diamond Suit

2. HTML chars

Character	Entity Name	HTML	CSS	Unicode	Description
ℵ	ℵ	ℵ	\2135	U+2135	Alef Symbol
✗	✗	✗	\2717	U+2717	Ballot X
ב	ℶ	ℶ	\2136	U+2136	Bet Symbol
♣	♣	♣	\2663	U+2663	Black Club Suit

3. Unicode chars

Block Range	Block Name (+Link to Unicode PDF)
U0000 - U007F	Basic Latin
U0080 - U00FF	Latin-1 Supplement
U0100 - U017F	Latin Extended-A
U0180 - U024F	Latin Extended-B

4. EBCDIC chars

Dec	Hex	Oct	Binary	Character	Description
0	00	000	00000000	NUL	Null
1	01	001	00000001	SOH	Start Of Heading
2	02	002	00000010	STX	Start of TeXt
3	03	003	00000011	ETX	End of TeXt
4	04	004	00000100	PF	Punch ofF

Look up table.com is a very useful site. Do refer & read.

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

11 Sequences

1 Lesson Overview

In has lesson we shall understand what are homogenous & heterogenous, mutable & immutable sequences, types of sequences, unpacking of sequences, functions & methods supported by sequences, & formatting of sequences.

2 What is a Sequence

Python's data structure uses two methods to classify its elements:

- **As Sequences.** This is a generic term for elements having deterministic ordering. In sequences, elements **come out** in the **same order** as they are put in. Ex strings.
- **As Collections.** This is classification based on their **container** in the form of **brackets.**

In English, one or more alphabet placed one after another make a word.

```
>>> object = 'Apple'
```

In python, one or more characters placed one **after another** make a **string.**

In much the same way, one or more **strings** placed one after another make a **Sequence.**

```
>>> sequence = 'Apple', 'Orange', 'kivi'
```

In essence sequences are a **positionally ordered collection** of items, where all the items can be assessed by their position called **index value.**

2.1. Difference between Homogeneous & Heterogeneous Sequences. A sequence can be homogeneous or heterogeneous. In a homogeneous sequence, all elements are of the same type. For ex in a string of nums all elements are nums. In heterogeneous, you can store elements of different types including integer, strings, objects, etc. In terms of storage & operation, homogeneous are more efficient than heterogeneous.

2.2. Difference between Mutable & Immutable Sequences. Elements in mutable sequence can be changed. It includes lists & byte arrays. Those in immutable sequence cannot be changed. It includes strings, tuples, range objects, & bytes.

2.3. Relationship between Sequences & Iterables. An iterable is a collection of objects where you can get to each element or object one by one. Thus, **any sequence is iterable.** For ex, a list or a tuple.

However, an iterable may not be a sequence type. For ex, a **set is iterable, but it's not a sequence type.** In general , iterables are more general than sequences.

3 Types of Python Sequences

Traditionally, python supports six main types of sequences. These are **strings, lists, tuples byte sequences, byte arrays,** & **range objects.** De-facto, they are a different method, to create variables for our use. We have full chapters on these ahead.

3.1. Strings.

- String is a **sequence of Unicode** characters making items wrapped in single or double quotes:
  - String 1 of alphabets.
  - String 2 of words.
  - String 3 of numbers.
  - String 4 of single alphabet.
- Once made, order of characters in a string cannot be changed. They can be iterated upon (goneover) & **indexed.** They are **immutable.** If changed, it becomes a new string. For details, refer to lesson 12.

3.2. Lists. Is an heterogenous collection of items wrapped in square brackets **[ ].** A list may contain numbers & strings.A num can be with or without quotes. Num with quotes acts as a string.

```
>>> list = [22, 'Apples', 'I am learning Python']
```

Lists are **Ordered.** Order of objects in a list cannot be changed. They can be **iterated** upon & **indexing** applies. Lists allow for **duplicates.** List items are **mutable** & dynamic. We can add or remove, allowing then to grow or shrink. We can assign new value to the objects to create a **new list.**
Note: There are **workarounds** that can change the order. For details refer to lesson 14.

3.3. Tuples. small Tuple is an **heterogenous collection** of items wrapped in round brackets ( ). Tuple items could be numbers, strings, lists etc or a mix.

Tuples are **Immutable, ordered** & have **duplicates.** They can be **iterated** upon & indexed. In Tuples, use of **Parenthesis ()** is not mandatory, but is a **good practice.**Its use also avoids confusion & imparts efficiency in writing & de-bugging.

```
>>> tuple = ('cat', 'rat', 'bat')
>>> tuple = ('cat',) # Single tuple.
>>> tuple = ( ) # Empty Tuple
```

In line 2, a **single item tuple** should have a comma at the end. If not used its type will be string & not tuple. In line 3, an **empty tuple** must have a parenthesis. For details refer to lesson 15.

3.4. Range() Objects. It is is a built-in sequence that returns a range of objects. In this ex: 1st line def the variable **x** with range of **5** nums. 2nd is a **for** loop. On call for print, it returns a sequence of nums ()as a string (see red box) stating from **0**, with increments by **1** & stopping before the specified num (**5**) as per indexing rules.

```
>>> x = range(5)
>>> for nums in x:
        print(nums)
```

```
0
1
2
3
4
```

Note: In all codes, the output is shown in red box.

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

12 Python Strings & Methods

1 Lesson Overview

In has lesson we shall understand the basics of strings, types of strings, string operations, functions & methods, difference between methods & functions, concatenation of strings, & looping through a string.

2 What are Strings

Strings are an array or a sequence of Unicode characters called elements inside single or double quotes. Or in **triple** single or triple double quotes. It could be a string of:

- A single character.
- Alphabets..
- Nums. Use of quotes with nums is not mandatory. It shall pass, but its type will be an int & not string.

```
>>> x = 'H'
>>> x = 'Multiple'
```

```
>>> x = '2222'
>>> x1 = 2222
```

```
<class 'int'>
>>> type(x)
<class 'str'>
```

- Mix of alphabets & nums.
- Mix of alphabets, nums & symbols.
- Or even an empty string.

```
>>> x= 'Highway 101'
>>> x = 'Mix of Alpha, 22 & symbols'
>>> x = ''
```

3 Understanding String Basics

3.1. Assigning Strings. Use assignment operator = On assignment it gets stored with a unique 13-digit ID. Strings are **immutable.** **'Marut car'** will be a new string with new location ID.

```
>>> x = 'Maruti car'
```

```
>>> id(x)
2100517267568
```

3.2. Calling strings. They are called using a calling func like **print(), return()** etc

```
>>> print(x)
Maruti car
```

3.3. Avoiding Confusion in use of Quotes. Two single quotes & this apostrophe in **'rohan's cellphone'** confuses python resulting in error. To avoid this, the entire string needs to be **placed in a double quote.** This removes the confusion.

```
>>> x = 'Rohan's Cellphone'
SyntaxError: invalid syntax
>>> x = "Rohan's Cellphone"
>>> print(x)
Rohan's Cellphone
```

Second method is the use of \ backslash before, what you want to be ignored. Python will skip or **ignore the apostrophe** associated with Rohan's. Doing so, the entire string **practically** comes under single quotes & thus works fine.

```
>>> print('Rohan\'s Cellphone')
Rohan's Cellphone
```

3.4. Use of Indent while Assigning. It is a good practice to indent strings according to the view you desire in the output. Note the two boxes in this example. In 2nd line of box 2, we haveleft a space before Apples, because we want one in the output.

```
>>> x = '22'
>>> y = 'Apples'
>>> z =(x+y)
>>> z
'22Apples'
```

```
>>> x='22'
>>> y=' Apples'
>>> z=(x+y)
>>> z
'22 Apples'
```

3.5. Sequence of Strings. Made using comma separated Strings.

```
>>> sequence = 'apple', 'orange', 'mango'
```

To be of use they must be **assigned to a literal.** This is done by enclosing it in a **bracket.**

Enclosing it in **round bracket** makes a Tuple in **square** a List & in **curly bracket** a Set.

```
>>> tuple = ('apple', 'orange', 'mango')
>>> list = ['apple', 'oragne', 'mango']
>>> set = {'apple', 'orange', 'mango'}
```

4 Types of Strings

- **Single line strings.** Text occupying one **continuous** segment of any length, in same set of quotes, makes a single line string. Trying to enter in between returns >>>.

```
>>> text1 = 'This is an example of a single line string. It can spill over to th
e second line of IDLE as long as it is in one set of quotes'
>>> print(text1)
This is an example of a single line string. It can spill over to the second line
 of IDLE as long as it is in one set of quotes
```

- **Multi-line strings.** These are strings spread over more than one line.

4.1. Methods of Making Multi-line Strings:

- **Method 1** – Use **( ).** Three separate strings split in three lines. Note indent left at beginning of lines 2 & 3. This is because we want one in the output.

```
>>> t = ('I am Ram.'
      ' He is Shyam.'
      ' We are Friends')
```

```
>>> t = 'I am Ram.'\
      ' He is Shyam.'\
      ' We are friends.'
```

- **Method 2** – Add backslash \ at the end of each line.

- **Method 3** – Use set of three single ‘’’ or three double “”” quotes at either end of the text of the string.

```
>>> text4 = """ I am learning Python.
I chose BDS Education for it.
Because there teaching is logical."""
```

- **Method 4** – Use **.join** func **\n.**

Note: Using join, we need a string to join with. So, we have put an empty string at the start of its def. Also note **\n** for new line at the end of each. **Method 3 is most preferred.**

```
>>> text5 = ''.join(("I am learning Python.\n",
                     "I chose BDS Education.\n",
                     'Because their teaching is logical.'))
```

5 String Operations

5.1. Assignment Operation. String can be assigned to any var (y), class (fruit), collection (list) or sequence (tuple) using **name = ‘string’.**

```
>>> y = '44'
>>> fruit = 'Apple'
>>> list = [11,22]
>>> tuple = ('Rat', 'Cat')
```

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

13 String Formatting, Indexing & Escape sequences

1 Lesson Overview

In has lesson we shall understand the concept of placeholders, what is formatting, methods of string formatting, what is indexing, indexing of strings, what are escape sequences & their types & working.

2 Concept of Placeholders

Placeholders are an **important entity** for formatting. Think of them as public transport like taxi, designed to carry passengers. Who those passengers are, or will be, is not known, but we know it will carry x num of passengers & has seats for them. When used, seats are temporarily assigned to them, till they get down at their destination. Thereafter, seats are ready for next set of passengers.

While coding, we know we will need certain num of values, at a given place in a code; but at that point of time, we do not know those values. In such cases, we put a container of **curley brackets { }** or a **value holder,** until a proper value can be assigned to it.

Now as & when it is used, the code will ask its user to give the final values. These get entered in placeholders in line 4 of blue box, to be printed in line 4 of red box.

```
name = input('What is your name? ')
age = input('What is your age? ')
team = input('Which IPL team you support? ')
print('I am {0}, age {1}, & I support {2}'.format(name, age, team))
```

```
What is your name? Rohit Sharma
What is your age? 32
Which IPL team you support? MI.
I am Rohit Sharma, age 32, & I support MI.
```

2.1. Types of Placeholders. They get their type by the type of data put inside them:

- **Named** placeholder **{Age}.** They have a var name inside them.
- **Num** placeholder **{1}.** They have a specify index num inside them.
- **Empty** placeholders **{ }.** They are kept empty.

Putting placeholders in code is not mandatory. Once placed, their values can be **changed** or **substituted** at time of use. **Place holder values** can be a **list** of values separated by commas, a **key=value** list, or a combination of both.

```
>>> named_ph = 'I am {fname}, I am {age}'.format(fname = 'Tom', age = 22)
>>> num_ph = 'I am {0}, I am {1}'.format('Tom', 22)
>>> empty_ph = 'I am {}, I am {}'.format('Tom', 22)
```

3 Formatting

Formatting is the process of arranging data, as per a specific arrangement called **format,** in accordance with which computer data is processed, stored, printed, etc. **Formatting** involves **interpolation** or **substitution** or **manipulation** of values of string var in code.

Such substitution is required in interactive & data base applications. Tools that enable this are called Formatters. They work by fixing one or more **placeholders** in the code.

4 Methods of String Formatting

Python offers **three** different ways to format Strings:

- Formatting with **.format()** method.
- Formatting with **Modulo** operator (%).
- Formatting with **string literals,** called **f-strings.**

5 String Formatting using .format() Method

- **Inserting objects by def index-based positions**. In this ex original txt is **'Notifications'** (line 1)..At the time of print (line2) we have inserted **"All** & **Our".** Thus, **all** which needs to be printed first has been given index 2 followed by our, with index **1** & **notification** with index **0,** to print - **all our notifications.**Its logic comes from the **syntax** for formatting - **string.format(value1, value2...).**

```
>>> txt = 'Nofifications'
>>> print('{2} {1} {0}'.format('Notifications', 'Our', 'All'))
All Our Notifications
```

Inserting objects in a string using assigned keywords. In this three objects have been given default values in lines 1 to 3. Line 4 assigns them as keywords (**x, y, z**) to string s. line 5 puts them in key-value pairs with a placeholder for each . On print, .**format** will insert values in respective placeholder to **print** the output. This has also used **named placeholders.**

```
>>> x=22
>>> y='Apples'
>>> z=8.5
>>> s = (x, y, z)
>>> print('x: {x}. y: {y}, z: {z}'.format(x=11, y='Cat', z=2.0)
x: 11. y: Cat, z: 2.0
```

6 String Formatting using Modulo Operator

Symbol **%** is an arithmetic operator. In context of strings, it is called the **Formatting Operator.** To use this with strings, we need to add an alphabet to it (**%s**). This alphabet is called the **formatting specifier,** as it specifies the data type on which has to be formatted. Ex **%s** is used for string conversion while **%i** will perform operations on signed integers.

6.1. Types of Formatting Operators. Following specify different formatting operations:

%c – Formats characters.	**%x** – Hexadecimal integer using lower case letters.
%s – String conversion.	**%X** – Hexadecimal integer using upper case letters.
%i - Signed integer.	**%e** – Exponential notation with lower case e.
%d – Signed decimal integer.	**% E** – Exponential notation with upper case e.
%u – Unsigned decimal integer.	**%f** – Floating point real number.
%o – Octal integer.	**%g** – The shorter of %f and %e.
	%G – The shorter of %f and %E.

Signed integer is a **32-bit integer** containing positive or negative nums. Unsigned contains only positive nums.

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

14 Lists

1 Lesson Overview

In has lesson we shall understand what are Python collections, basics of lists, procedures of making lists, list indexing, list methods, miscellaneous list operations, & making one liners through list comprehensions.

Python is an object-oriented language, wherein variables & sequences constitute the core of these **objects. Python collections** are created by putting these in containers, defined by the type of brackets in which they are placed, & by the method of their ordering.
Over the next four lessons, we shall learn how the four key **collections** – **lists, tuples, dictionaries & sets** are made, & how they are used to code.

2 List Basics

2.1. List is **an ordered** data structure, in which elements are separated by a **comma, &** enclosed **within square brackets.**

```
>>> list = ['Welcome', 2022, 22<23]
>>> list
['Welcome', 2022, True]
```

Lists are created to store **sequential data** using strings, nums, symbols, char & Booleans.

2.2. List Items are Ordered. This means the order in which elements are specified when def, is the order in which they are returned.

```
>>> list = ['Rat', 'Bat', 'Mat']
>>> print(list)
['Rat', 'Bat', 'Mat']
```

If this is compared with a list of same elements, in **different order,** we get false. Meaning two are not equal. It is a **separate list.**

```
>>> ['Rat', 'Bat', 'Mat'] == ['Bat', 'Mat', 'Rat']
False
```

2.3. Lists can Contain:

- Objects of **same** data type.
- Objects of **mixed** data types.
- **Duplicate** objects. **List items need not be unique.** They can appear any num of times.

```
>>> list1 = ['Oil', 'Egg', 'Pin']
>>> list2 = ['Oil', 11, 22<24]
 >>> list3 = [11, 22, 33, 11, 44 , 11, 55]
```

2.4. Lists are Mutable & Dynamic. Lists are akin to arrays in other languages with the benefit of being dynamic. Once created, elements can be **modified** & individual values **replaced.** Elements can be **added & deleted,** allowing it to **grow** or **shrink.** This makes them **dynamic.**

3 Making Lists

We have two methods:

- **Conventional Method.** In this, we directly input values into the list.
- **Using Built-in List Constructor.** This takes as input any data type like tuple set etc, & outputs a list.

3.1. Conventional Methods. Kindly study the following:

- Making a List:
  - Defining a list -1st line.
  - Calling a list - 2nd line.

```
>>> list = [22,33,44]
>>> print(list)
[22, 33, 44]
```

- Types of Lists
  - List of integers.
  - List of strings.
  - List of Tuple.
  - Mix list.
  - Empty list.
  - Nested List.

```
>>> list_int = [11,55,99]
>>> list_string = ['Cat', 'Bat', 22, 44]
>>> list_tuple = ['A', 'B', 'C']
>>> list_mix = ['A', 'B', False, [1,2,3]]
>>> list_empty = [ ]
>>> list_nested = [ 'Apple', ['AB', 'CD'], [4,8,12]]
```

3.2. List Constructor List(). It is a built-in func, to make lists based on parameters passed. **Input objects of a list constructor** include Iterable like Strings, tuples, dict, sets, range objects, numpy array & pandas. If no parameters are passed, it returns an empty list.

```
>>> tuple = ('Apple', 'Orange', 'Mango')
>>> list = list(tuple)
>>> print(list)
['Apple', 'Orange', 'Mango']
```

4 List Indexing

Since lists are ordered, they can be indexed. Individual objects in a list can be accessed using index num in **square brackets**. It is similar to Indexing strings.

4.1. Examples of Indexing. Say our list & its index table are as below:

```
>>> myList = ['Apple', 'Orange', 44, 'Eggs', 'Cake', 3322, 'Salt', 'Oil']
```

List objects:	Apple	Orange	44	Eggs	Cake	3322	Salt	Oil
Index Value:	0	1	2	3	4	5	6	7
Negative Index	-8	-7	-6	-5	-4	-3	-2	-1

- **Retrieving** a single item.

```
>>> myList[7]
'Oil'
```

- **Negative** indexing.

```
>>> myList[-5]
'Eggs'
```

- **Single step** range slice.

```
>>> myList[2:5]
[44, 'Eggs', 'Cake']
```

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

15 Tuples

1 Lesson Overview

In has lesson we shall understand what are tuples, tuple basics, making tuples, working with tuples, tuple operations, methods & functions, miscellaneous tuple operations, tuple workarounds, & reasons of using tuples over lists.

2 Tuple Basics

2.1. Tuple Items are in Round bracket & Ordered. The order in which you specify elements when you def a tuple is maintained for its lifetime. Tuples that have same elements in a different order are a different tuple.

```
>>> (11, 22, 33, 44) == (33, 22, 44, 11)
False
```

2.2. Tuples are Immutable. Unlike lists, once a tuple is made, its elements cannot be **modified,** individual values cannot be **replaced** & order of elements cannot be **changed.** To overcome this, some **workarounds** exist. We shall see these later in the lesson.

2.3. Tuples can contain:

- Objects of **same** data type (1st & 2nd lines).
- Objects of **mixed** data types (3rd line).

```
>>> t1 = ('Egga', 'Bread', 'Butter')
>>> t2 = (22, 33, 44,44, 3+4j)
>>> t3 = ('Oil', 22, 8.5, 3+5j)
```

Like lists, tuples also allow for **duplicate** values.

```
>>> t3 = ('AB', 'AC', 'AB', 'AD')
```

In terms of **indexing, nesting & repetition,** they are similar to lists.

3 Making Tuples

We have two methods:

- **Conventional Method.** We feed in directly.
- **Using Built-in Tuple Constructor.** This uses a tuple constructor

3.1. Conventional Method.

Study carefully:

```
>>> tuple = ('India', 'USA', 'Aus')
>>> print(tuple)
('India', 'USA', 'Aus')
```

- **Normal tuple.**
- **Single item tuple.**

```
>>> t1 = ('Car',)
>>> t1
('Car',)
>>> print(type(t1))
<class 'tuple'>
```

Note: In a single item tuple, we put a comma after the item ('Car' in 1st line). This signifies there is another item. Its use is optional.

- **Empty tuple.**

```
>>> t4 = ( )
```

- **Nested tuple.**

```
>>> t5 = (('AA', 'BB'), [22, 33, 44], ( ))
```

This nested tuple containing one tuple, one list & one empty tuple.

- **Tuple without brackets.** It shall pass even if bracket is not used. Use of Bracket is not mandatory, but a **good practice.**

```
>>> t7 = 'Cars', 22, 'Bikes', 8
```

3.2. Tuple Constructor tuple(). This is a built-in func used to make tuples based on the parameters passed to it. It is similar to the list constructor. Best used with another seq.

```
>>> list = [11, 22, 33, 44, 55]
>>> tuple = tuple(list)
>>> print(tuple)
(11, 22, 33, 44, 55)
```

4 Working with Tuples

4.1. Accessing Tuple Items. Say our tuple is:

```
>>> tuple2 = (11, 22, 33, 44, 55)
```

To access items:

- Single item.
- Negative index.
- Range of items.
- Range to end.
- Range from beginning.

```
>>> print(tuple2[2])
33
>>> print(tuple2[-3:-1])
(33, 44)
>>> print(tuple2[1:3])
(22, 33)
>>> print(tuple2[2:])
(33, 44, 55)
>>> print(tuple2[:3])
(11, 22, 33)
```

4.2. Unpacking a Tuple. In this ex, line 1 offers a simple method of unpacking. It places the tuple on the right hand side of an assignment operator with var (x,y,z) on the left. Num of var must equal num of items in tuple. If not, use Asterix method.

Unpacked items come out as a List.

```
>>> x,y,z = ('Spong', 'Choco', 'Pineapple')
>>> print(y)
Choco
```

4.3. Asterix Method of Unpacking. If num of var in line 2 is less than the num of values in line 1, then in 2nd line LHS you can add * to the last var **(d).** Then **d** will be assigned a value that will take it from **d** to the end, to return a list of unpacked items.

```
>>> numsl = (11,22,33,44,55,66)
>>> (a, b, c, *d) = numsl
>>> print(d)
[44, 55, 66]
```

5 Tuple Operations

5.1. Concatenation of Tuples.
(+) used to concatenate or combine two tuples (3rd line). Returns a new Tuple.

```
>>> starters = ('Vada', 'Chips', 'Fries')
>>> add_on = ('Nuggets', 'Rings')
>>> appitizers = starters + add_on
>>> print(appitizers)
('Vada', 'Chips', 'Fries', 'Nuggets', 'Rings'
```

5.2. Repetition of a tuple.
Use **(*)** to repeat a tuple (Hip Hip Hurray) a specified num of times.

```
>>> tuple = ('Hip Hip Hurray, ')
>>> cheer = (tuple * 3)
>>> print(cheer)
Hip Hip Hurray, Hip Hip Hurray, Hip Hip Hurray,
```

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

16 Dictionaries

❶ Lesson Overview

In has lesson we shall understand what are dictionaries, dictionary basics, making of dictionaries,methods & functions in dictionaries, sorting & looping of dictionaries, & nesting of dictionaries.

In Python, a dict stores data in **key – value** pairs.
In this example, each **key,** on the left of colon : `>>> dict = {'Brand': 'Tata', 'Qty': 44}` is associated with a **value** on its right. Keys & values appear in green. Key-value pairs are separated **by comma.** They are then put into a **sequence** inside **Curley** brackets, storing multiple items in a single var.

❷ Dictionary Basics

Dictionary (dict) Items are **Ordered** as key-value pairs. `{'Cars': 4, 'Bus': 2}`
The order in which they are defined, is maintained for its lifetime. Dict items can be **referred to & called** using the key name. They **cannot be indexed.** If the order of elements is changed, it becomes a new dict with a new loc ID.

2.1. Dict items are mutable. Like lists, once a dict has been created:

- Elements can be **modified.** `'Cars': 4`
- Individual values of these elements can be **replaced.**
- Dict **do not** allow for **duplicates.**

2.2. Calling Dict Items. To call, use **key name in square brackets** (2nd line).

```
>>> dict = {'Family Name': 'Tom', 'Age': 22, 'Sex': 'M'}
>>> print(dict['Age'])
22
```

Note: In the square bracket key name must appear as def in the dict. Not doing so returns Name error

```
>>> print(dict[Age])
NameError: name 'Age' is not defined
```

2.3. Data Type of Keys & Values. In a dict:

- **Keys** are **unique** while **values** may not be.
- Keys must be of an **immutable** data type such as strings, nums, or tuples.
- We cannot use **mutable object like a list** as a key, but it can be used as a **value.**
- **Improper use** of keys or values in terms of mutability will result in Type error.

```
>>> mydict = {[1,2]: 'Cars'}
TypeError: unhashable type: 'list'
```

- **Values** can be strings, nums, Booleans & lists.

3 Making Dictionaries

We have five methods of making dict.

```
>>> dictA = {'Boys': 210, 'Girls': 178}
>>> dictA
{'Boys': 210, 'Girls': 178}
```

3.1 Normal Method. Dictionaries are created using multiple key-value pairs having keys separated from values by colons : & in turn separated from each other by comma, & enclosed in curley brackets **{ }.**

3.2. Method 2. Passing Parameters to Dict Constructor. This is similar to the earlier ex. In this the key-value pairs are entered in the round bracket of the dictionary constructor (line 1).

```
>>> dictA = dict(Grade = 6, Sec = 'B', Students = 44)
>>> dictA
{'Grade': 6, 'Sec': 'B', 'Students': 44}
```

3.3. Method 3. Using a List of Tuples. This is a method of passing raw parameters from a tuple to a dict. Here we have two tuples. They have been used to create a **list of tuples** (line 3). This list is then **passed** to the constructor in line 4 to make the dict. Can have **any num** of comma separated tuples. Its use can be to make a **data base** based on dict.

```
>>> tup1 = ('Apples', 22)
>>> tup2 = ('Oranges', 44)
>>> tupList = [tup1, tup2]
>>> dict1 = dict(tupList)
>>> dict1
{'Apples': 22, 'Oranges': 44}
```

3.4. Method 4. Using List of Keys Initialised to same Value.
This uses method **fromkeys()** along with the dictconstructor to get the dict. Value (**4** in this case)given now, can be changed when required. Thisis useful when **values are initially not known,** or will be made available later through **sensors or other parts** of the program.

```
>>> fruitbasket = ['Apples', 'Oranges', 'Kivi']
>>> dict2 = dict.fromkeys(fruitbasket, 4)
>>> dict2
{'Apples': 4, 'Oranges': 4, 'Kivi': 4}
```

3.5. Method 5. Using Two lists. In this line 1 is a listof strings. Line 2 is a list of integers. In line 3 we have used the dict constructor with the **zip()** method. Line 3 can have any num of comma separated lists. A typicalex of its use can be for **making an app to compile results** from various sources.

```
>>> listString = ['cata', 'Rats', 'Bats']
>>> listInt = [8, 4, 6]
>>> dict3 = dict(zip(listString, listInt))
>>> dict3
{'cata': 8, 'Rats': 4, 'Bats': 6}
```

4 Dictionary Methods

Dictionaries support **11 methods:**

4.1. Makes a dictionaries copy. In 1st line of code, variable **(mydict)** assigns it the values of the dict to be copied using .**copy()**. Line 2 prints variable mydict with the assigned values as a copy. The other method of doing so is to use **of dict constructor** to make a copy of the dict. Link with 2.2.

```
>>> mydict= dict.copy()
>>> print(mydict)
{'Name': 'Tom', 'Age': 23, 'Sex': 'M'}
```

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

17 Sets

❶ Lesson Overview

In has lesson we shall understand what are sets, area of prime use of sets, types of sets, set basics, making of sets, methods & functions in sets, Use of sets in mathematical computation, Vien diagram, ready reckoner of mathematical set operations, looping through sets, nesting of sets, set comprehensions, & frozen sets.

Everything about sets is **un** - **un**ordered, **un**changeable, & **un**indexed.
Sets are the preferred data type in two distinct areas.

- To **efficiently remove** duplicate values from a list or tuple.
- Perform common **mathematical operations** like unions, intersections & differences.

❷ Set Basics

Sets are used to make **sequences** that store multiple items in a single variable. They are created using strings, int, nums & boolean values placed in **curley** brackets **{ }.**

```
>>> set = { 'India', 22, 6.5, 3+4j, True}
```

2.1. Set items are Unordered. Order in which a set is def is not maintained at time of return. In this ex in our eyes India is at **index 0**, but python is seeing it at **3**. Thus, items of a set **cannot** be indexed using **index** or **keys** values.

```
>>> set = { 'India', 22, 6.5, 3+4j, True}
>>> set
{True, 6.5, 22, 'India', (3+4j)}
```

2.2. Mutability of Sets. In sets, mutability reflects itself in two ways:

- Python sets are **mutable** because the set itself **may be modified.**
- **Elements** contained in the set must be of an **immutable type.**

To be precise, Always Remember:

- Nums, strings, tuples & frozen sets **can be** used as elements in a set.
- On the other hand, lists, dict & sets **cannot be used** as elements in s set.

2.3. Data Type of Set Items can be:

- Of **same** data type.
- Of **mixed** data types.

```
>>> set= {'India', 'Nepal', 'Bhutan'}
>>> set = {'India', 22, True}
```

They automatically remove **duplicates** in provided data.

```
>>> set = {'M', 'O', 'R', 'O', 'T', 'O'}
>>> set
{'R', 'T', 'O', 'M'}
```

❸ Types of Sets

Based on usage, sets can be of the following basic types:

- **Empty.** No defined value.
- **Singleton.** One defined value.

```
>>> { }
>>> {'Apple',}
```

- **Finite.** Finite defined values.
- **Infinite.** Infinite defined values.
- **Equivalent.** Two sets with same num of elements.
- **Equal.** Sets have same elements. Order could be different.
- **Disjoint set.** The sets have no common elements.
- **Subset.** Every element of set A is part of B. Say A = {1,2,3}. Subsets of A are: {1}, {2}, {3}, {1, 2}, {2,3}, {1,3}, {1,2,3} & { }.
- **Proper subset.** A is proper subset of B, but not = B.
- **Superset.** Taking above ex, B is super set of A.
- **Universal.** Contains all values relevant to a condition. It is like a set of all possible values.

```
>>> set = {'Cat', 'Bat', 'Rat'}
```

```
>>> A = {2, 4}
>>> B = {2, 4, 6}
```

```
>>> s1 = {1,2,3,4}
>>> s2 = {3,4,5,6}
>>> U_set = {1,2,3,4,5,6}
```

Note: An empty set cannot use curley brackets as it is used by a dict. Therefore, **a pair of round brackets** is used.

4 Making, Calling & Accessing Sets

4.1. Method 1. Normal Method. Entering values in **{ }**

```
>>> set1 = {1,2,3,4}
>>> print(set1)
{1, 2, 3, 4}
```

4.2. Making Sets using Set Constructor set():

- **Case 1.** Set from strings.
- **Case 2.** Set from a tuple.
- **Case 3.** Set from a list.
- **Case 4.** Using iterable unpacking operator *.
- **Case 5.** Using set constructor.

```
>>> set1 = set('ABC')
>>> set1
{'C', 'A', 'B'}
```

```
>>> set2 = set((1, 2, 3))
>>> set2
{1, 2, 3}
```

```
>>> set3 = set([6, 7, 8])
>>> set3
{8, 6, 7}
```

```
>>> x = [1, 2, 3]
>>> set4 = {*x}
>>> set4
{1, 2, 3}
```

```
>>> list = [22, 33, 44]
>>> set5 = set(list)
>>> print(set5)
{33, 44, 22}
```

4.3. Calling Sets. This is similar to other sequences. Returns a set in which **duplicates are removed.**

```
>>> N1 = {1,2,3,2,4,5,2,6,7,2,8,9}
>>> N1
{1, 2, 3, 4, 5, 6, 7, 8, 9}
```

4.4. Accessing Set Items. You cannot access items by referring to an **index** or a **key**. But you can loop through the set items using **for loop.**

```
>>> for x in S1:
        print(x)
```

```
1
2
3
4
```

You can check presence of an item using in keyword.

```
>>> S1 = {1, 2, 3, 4}
>>> print(3 in S1)
True
```

5 Set Methods

These are similar to lists & tuples. Sets supports dot format syntax for:

- **add()** – Adds one item.
- **update()** - Adds multiple items.
- **Union()** – joins two sets.
- **update()** - Object can be a set or an iterable – tuple, dictionary & list.
- **remove()** Removes specified Item.

To view more, please contact:

Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

www.bdseducation.in

CHAPTER

18 Common Errors

1 Lesson Overview

In has lesson we shall understand types of errors, working of trackback module, list of common errors/exceptions, & understanding common occurring errors.

There are two main **types of errors** that can occur in any code. These are:

- **Syntactic errors.** These are the errors that occur when we mistype commands, or when we use a var or a func without defining them, or if the indentations are not correct. In most cases, these are easy to fix if we follow the instruction of **Traceback module.**
- **Semantic errors.** These are errors where the code runs, but wrong answers are returned, or when your code behaves differently from what you expected. These are more involved, time consuming & laborious to resolve.

2 Traceback Module

It is an error correction **module,** providing an interface, **mimicking behaviour of the interpreter.** In so doing, it gives information regarding all error, making it **easier to track, trace & fix.** It ensures that a code, or block of code, moves ahead only once it is error free.

2.1. Structure of Traceback. This has four parts, spread over four or more lines.

- Part1 declares traceback is for most recent call.
- Part 2 gives location in the program.
- Part 3 gives line where error is encountered.
- Part 4 gives name & relevant information about the error.

```
Traceback (most recent call last):
  File "<pyshell#195>", line 1, in <module>
    print(dish)
NameError: name 'dish' is not defined
```

When we encounter this, our attention should **go to last & second last line.**

2.2. Long Tracebacks. In an IDLE we code is tested for errors very frequently. Thus, we mostly see four line deep trackbacks as shown above. However, in professional coding using an IDE, it is common to see long tracebacks like 20 levels deep. However, length of the error message does not reflect severity, rather, it indicates that the program called many functions before it encountered the error. Most of the time, the actual place where the error occurred is at the bottom-most level, so you can **skip down the traceback, & focus on the bottom lines.**

2.3. Working with Traceback Mechanism of IDE. Traceback analysis using PyCharm or any other IDE is done in two parts. To do so, look at the code & its trackback in red.

```
Traceback (most recent call last):
  File "C:\Users\prasu\PycharmProjects\pythonProject\Opencv\code.py", line 8, in <module>
    favorite_car()
  File "C:\Users\prasu\PycharmProjects\pythonProject\Opencv\code.py", line 7, in favorite_car
    print(cars[3])
IndexError: list index out of range
```

```
1  def favorite_car()
2      cars = [
3          'Honda',
4          'Tata',
5          'Hundai'
6      ]
7      print(cars[3])
8  favorite_car()
```

- **Part 1 Source of error.** Focus on the red box. The last line of red box mentions that it is an **indexerror.** It also mentions **list out or range**. With this information we need to look at how many elements we have in the list. Since the num of elements is three, the index value must be between 0 to 2. As against this, we gave index value 3 in line 7 of blue box. If now we fix this, code will run fine.
- **Part 2 Sequence of Pointing.** Trackback starts with line 2 of red box below (that mentions line 8 of blue box). It then moves to line 4 of red box (mentioning line 7 of blue). After this it goes to line 5 of red box, in which it mentions place of **mistake** & hint below it.

3 List of Common Python Errors/Exceptions

Listed below are 24 common errors that you could experience. While you may not experience all of them, at least half of them you will for sure experience. We shall go over the major ones.

S.No	Exception	Description
1	SyntaxError	Raised by the parser when a syntax error is encountered.
2	NameError	Raised when a variable is not found in the local or global scope.
3	IndentationError	Raised when there is an incorrect indentation.
4	TabError	Raised when the indentation consists of inconsistent tabs and spaces.
5	IndexError	Raised when the index of a sequence is out of range
6	KeyError	Raised when a key is not found in a dictionary.

Rest would follow similar logic & by that time you will be in a position to rectify them yourself.
It is good to be aware of all errors. Procedure to fix is similar.

7	KeyboardInterrupt	Raised when the user hits the interrupt key (Ctrl+c or delete).
8	FloatingPointError	Raised when a floating point operation fails.
9	MemoryError	Raised when an operation runs out of memory.
10	RuntimeError	Raised when an error does not fall under any other category.
11	StopIteration	Raised by the next() function to indicate that there is no further item to be returned by the iterator.
12	OSError	Raised when a system operation causes a system-related error.

Errors are Python's Best Learning Tool.
No statement can be More correct than this. Never be disappointed when it returns an error.

13	OverflowError	Raised when the result of an arithmetic operation is too large to be represented.
14	SystemError	Raised when the interpreter detects internal error.
15	SystemExit	Raised by the sys.exit() function.
16	TypeError	Raised when a function or operation is applied to an object of an incorrect type.
17	ValueError	Raised when a function gets an argument of correct type but improper value.
18	ZeroDivisionError	Raised when the second operand of a division or module operation is zero.

In the next sec we will explain you the types layout & reasons for various types of errors. Kindly revert to these if required. **More you Error More you Learn**

19	AssertionError	Raised when the assert statement fails.
20	AttributeError	Raised on the attribute assignment or reference fails.
21	EOFError	Raised when the input() function hits the end-of-file condition.
22	ImportError	Raised when the imported module is not found.
23	MemoryError	Raised when an operation runs out of memory.
24	IOError	Raised when an input/ output operation fails, such as the print statement or the open() function when trying to open a file that does not exist.

To view more, please contact:

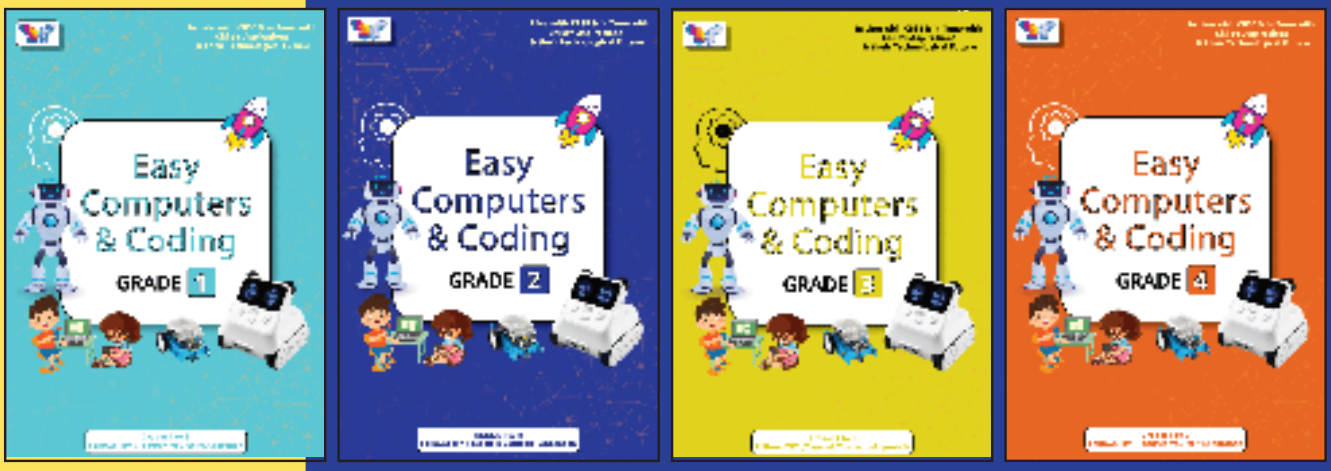
Diwaker :- +91 93122 64502

Pankaj Kabir Panthi :- +91 88514 60895

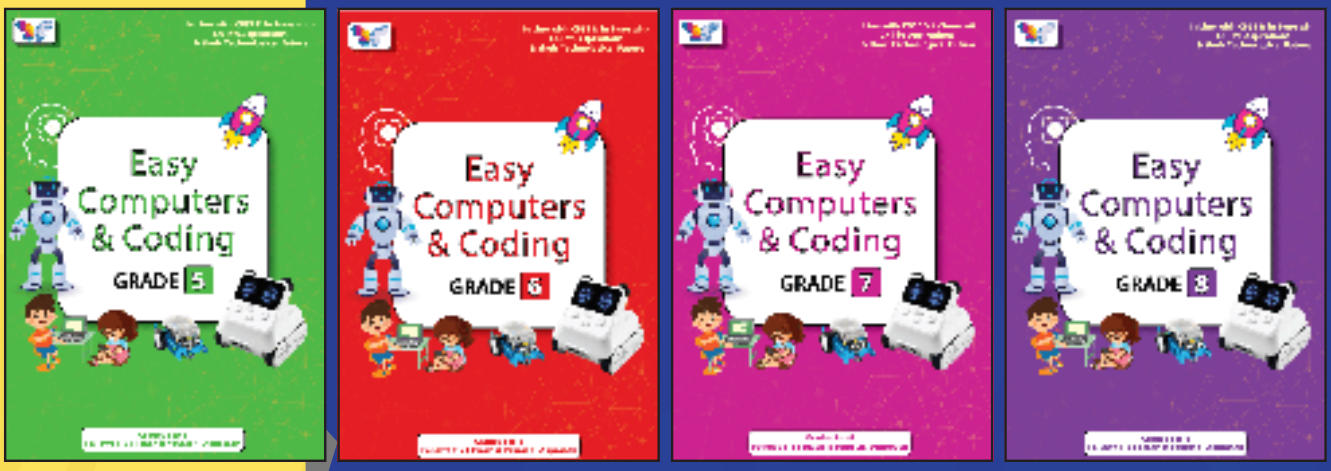
www.bdseducation.in

OUR BOOK TITLES

Level-1



Level-2



Level-3

BDS EDUCATION