Programming in Python

BOOKS:

- 1. Kenneth Lambert – “Fundamentals of Python_ Data Structures”, *Cengage Learning PTR* (2013).
- 2. Gowrishankar S, Veena A, “Introduction to Python Programming”, 1st Edition, *CRC Press/Taylor & Francis*, 2018. ISBN-13: 978-0815394372.
- 3. Mark Lutz, “Programming Python”, 4th Edition, *O’Reilly Media*, 2011.ISBN-13: 978-9350232873.

REFERENCE MATERIALS:

- 1. Cody Jackson ,“Learning to Program using Python”, Second Edition, 2014.
- 2. Michael DAWSON, ”Python Programming”,3rd Edition, Course technology PTR, 2010
- 3. Charles R. Severance, “Python for Everybody: Exploring Data Using Python 3”, 1st Edition,

  CreateSpace Independent Publishing Platform, 2016.

  http://do1.drchuck.com/pythonlearn/EN_us/pythonlearn.pdf
- 4. Allen B. Downey, “Think Python: How to Think Like a Computer Scientist”, 2nd Edition, *Green Tea Press*, 2015. (http://greenteapress.com/thinkpython2/thinkpython2.pdf)

Module - 1

- **Introduction to Python Programming:** History, Application of Python, Identifiers, Keywords, Statements and Expressions, Variables, Operators, Data Types, Type Conversions.
- **Control Flow Statements:** The if, if…else, if…elif…else, Decision Control Flow Statement, Nested if Statement, The while, for Loop, The continue and break Statements,
- **Functions**: Built-In Functions, Commonly Used Modules, Function Definition and Calling the Function, The return Statement and void Function,
- **Strings:** Basic String Operations, Accessing Characters in String by Index Number, String Slicing and Joining, String Methods.

Introduction

Why the name Python??

Inspired by the TV show

The Complete Monty Python's Flying Circus

Guido van Rossum

Author of the Python programming language

Python Versions

Python released on 20-02-1991 -Ver-0.9.9

- **Python 1.0 - Jan 1994**
- Python 1.5 - 31 Dec 1997
- Python 1.5.2 - April 1999
- Python 1.6 - 05 Sep 2000
- **Python 2.0 - 16 Oct 2000**
- Python 2.0.1 - 22 Jun 2001
- Python 2.1 - 17 Apr 2001
- Python 2.2 - 21 Dec 2001
- Python 2.3 - 29 Jul 2003
- Python 2.4 - 30 Nov 2004
- Python 2.5 - 19 Sep 2006
- Python 2.6 - 01 Oct 2008
- Python 2.7 - 03 Jul 2010

- **Python 3.0 - 03 Dec 2008**
- Python 3.1 - 27 Jun 2009
- Python 3.2 - 20 Feb 2011
- Python 3.3 - 29 Sep 2012
- Python 3.4 - 16 Mar 2014
- Python 3.5 - 13 Sep 2015
- Python 3.6 - 23 Dec 2016
- Python 3.7 - 27 Jun 2018
- Python 3.8 - 14 Oct 2019

[May Not be backword compatible with 2.x]

Introduction

- File Extension - py [Command Window Execution - py prog.py or python prog.py]
- More popularity because of its simplicity, Concise code, Applications Include Machine learning, Deep Learning, Artificial Intelligence, Neural Networks, Data Science, IoT etc..
- The **Python Software Foundation** (**PSF**) is a non-profit organization devoted to the Python programming language
- Python software and few IDEs: https://www.python.org/

https://www.jetbrains.com/pycharm/

Introduction

- Python is an all rounder
  - Functional programming features from C
  - OOP features from C++
  - Scripting language features from Perl, Shell Script
    - Group of lines executed one by one

- **Use of python:**
  - Desktop Applications
  - Web Applications
  - Network Applications
  - Games Development
  - Data Analysis, Data Science
  - Machine Learning, Deep Learning, Neural Network, Artificial Language, IoT..

Features of Python

- Simple and easy to learn
- Freeware and Open Source
- High level Programming language
- Platform independent, Portable
- Dynamically typed language
- Procedure and Object Oriented
- Interpreted
- Extensible and Embedded
- Extensive Library

Limitations and Flavours of Python

Limitations of Python

- Not suitable for
  - ❑ Mobile applications
  - ❑ Enterprise applications - Banking Application, Telecom Application-End to End Support
  - ❑ Performance is low because of Interpreted nature
- Flavours of Python
  - ❑ Free ware and open source-Customised python versions
    - ✔ C-Python- C-language applications
    - ✔ Jpython/Jython- Java Applications
    - ✔ Iron Python- To work with C#, .net
    - ✔ Ruby Python- Ruby applications
    - ✔ Anaconda Python- Large volume data, ML, Data Science..
    - ✔ Stackless- Concurrent applications
    - ✔ PyPy- Python for speed – PVM+JIT(Just in Time) Compiler

Identifiers

- An identifier is a name given to a variable, function, class or module.

Identifiers may be one or more characters in the following format:

- Identifiers can be a combination (a to z) or (A to Z)or (0 to 9) or an underscore (_).
- A Python identifier can begin with an alphabet (A – Z and a – z and _).
- An identifier cannot start with a digit but is allowed everywhere else.
- Keywords or reserved words cannot be used as identifiers.
- One cannot use spaces and special symbols like !, @, #, $, % etc. as identifiers.
- **Identifier can be of any length (But not recommended to take lengthy variable)**
- **Case Sensitive**

Keywords

- Keywords are a list of reserved words that have predefined meaning.
- Keywords are special vocabulary and cannot be used by programmers as identifiers for variables, functions, constants or with any identifier name.
- Attempting to use a keyword as an identifier name will cause an error.
- 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

33 - Keywords

- False True None
- and not or is
- if elif else
- break for while continue return in yield
- try except finally raise assert
- Import from as class def pass
- nonlocal global del with lambda

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
  ```

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.*
- If a need for variable arises you need to think of a variable name based on the rules mentioned in the following subsection and use it in the program.

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

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
```

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”.

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

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.

Examples of bitwise logical 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`

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

- **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.
- The Boolean values, True and False are treated as reserved words.
- **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(“ENGG")
- print('''ECE BENGALURU''')
- s=(''' ECE
-               ENGG
-                   Bengaluru ''')
- print(s)
- print(''' ECE ‘ENGG‘ Bengaluru''')

None

- *None* is another special data type in Python. *None* is frequently used to represent the absence of a value.
- money = None

Indentation

Code blocks and indentation in Python.

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.
- Comments are used to help you and other programmers understand, maintain, and debug the program.
- Python uses two types of comments: single-line comment

#This is single line Python comment

- multiline comments.

#This is

#multiline comments

#in Python

Reading Input

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

variable_name = ***input([prompt])***

- >>> 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.
- print("Hello World!!")

```
Hello World!!
```

- Two major string formats which are used inside the *print()* function to display the contents onto the console
- 1. str.format()
- 2. f-strings

str.format() Method

- The syntax for *format()* method is, ***str.format(p0, p1, ..., k0=v0, k1=v1, ...)***

p0, p1,... are called as positional arguments and, k0, k1,... are keyword arguments with their assigned values of v0, v1,... respectively.

- Formatted strings or f-strings were introduced in Python 3.6. A *f-string* is a string literal that is prefixed with “f”. These strings may contain replacement fields, which are expressions enclosed within curly braces {}. The expressions are replaced with their values.

Example:

```
USN = input("Enter your USN: ")

Name = input("Enter your Name: ")

print("Student USN is {0} and  name {1}".format(USN, Name))

print("Student name is {1} and USN is {0}".format(USN, Name))

print("Student belongs to {Section}, ECE".format(Section = "6ABCD"))

print(f"Student USN {USN} and {Name}")
```

Type Conversions

You can explicitly cast, or convert, a variable from one type to another.

The *int()* Function
The *float()* Function
The *str()* Function
The *chr()* Function
The *complex()* Function
The *ord()* Function
The *hex()* Function
The *oct()* Function

The *type()* Function and is Operator
type(object)
The *type()* function returns the data type of the given object

```
type(6.4)
<class 'float'>
```

is , is not

- 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.
- `>>> a=2`
- `>>> b=2`
- `>>> a is b`

  True
- `>>> a is not b`

  False
- `>>> c=4`
- `>>> a is not c`

  True

Control Flow Statements

control flow statements

The control flow statements in Python Programming Language are

- 1. **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.
- 2. **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).
- 3. **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.

The *if* Decision Control Flow Statement

If the Boolean expression evaluates to *True* then statements in the *if* block will be executed; otherwise the result is *False* then none of the statements are executed.

In Python, the *if* block statements are determined through indentation and the first unindented statement marks the end.

Program Reads a Number and Prints a Message If It Is Positive

```
number = int(input("Enter a number"))
if number >= 0:
print(f"The number entered by the user is a positive number")
```

The *if…else* Decision Control Flow Sta

- If the Boolean_Expression evaluates to *True*, then statement_1 is executed, otherwise it is evaluated to *False* then statement_2 is executed.
- Indentation is used to separate the blocks.
- After the execution of either statement_1 or statement_2, the control is transferred to the next statement after the *if* statement.
- Also, *if* and *else* keywords should be aligned at the same column position.

Program to Find If a Given Number Is Odd or Even

```
 number = int(input("Enter a number"))
if number % 2 == 0:
print(f"{number} is Even number")
else:
print(f"{number} is Odd number")
```

The *if…elif…else* Decision Control 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.

Write a Program to Prompt for a Score between 0.0 and 1.0. Score Grade: >= 0.9 A, >= 0.8 B, >= 0.7 C, >= 0.6 D, < 0.6 F

```
score = float(input(Enter your score"))
if score < 0 or score > 1:
print('Wrong Input')
elif score >= 0.9:
print('Your Grade is "A" ')
elif score >= 0.8:
print('Your Grade is "B" ')
elif score >= 0.7:
print('Your Grade is "C" ')
elif score >= 0.6:
print('Your Grade is "D" ')
else:
print('Your Grade is "F" ')
```

Nested *if* Statement

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.

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')
```

The *while* Loop

With a *while* statement, the first thing that happens is that the Boolean expression is evaluated before the statements in the *while* loop block is executed.
If the Boolean expression evaluates to *False*, then the statements in the *while* loop block are never executed.
If the Boolean expression evaluates to *True*, then the *while* loop block is executed.
After each iteration of the loop block, the Boolean expression is again checked, and if it is *True*, the loop is iterated again.
Each repetition of the loop block is called an iteration of the loop.
This process continues until the Boolean expression evaluates to *False* and at this point the *while* statement exits. Execution then continues with the first statement after the *while* loop.

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

```
i = 0
while i < 10:
    print(f"Current value of i is {i}")
    i = i + 1
```

Write a Program to Find the Average of *n* Natural Numbers Where *n* Is the Input

```
number = int(input("Enter a number up to which you want to find the average "))
i = 0
sum = 0
count = 0
while i < number:
    i = i + 1
    sum = sum + i
    count = count + 1
    average = sum/count
print(f"The average of {number} natural numbers is {average}")
```

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

```
number = int(input('Enter a number'))
result = 0
remainder = 0
while number != 0:
    remainder = number % 10
    result = result + remainder
    number = int(number / 10)
print(f"The sum of all digits is {result}")
```

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

```
nterms = int(input('How many terms? '))
current = 0
previous = 1
count = 0
next_term = 0
if nterms <= 0:
    print('Please enter a positive number')
elif nterms == 1:
    print('Fibonacci sequence')
    print('0')
else:
    print("Fibonacci sequence")
while count < nterms:
    print(next_term)
    current = next_term
    next_term = previous + current
    previous = current
    count += 1
```

The *for* Loop

The *for* loop starts with *for* keyword and ends with a colon.

The first item in the sequence gets assigned to the iteration variable *iteration_variable*.

Here, *iteration_variable* can be any valid variable name.

Then the statement block is executed. This process of assigning items from the sequence to the *iteration_variable* and then executing the statement continues until all the items in the sequence are completed.

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.

Demonstrate for Loop Using *range()* Function

```
print("Only "stop" argument value specified in range function")
for i in range(9):
    print(f"{i}")
print("Both "start" and "stop" argument values specified in range function")
for i in range(2, 9):
    print(f"{i}")
print("All three arguments "start", "stop" and "step" specified in range function")
for i in range(2, 9, 3):
    print(f"{i}")
print("Iterate through Each Character in the String")
for i in “BENGALURU":
        print(i)
```

Write a Program to Find the Sum of All Odd and Even Numbers up to a Number

```
number = int(input("Enter a number"))
even = 0
odd = 0
for i in range(number):
    if i % 2 == 0:
        even = even + i
    else:
        odd = odd + i
print(f"Sum of Even numbers are {even} and Odd numbers are {odd}")
```

Write a Program to Find the Factorial of a Number

```
number = int(input('Enter a number'))
factorial = 1
if number < 0:
   print("Factorial doesn't exist for negative numbers")
elif number == 0:
   print('The factorial of 0 is 1')
else:
   for i in range(1, number + 1):
   factorial = factorial * i
   print(f"The factorial of number {number} is {factorial}")
```

The *continue* and *break* Statements

- 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.

Program to Demonstrate Infinite *while* Loop and *break*

```
n = 0
while True:
    print(f"The latest value of n is {n}")
    n = n + 1

n = 0
while True:
    print(f"The latest value of n is {n}")
    n = n + 1
    if n>5:
        print(f"The value of n is greater than 5")
        break
```

break

```
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

```
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
```

continue

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

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

print("Thanks")
```

Catching Exceptions Using try and except Statement

- There are at least two distinguishable kinds of errors:

1. Syntax Errors
2. Exceptions

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.

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

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.

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.

Program to check for *ValueError* Exception

Program

- Program to Check for ZeroDivisionError Exception

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

Try except finally

#compile time error- syntax error, #logical error-getting wrong answers, #runtime error-6/0-dividebyzero error

```
a=5
b=0
try:
    print("resource open")
    print(a/b)
    k=int(input("Enter "))
    print(k)
except ZeroDivisionError as e:
    print(" cant divide by zero",e)
except ValueError as v:
    print(" invalid input",v)
except Exception as p:
    print(" oops: something wrong",p)
finally:
    print("resource closed")
print("thanks")
```

Write a Program Which Repeatedly Reads Numbers Until the User Enters 'done'. Once 'done' Is Entered, Print Out the Total, Count, and Average of the Numbers. If the User Enters Anything Other Than a Number, Detect Their Mistake Using try and except and Print an Error Message and Skip to the Next Number

Program it:

- print the grade report considering both thresholds for the grade and grades considered in increasing order
- program for the following scenario:

  print all the numbers till n(entered by the user) but ..

  print “CLAP” for the numbers divisible by 3

  print “Hha Ha” for the numbers divisible By 5

  print both for the numbers divisible by both
- 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

Program

- Generate Fibonacci numbers between the interval (m , n) and display the count also.
- check whether given number
  - belongs to Fibonacci series, if so mention its position in the series
  - is Armstrong number.
  - Is perfect number

- Python Program to Read a Number n And Print the Series "1+2+…..+n= “
- Program to accept 5 digits and print all possible combinations from these digits
- Python Program to Check if given input is a Palindrome
- Count for vowels and consonants in the given alphabets

- Program for the following:Enter the numbers till the user enters word “end”

  Count number of odd, even, positive, negative, prime, Fibonacci numbers in the entered numbers.

Armstrong number.

- Start with the procedure for testing to see if a number is an Armstrong number. Determine how many digits are in the number. Call that n. Then take every digit in the number and raise it to the n power. Add all those together, and if your answer is the original number then it is an Armstrong number. All single digit numbers are Armstrong numbers.
- For example, test 6. One digit. 6^1 = 6. Armstrong number.
- Test 371. Three digits. 3^3 + 7^3 + 1^3 = 27 + 343 + 1 = 371. Armstrong number.
- Test 1634. Four digits. 1^4 + 6^4 + 3^4 + 4^4 = 1 + 1296 + 81 + 256 = 1634. Armstrong number.

STRONG number.

- If the sum of factorial of the digits in any number is equal the given number then the number is called as STRONG number.

  Ex=1! +4! +5!= 1+24+120 = 145

perfect number.

- **In number theory, a perfect number is a positive integer that is equal to the sum of its positive divisors, excluding the number itself. For instance, 6 has divisors 1, 2 and 3 (excluding itself), and 1 + 2 + 3 = 6, so 6 is a perfect number.**

Patterns