Programming in Python

Maheshkumar N

Introduction

- What is Python?
  - ❖ Programming Language
  - ❖ High-Level language
  - ❖ General purpose programming language

1. Programming language:
   a) Machine language
   b) Assembly level language
   c) High level language
2. High level language:
   a) programmer friendly language but not machine friendly language
3. General purpose language:
   a) Implement any application using python like calculator, Gmail app.. etc

Python History

- Python laid its foundation in the late 1980s.
- The implementation of Python was started in the December 1989 by **Guido Van Rossum** at NRI in Netherland.
- In February 20th ,1991, van Rossum published the code (labeled version 0.9.0) to alt.sources.
- In 1991, Python 1.0 was released with new features like: lambda, map, filter, and reduce.
- Python 2.0 added new features like: list comprehensions, garbage collection system.
- On December 3, 2008, Python 3.0 (also called "Py3K") was released. It was designed to rectify fundamental flaw of the language.
- *ABC programming language* is said to be the predecessor of Python language which was capable of Exception Handling and interfacing with Amoeba Operating System.

Python History

Guido Van Rossum

Python Applications

- Web Development
- Game Development
- Machine Learning and Artificial Intelligence
- Data Science and Data Visualization
- Desktop GUI
- Web Scraping Applications
- Business Applications
- Audio and Video Applications
- CAD Applications
- Embedded Applications

Python Features

- ❖ Simple and easy to learn
- ❖ Freeware and Open Source
- ❖ High level programming Language
- ❖ Platform independent
- ❖ Portability
- ❖ Dynamically typed
- ❖ Both Procedure-Oriented and Object-Oriented Language
- ❖ Interpreted Language
- ❖ Extensible
- ❖ GUI Programming Support
- ❖ Embedded
- ❖ Extensive Library

Python Features

- Platform independent

Limitation of Python

- Not suitable for
  - Mobile applications
  - Enterprise applications - Banking Application, Telecom Application-End to End Support
  - Performance is low because of Interpreted nature

❖ Speed

- Python is **slower** than C or C++. But of course, **Python** is a high-level language, unlike C or C++ it's not closer to hardware.

❖ Mobile Development

- Python is not a very good language for **mobile development** . It is seen as a **weak language** for mobile computing. This is the reason very few mobile applications are built in it like Carbonnelle.

❖ Memory Consumption

- Python is not a good choice for **memory intensive** tasks. Due to the flexibility of the data-types, Python's memory consumption is also high.

❖ Database Access

- Python has limitations with **database access** . As compared to the popular technologies like JDBC and ODBC, the Python's database access layer is found to be bit underdeveloped and **primitive** . However, it cannot be applied in the **enterprises** that need smooth interaction of complex **legacy data** .

❖ Runtime Errors

- Python programmers cited several issues with the **design** of the language. Because the language is **dynamically typed** , it requires more testing and has errors that only show up at **runtime** .

Python Versions

- **Python 0.9.0** – Feb 20th, 1991
- **Python 1.0** - January 1994
- Python 1.5 - December 31, 1997
- Python 1.6 - September 5, 2000
- **Python 2.0** - October 16, 2000
- Python 2.1 - April 17, 2001
- Python 2.2 - December 21, 2001
- Python 2.3 - July 29, 2003
- Python 2.4 - November 30, 2004

Python Versions

- Python 2.5 - September 19, 2006
- Python 2.6 - October 1, 2008
- Python 2.7 - July 3, 2010
- **Python 3.0** - December 3, 2008
- Python 3.1 - June 27, 2009
- Python 3.2 - February 20, 2011
- Python 3.3 - September 29, 2012
- Python 3.4 - March 16, 2014
- Python 3.5 - September 13, 2015

Python Software and IDE

- **12 Best Python IDEs:**
  - **PyCharm Spyder Pydev IDLE Wing Vim**
  - **Eric Python Rodeo Thonny Sublime Text Atom**

Python Software and IDE

- https://www.python.org/downloads/
- Installation steps

Differences between Interpreter and Compiler

Interpreter

- Interpreter translates just one statement of the program at a time into machine code.
- An interpreter takes very less time to analyze the source code. However, the overall time to execute the process is much slower.
- An interpreter does not generate an intermediary code. Hence, an interpreter is highly efficient in terms of its memory.
- Keeps translating the program continuously till the first error is confronted. If any error is spotted, it stops working and hence debugging becomes easy.
- Examples: Ruby, Python, matlab

Compiler

- Compiler scans the entire program and translates the whole of it into machine code at once.
- A compiler takes a lot of time to analyze the source code. However, the overall time taken to execute the process is much faster.
- A Compiler always generates an intermediary object code. It will need further linking. Hence more memory is needed.
- A compiler generates the error message only after it scans the complete program and hence debugging is relatively harder while working with a compiler.
- Examples: C, C++ etc...

Identifiers

- A **Python identifier** is a name used to identify a variable, function, class, module or other object.
- A Python identifier can be a combination of lowercase/ uppercase letters, digits, or an underscore. The following characters are valid:
- Lowercase letters (a to z), Uppercase letters (A to Z), Digits (0 to 9), Underscore (_)
- Examples:
  - myVar
  - var_3
  - this_works_too
- An identifier cannot begin with a digit. Example: **9mahesh**
- We cannot use special symbols in the identifier name.**(! @ # $ % .)**
- We cannot use a keyword as an identifier. Keywords are reserved names in Python and using one of those as a name for an identifier will result in a **SyntaxError.**
- An identifier can be as long as you want **(maximum of 79 characters).**
- **Case Sensitive**

Keywords

- Keywords are a list of reserved words and that have predefined meaning.
- You cannot use them as constant or variable or any other identifier names.
- All the Python keywords contain lowercase letters only.
- There 33 Keywords –
  - contains only alphabets
  - Except True, False, None- all contains only lowercase alphabet
  - Switch concept is not there in python
  - do-while is not there
  - Int, float, complex .. keywords are not there in python

Statements and Expressions

- A statement is an instruction that the Python interpreter can execute. Python program consists of a sequence of statements.

  Ex: z = 1 is an assignment statement.

- Expression is an arrangement of values and operators which are evaluated to make a new value. Expressions are statements as well.

  EX: >>> 20 or >>> z or >>> z + 20

- A value is the representation of some entity like a letter or a number that can be manipulated by a program.

```
>>> 8 + 2
10
```

Quotation in Python

- Python accepts single ('), double (") and triple (''' or """) quotes to denote string literals, as long as the same type of quote starts and ends the string.
  - word = 'word'
  - sentence = "This is a sentence."
  - paragraph = """This is a paragraph. It is
  - made up of multiple lines and sentences."""

Variables

- Variable is a named placeholder to hold any type of data which the program can use to assign and modify during the course of execution.
- In Python, there is no need to declare a variable explicitly by specifying whether the variable is an integer or a float or any other type.
- *To define a new variable in Python, we simply assign a value to a name.*
- The declaration happens automatically when you assign a value to a variable. The equal sign (=) is used to assign values to variables.

```
ounter = 100        # An integer assignment
miles   = 1000.0    # A floating point
name    = "John"    # A string
```

- Python allows you to assign a single value to several variables simultaneously. For example –
  - **a = b = c = 1**

Legal Variable Names

Follow the below-mentioned rules for creating legal variable names in Python.

- Variable names can consist of any number of letters, underscores and digits.
- Variable should not start with a number.
- Python Keywords are not allowed as variable names.
- Variable names are case-sensitive. For example, computer and Computer are different variables.
- Ensure variable names are descriptive and clear enough. This allows other programmers to have an idea about what the variable is representing.

Assigning Values to Variables

- The general format for assigning values to variables is as follows:

variable_name = ***expression***

```
number = 100
>>> print(number)
100
>>> print(type(number))
<class 'int'>
>>> number =100.0
>>> print(type(number))
<class 'float'>
>>> number =" one two"
>>> print(number)
 one two
>>> print(type(number))
<class 'str'>
```

Operators

- Operators are symbols, such as +, –, =, >, and <, that perform certain mathematical or logical operation to manipulate data values and produce a result based on some rules.
- An operator manipulates the data values called operands.
- Python language supports a wide range of operators. They are
  1. Arithmetic Operators
  2. Assignment Operators
  3. Comparison Operators
  4. Logical Operators
  5. Bitwise Operators

Arithmetic Operators

Assignment Operators

- Assignment operators are used for assigning the values generated after evaluating the right operand to the left operand.
- Assignment operation always works from right to left.
- Assignment operators are either simple assignment operator or compound assignment operators.
- Simple assignment is done with the equal sign (=) and simply assigns the value of its right operand to the variable on the left.

```
>>> x=5
>>> x=x+1
>>> x
6

>>> x+=1
>>> x
7
```

Assignment Operators

Python programming language doesn’t support Auto increment (+ +) and Auto decrement (- -) operators.

Comparison Operators

- When the values of two operands are to be compared then comparison operators are used.
- The output of these comparison operators is always a Boolean value, either True or False.
- The operands can be Numbers or Strings or Boolean values.
- Strings are compared letter by letter using their ASCII values, thus, “P” is less than “Q”, and “Aston” is greater than “Asher”.

Comparison Operators

Logical Operators

- The logical operators are used for comparing or negating the logical values of their operands and to return the resulting logical value.
- The values of the operands on which the logical operators operate evaluate to either True or False. The result of the logical operator is always a Boolean value, True or False.

Logical AND Operators

Logical OR Operators

Logical NOT Operators

Logical Operators

-

Bitwise Operators

- Bitwise operators treat their operands as a sequence of bits (zeroes and ones) and perform bit by bit operation.
- For example, the decimal number ten has a binary representation of 1010.
- Bitwise operators perform their operations on such binary representations, but they return standard Python numerical values.

Bitwise Operators

Bitwise Operators

Bitwise Operators

Precedence and Associativity

- Operator precedence determines the way in which operators are parsed with respect to each other.
- Operators with higher precedence become the operands of operators with lower precedence.
- Associativity determines the way in which operators of the same precedence are parsed.
- Almost all the operators have left-to-right associativity.

```
>>> 2 + 3 * 6

>>> (2 + 3) * 6

>>> 6 * 4 / 2
```

Precedence and Associativity

Expressions with higher-precedence operators are evaluated first.

Data Types

- Python contains the following inbuilt data types
  1. int
  2. float
  3. complex
  4. bool
  5. str
  6. bytes
  7. bytearray
  8. range
  9. list
  10. tuple
  11. set
  12. frozenset
  13. dict
  14. None

Data Types

Data Types

- Data types specify the type of data like numbers and characters to be stored and manipulated within a program. Basic data types of Python are
  - Numbers
  - Boolean
  - Strings
  - None

- **Numbers:**
  - Integers, floating point numbers and complex numbers fall under Python numbers category. They are defined as **int, float and complex** class in Python.
  - Integers can be of any length; it is only limited by the memory available.
  - A floating point number is accurate up to 15 decimal places.
  - Integer and floating points are separated by decimal points.
  - Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part.

Data Types

Numbers:

- Int:
  - Int, or integer, is a whole number, positive or negative, without decimals, of unlimited length.
    - x = 1
    - y = 35656222554887711
    - z = -3255522

    - print(type(x))
    - print(type(y))
    - print(type(z))

Data Types

Numbers:

- Float:
  - Float, or "floating point number" is a number, positive or negative, containing one or more decimals.
    - x = 1.10
    - y = 12E4
    - z = -87.7e100

    - print(type(x))
    - print(type(y))
    - print(type(z))
  - Float can also be scientific numbers with an "e" to indicate the power of 10.

Data Types

- **Numbers:**
  - Complex :
    - Complex numbers are written with a "j" as the imaginary part:
      - x = 3+5j
      - y = 5j
      - z = -5j

      - print(type(x))
      - print(type(y))
      - print(type(z))

Data Types

- Type Conversion:
  - You can convert from one type to another with the int(), float(), and complex() methods:

Convert from one type to another:

```
x = 1 # int
y = 2.8 # float
z = 1j # complex

#convert from int to float:
a = float(x)

#convert from float to int:
b = int(y)

#convert from int to complex:
c = complex(x)

print(a)
print(b)
print(c)

print(type(a))
print(type(b))
print(type(c))
```

- Note: You cannot convert complex numbers into another number type.

Data Types

Data Types

Data Types

Data Types

Data Types

- **Boolean:**
  - Booleans –useful in conditional statements. since a condition is really just a yes-or-no question, the answer to that question is a Boolean value, either True or False.
  - In programming you often need to know if an expression is True or False.
  - You can evaluate any expression in Python, and get one of two answers, True or False.
  - When you compare two values, the expression is evaluated and Python returns the Boolean answer:
    - print(10 > 9)
    - print(10 == 9)
    - print(10 < 9)

Data Types

Strings:

- A string consists of a sequence of one or more characters, which can include letters, numbers, and other types of characters.
- A string can also contain spaces.
- You can use single quotes or double quotes to represent strings and it is also called a string literal.
- Multiline strings can be denoted using triple quotes, ''' or """. These are fixed values, not variables that you literally provide in your script.
  - print('ECE')
  - print("DSCE")
  - print("'ECE DSCE'")
  - s=(''' ECE
  - DSCE
  - Bengaluru ''')
  - print(s)
  - print("' ECE 'DSCE‘ Bengaluru'")

Data Types

- Strings are Arrays

- Square brackets can be used to access elements of the string.
- Get the character at position 1 (remember that the first character has the position 0):

```
a = "Hello, World!"
print(a[1])
```

- Slicing

Data Types

Slicing

- You can return a range of characters by using the slice syntax.
- Specify the start index and the end index, separated by a colon, to return a part of the string.
  - Get the characters from position 2 to position 5 (not included):

```
>>>b = "Hello, World!"
>>>print(b[2:5])
```

- Negative Indexing

```
>>> b = "Hello, World!"
>>> print(b[-5:-2])
```

Data Types

- Slicing
- String Length
  - To get the length of a string, use the len() function.

```
>>> a = "Hello, World!"
>>> print(len(a))
```

- String Methods
  - The strip() method removes any whitespace from the beginning or the end:

```
>>> a = "  Hello, World! "
>>> print(a.strip()) # returns "Hello, World!“
```

  - The lower() method returns the string in lower case:

```
>>> a = "Hello, World!"
>>> print(a.lower())
```

Data Types

- Slicing
- The upper() method returns the string in upper case:

```
>>> a = "Hello, World!"
>>> print(a.upper())
```

- The replace() method replaces a string with another string:

```
>>> a = "Hello, World!"
>>> print(a.replace("H", "J"))
```

- The split() method splits the string into substrings if it finds instances of the separator:

```
>>> a = "Hello, World!"
>>> print(a.split(",")) # returns ['Hello', ' World!']
```

- Use the format() method to insert numbers into strings:

```
>>> age = 36
>>> txt = "My name is John, and I am {}"
>>> print(txt.format(age))
```

Data Types

- **None:**
  - None is another special data type in Python. None is frequently used to represent the absence of a value. For example,

```
>>> money = None
```

Indentation

- Indentation in Python is used to create a group of statements. Many popular languages such as C, and Java uses braces ({ }) to define a block of code, Python use indentation.
- In Python, Programs get structured through indentation.
- This principle makes the code look cleaner and easier to understand and read.
- When writing python code, we have to define a group of statements for functions and loops. This is done by properly indenting the statements for that block.
- The leading whitespaces (space and tabs) at the start of a line is used to determine the indentation level of the line. You have to increase the indent level to group the statements for that code block. Similarly, reduce the indentation to close the grouping.
- Usually, **four whitespaces are used for indentation** and are preferred over tabs.

Indentation

```
def foo():
    print("Hi")
    if True:
        print("true")
    else:
        print("false")
print("Done")
```

Indentation

- **Python Indentation Rules:**
  - We can’t split indentation into multiple lines using backslash.
  - The first line of Python code can’t have indentation, it will throw IndentationError.
  - You should avoid mixing tabs and whitespaces to create indentation. It’s because text editors in Non-Unix systems behave differently and mixing them can cause wrong indentation.
  - It is preferred to use whitespaces for indentation than the tab character.
  - The best practice is to use 4 whitespaces for first indentation and then keep adding additional 4 whitespaces to increase the indentation.

Comments

- Comments are an important part of any program. A comment is a text that describes what the program or a particular part of the program is trying to do and is ignored by the Python interpreter.
- In Python, use the **hash (#)** symbol to start writing a comment.
- Comments are used to help you and other programmers understand, maintain, and debug the program.

Single Line Comment

```
#This is single line Python comment
```

Multiline Comments

```
#This is
#multiline comments
#in Python
```

Another way of doing this is to use triple quotes, either ''' or """.

```
'''This is
   multiline comment
   in Python using triple quotes'''
```

Reading Input

- In Python, input() function is used to gather data from the user. The syntax for input function is,

  variable_name = input([prompt])

- prompt is a string written inside the parenthesis that is printed on the screen.
- The prompt statement gives an indication to the user of the value that needs to be entered through the keyboard.
- When the user presses Enter key, the program resumes and input returns what the user typed as a string.
- Even when the user inputs a number, it is treated as a string which should be casted or converted to number explicitly using appropriate type casting function.

```
>>> person = input("What is your name?")
What is your name? Carrey
>>> person
'Carrey'
```

Print Output

- The print() function allows a program to display text onto the console.
- The print function will print everything as strings and anything that is not already a string is automatically converted to its string representation. For example,

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

- Even though there are different ways to print values in Python, there are two major string formats which are used inside the print() function to display the contents onto the console as they are less error prone and results in cleaner code.

1. str.format()

2. f-strings

Print Output

- **f-strings:**
  - Formatted strings or f-strings were introduced in Python 3.6.
  - A f-string is a string literal that is prefixed with “f”.
  - You need to specify the name of the variable or the expression itself inside the curly braces {} to display its value.

Print Output

- **Program 1**

Type Casting

- Process of converting one type to another type is called type casting or type Conversions.
  - int() - constructs an integer number from an integer literal, a float literal (by rounding down to the previous whole number), or a string literal (providing the string represents a whole number)
  - float() - constructs a float number from an integer literal, a float literal or a string literal (providing the string represents a float or an integer)
  - str() - constructs a string from a wide variety of data types, including strings, integer literals and float literals

Type Casting

- int():

```
>>> x = int(1)   # x will be 1
>>> y = int(2.8) # y will be 2
>>> z = int("3") # z will be 3
```

- float()

```
>>> x = float(1)     # x will be 1.0
>>> y = float(2.8)   # y will be 2.8
>>> z = float("3")   # z will be 3.0
>>> w = float("4.2") # w will be 4.2
```

- str()

```
>>> x = str("s1") # x will be 's1'
>>> y = str(2)    # y will be '2'
>>> z = str(3.0)  # z will be '3.0'
```

Type Casting

The chr() Function

- Convert an integer to a string of one character whose ASCII code is same as the integer using chr() function.
- The integer value should be in the range of 0–255.

Type Casting

The complex() Function

- Use complex() function to print a complex number with the value real + imag*j or convert a string or number to a complex number.
- If the first argument for the function is a string, it will be interpreted as a complex number and the function must be called without a second parameter.
- The second parameter can never be a string. Each argument may be any numeric type (including complex).
- If imag is omitted, it defaults to zero.
- If both arguments are omitted, the complex() function returns 0j.

Type Casting

The ord() Function

- The ord() function returns an integer representing Unicode code point for the given Unicode character.
- Display equivalent ASCII value.

Type Casting

The type() Function

- The syntax for type() function is,

  type(object)

- The type() function returns the data type of the given object.

```
>>> type(1)
<class 'int'>
>>> type(6.4)
<class 'float'>
>>> type("A")
<class 'str'>
>>> type(True)
<class 'bool’>
```

Type Casting

- is and is not Identity Operators
  - The operators **is** and **is not** are identity operators.
  - Operator **is** evaluates to **True** if the values of operands on either side of the operator point to the same object and False otherwise.
  - The operator **is not** evaluates to False if the values of operands on either side of the operator point to the same object and True otherwise.

```
>>> x = "Seattle"
>>> y = "Seattle"
>>> x is y
True

>>> place = "paris"
>>> place is not "USA"
True
```

Type Casting

Type Casting

 write a python program for checking the given number is

- ✔ **Odd, even,**
- ✔ **Prime,**
- ✔ **Positive, Negative**
- ✔ **Display using formatted strings**
- ✔ **The ASCII value,**
- ✔ **In other number systems**
- ✔ **Type and Address of the number**
- ✔ **Unpacked decimal**
- ✔ **Swap the digits**

Control Flow Statements

- LEARNING OUTCOMES
- At the end of this chapter, you are expected to
  - Use if, if…else and if…elif…else statements to transfer the control from one part of the program to another.
  - Write while and for loops to run one or more statements repeatedly.
  - Control the flow of execution using break and continue statements.
  - Improve the reliability of code by incorporating exception handling mechanisms through try-except blocks.

Control Flow Statements

- The control flow statements in Python Programming Language are

- **Sequential Control Flow Statements:** This refers to the line by line execution, in which the statements are executed sequentially, in the same order in which they appear in the program.

- **Decision Control Flow Statements:** Depending on whether a condition is True or False, the decision structure may skip the execution of an entire block of statements or even execute one block of statements instead of other (if, if…else and if…elif…else).

- **Loop Control Flow Statements:** This is a control structure that allows the execution of a block of statements multiple times until a loop termination condition is met (*for* loop and *while* loop). Loop Control Flow Statements are also called Repetition statements or Iteration statements.

Control Flow Statements

The if Decision Control Flow Statement

- The if Decision Control Flow Statement

  The syntax for if statement is

You don’t need to use the == operator explicitly to check if a variable’s value evaluates to True as the variable name can itself be used as a condition.

The if Decision Control Flow Statement

- Program 1

The if Decision Control Flow Statement

- Program 2

The if….else Decision Control Flow Statement

- The if….else Decision Control Flow Statement

  The syntax for if….else statement is

The if….else Decision Control Flow Statement

- Program 1

The if….else Decision Control Flow Statement

- Program 2

The if…elif…else Decision Control Flow Statement

- The if…elif…else Decision Control Flow Statement

  The syntax for if…elif…else statement is

The if…elif…else Decision Control Flow Statement

- The if…elif…else Decision Control Flow Statement
  - This *if…elif…else* decision control statement is executed as follows:
  - In the case of multiple Boolean expressions, only the first logical Boolean expression
    which evaluates to True will be executed.
  - If Boolean_Expression_1 is *True*, then statement_1 is executed.
  - If Boolean_Expression_1 is *False* and Boolean_Expression_2 is *True*, then statement_2 is executed.
  - If Boolean_Expression_1 and Boolean_Expression_2 are *False* and Boolean_Expression_3 is *True*, then statement_3 is executed and so on.
  - If none of the Boolean_Expression is *True*, then statement_last is executed.

The if….else Decision Control Flow Statement

- Program 1

The if….else Decision Control Flow Statement

- Program 1

The if….else Decision Control Flow Statement

- Program 2

Nested if Statement

- In some situations, you have to place an if statement inside another statement.
- An if statement that contains another if statement either in its if block or else block is called a Nested if statement.
- The syntax of the nested if statement is,

If the Boolean_Expression_1 is evaluated to *True*, then the control shifts to Boolean_Expression_2

and if the expression is evaluated to *True*, then statement_1 is executed,

if the Boolean_Expression_2 is evaluated to *False* then the statement_2 is executed.

If the Boolean_Expression_1 is evaluated to *False,* then statement_3 is executed.

Nested *if* Statement

Program 1

Program to Check If a Given Year Is a Leap Year

```
year = int(input('Enter a year'))
if year % 4 == 0:
if year % 100 == 0:
if year % 400 == 0:
print(f'{year} is a Leap Year')
else:
print(f'{year} is not a Leap Year')
else:
print(f'{year} is a Leap Year')
else:
print(f'{year} is not a Leap Year')
```

Nested *if* Statement

Program 1

Program to Check If a Given Year Is a Leap Year

- All years which are perfectly divisible by 4 are leap years except for century years (years ending with 00) which is a leap year only it is perfectly divisible by both 100 and 400.
- For example, years like 2012, 2004, 1968 are leap years but 1971, 2006 are not leap years. Similarly, 1200, 1600, 2000, 2400 are leap years but 1700, 1800, 1900 are not.

The while Loop

- A **while** loop statement in Python programming language repeatedly executes a target statement as long as a given condition is true.
- The syntax for while loop is,

```
count = 0
while (count < 9):
   print ('The count is:', count)
   count = count + 1

print "Good bye!"
```

The *for* Loop

- A for loop is used for iterating over a sequence (that is either a list, a tuple, a dictionary, a set, or a string).
- The syntax for the *for* loop is,

```
fruits = ["apple", "banana", "cherry"]
for x in fruits:
  print(x)
```

The for loop does not require an indexing variable to set beforehand.

The *range() function*

- The range() function generates a sequence of numbers which can be iterated through using for loop.
- The syntax for range() function is,

  range([start ,] stop [, step])
- Both start and step arguments are optional and the range argument value should always be an integer.
- **start** → value indicates the beginning of the sequence. If the start argument is not specified, then the sequence of numbers start from zero by default.
- **stop** → Generates numbers up to this value but not including the number itself.
- **step** → indicates the difference between every two consecutive numbers in the sequence. The step value can be both negative and positive but not zero.

NOTE: The square brackets in the syntax indicate that these arguments are optional. You can leave them out.

The *range() function*

- Always use the ***for*** loop to traverse through the elements generated by the range() function.

The *range() function*

- Programs:

The *range() function*

- Programs:

Program to Iterate through Each Character in the String Using ***for*** Loop.

Programs:

- Write a Program to Find the Sum of All Odd and Even Numbers up to a Number Specified by the User.

The *range() function*

Programs:

- Write a Program to Find the Factorial of a Number.

The *continue* and *break* Statements

Programs:

- The ***break*** and ***continue*** statements provide greater control over the execution of code in a loop.
- Whenever the ***break*** statement is encountered, the execution control immediately jumps to the first instruction following the loop.
- To pass control to the next iteration without exiting the loop, use the ***continue*** statement.
- Both ***continue*** and ***break*** statements can be used in while and for loops.

The *continue* and *break* Statements

Programs:

- Program to demonstrate infinite while loop and break

break Statements

Programs:

```
available=10
num=int(input("Enter "))

i=1
while i<=num:
    if i>available:
        print(“out of stock”)
        break
    print("chocolates")
    i=i+1
print("Thanks")
```

Program to Demonstrate continue Statement

Programs:

```
n=10
while n>0:
    print(f"current value is{n}")
    if n==5:
        print(f"breakin at{n}")
        n=10
        continue
    n=n-1
```

Program to Demonstrate continue Statement

Programs:

Scenario-1:

```
for i in range(1,24):
    if i%3==0 or i%5==0:
        continue
    print(i)
print("Thats it")
```

Scenario-2:

```
for i in range(1,24):
    if i%3==0 or i%5==0:
        print(i)
print("Thats it")
```

PASS

- Programs:

```
for i in range(1,21):
    if (i%2==0):
        pass
    else:
        print(i)

print("Thanks")
```

Catching Exceptions Using try and except Statement

Errors:

1. Compile time error : Syntax error ex: missing colon
2. Logical Error: ex: add 2+3=4 :wrong output.
3. Run time Error:
    a. ZeroDivisionError
    b. ValueError

Catching Exceptions Using try and except Statement

- Exceptions
- Exception handling is one of the most important feature of Python programming language that allows us to handle the errors caused by exceptions.
- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it.
- Errors detected during execution are called ***exceptions.***
- An exception is an unwanted event that interrupts the normal flow of the program.
- When an exception occurs in the program, execution gets terminated. In such cases, we get a system-generated error message. However, these exceptions can be handled in Python.
- By handling the exceptions, we can provide a meaningful message to the user about the issue rather than a system-generated message, which may not be understandable to the user.

Catching Exceptions Using try and except Statement

- Exceptions
- Exceptions can be either built-in exceptions or user-defined exceptions.

Catching Exceptions Using try and except Statement

- Exception Handling Using try…except…finally
- Handling of exception ensures that the flow of the program does not get interrupted when an exception occurs which is done by trapping run-time errors.
- Handling of exceptions results in the execution of all the statements in the program.
- Run-time errors are those errors that occur during the execution of the program.
- These errors are not detected by the Python interpreter, because the code is syntactically correct.

Catching Exceptions Using try and except Statement

- try…except…finally

The else block, which, when present, must follow all except blocks. It is useful for code that must be executed if the try block does not raise an exception.
A finally block is always executed before leaving the try statement, whether an exception has occurred or not.

Catching Exceptions Using try and except Statement

- Program: Program to check for *ValueError* Exception
-

Catching Exceptions Using try and except Statement

- Program to Check for ZeroDivisionError Exception

The ***else*** block gets executed if the ***try*** block does not raise an exception.

Programs:

1. Print the grade report considering both thresholds for the grade
   and grades considered in increasing order
2. Program for the following scenario:
   consider the numbers till n(entered by the user)
   printing “CLAP” for the numbers divisible by 3
   print “Hha Ha” for the numbers divisible By 5
   print both for the numbers divisible by both
3. Program for the following conditions
   print the numbers from 1 to 46 excluding the numbers divisible by 3 , 5
   print the numbers from 1 to 46, only divisible by 3 , 5
   print the numbers from 1 to 46 divisible by both 3 and 5

The while Loop

Programs:

1. Write Python Program to Display First 10 Numbers Using while Loop Starting from 0
2. Write a Program to Find the Average of n Natural Numbers Where n Is the Input from the User.
3. Program to Find the GCD of Two Positive Numbers
4. Write Python Program to Find the Sum of Digits in a Number
5. Write a Program to Display the Fibonacci Sequences up to nth Term Where n is Provided by the User
6. Program to Repeatedly Check for the Largest Number Until the User Enters **“done”.**

The while Loop

Programs:

1. Write Python Program to Display First 10 Numbers Using while Loop Starting from 0

The while Loop

Programs:

1. Write a Program to Find the Average of n Natural Numbers Where n Is the Input from the User.

The while Loop

Programs:

1. Program to Find the GCD of Two Positive Numbers

The while Loop

Programs:

1. Write Python Program to Find the Sum of Digits in a Number

The while Loop

Programs:

1. Write a Program to Display the Fibonacci Sequences up to nth Term Where n is Provided by the User

The while Loop

Programs:

1. Program to Repeatedly Check for the Largest Number Until the User Enters **“done”.**