* **What is Python?**

Ans: Pyhton is a general purpose high level programming language.

High Level Programming language means it is Machine Independent. The code is Hardware independent.

* **Where can be python used?**

Ans: It can be used for

* Console app
* Desktop Application
* Web app
* Mobile app
* Machine Learning
* IOT applications.

**About Python:**

* Very simple and straight forward syntax. Only one process is followed for a particular work.
* Python is case sensitive.
* It is an Object Oriented Language.
* Dynamically Typed: Variables’ data type doesn’t need to be declared which is the case for C and C++
* Indentation is used in place of curly braces, if we have given four spaces to open a braces then while closing also we need to use four spaces to represent that the bracket is closed.
* Use variable without declaration.
* Interpreted language.
* Automatic memory management.
* Large library
* Multi-Paradigm Programming language(Object oriented, structured oriented and Procedural etc)
* With the python interactive interpreter it is easy to check python commands for even a single line.
* Platform independent.
* Python was developed by Guido Van Rossum while working in national research institute in Netherlands.
* SETL>>ABC>>Python.
* Python Language was first published in 1991, however the first version was launched in 1994

**Python 2 vs Python 3**

|  |  |
| --- | --- |
| 1. Division operator work like C   3/2=1  3.0/2=1.5  3/2.0=1.5  3.0/2.0=1.5 | Division operator  3/2=1.5 (here it works mathematically) |
| 1. Print “hello”   Print(“hello”)  Python 2 has two process to print a statement. | Print(“hello”)  Python 3 has only one way to print a statement |
| 1. Implicit string type is ASCII, explicit support for UNICODE | Implicit string type is UNICODE |
| 1. Range() and xrange() | Range() does the work of both. |
| 1. Exception Handling | As keyword is required |
|  | raw\_input() is no more existing, input() function has the same behavior as raw\_input().  Here type casting will be needed. |

**Comments in python**

* Comments in python scripts are statements that we want it to be ignored by interpreter.
* We put comments to make code more understandable.
* Comments can be categorized into two types:

1. To indicate what a section of code does
2. Multi-line comment serves as documentation for others.

* Single line comment are simply created by beginning a line with hash(#) character
* Multiline comments are created by adding delimiter (‘’’’’’) on each end of the comment.

**Variables in Python**

**Topics:-**

1. **Identifiers.**
2. **What is variable**
3. **Variable declaration**
4. **Naming rules**
5. **Dynamically typed**
6. **Memory concept deep dive**
7. **Identifiers: -** The smallest identifying unit in the program is called identifier.

**(**It is same as a meaningful word in a meaningful sentence)

An identifier can denote various entities like variables, types, labels, subroutines or functions, class, packages and so on.

1. **Variables:** Variables are nothing but reserved memory locations to store values, which means when we create variable we reserve some space in memory.

Variables are name which refers to some memory location, which is needed for data processing.

The amount of data we need will decide how many variables are needed in program.

While the program is running, variables are accessed and sometimes changed i.e; a new value will be assigned to the variable.

**Variable Declaration:**

* We do not need to declare the type of variable before using them, or declare their data type
* If there is a need of a variable, we can think of a name and start using it as a variable.

**Variable naming Rules:**

1. Variable name can be a combination of alphabets, digits and underscore(Leaving those no other special character can be used)
2. Variable name cannot start with digit.
3. Variable names are case sensitive. (If we have declared some variable name as X and another variable as x then both the variables will be treated differently)
4. Keywords cannot be used as a variable names.

**Variables are dynamically typed**

1. Not only value of the variable may change during execution but also the type as well.
2. We can assign an integer value to a variable, use it as an integer for a while and then assign a string value to the variable.

**Object:**

Object is something which is capable to store set of values and invoke set of operations. Every variable in python is object.

**Memory concept of variable in Python:**

When we declare a variable let us say x then x carries the reference value of a block which will contain actual value of x, however if another variable y is made equivalent to x then the reference value will be same on the other hand if we assign some other value to y the the reference value will be changed.

To understand it we can write a code and print the ID of the x .

**Program to understand the memory concept of variables in python:**

#Dummy program to check type and ID

x=5.6

print (type(x))

x=5

print (type(x))

x="amit"

print (type(x))

"""the below code is to understand the memory concept"""

a=5

print (id(a))

b=a

print (id(b))

b=3

print(id(a))

print(id(b))

**Output:**

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>>

RESTART: C:/Users/654268/Desktop/Python Programming/Print data type and ID.py

<class 'float'>

<class 'int'>

<class 'str'>

1712415904

1712415904

1712415904

1712415872

>>>

**Keywords:**

**Q) What is Keyword?**

* Keywords are the predefined words or reserved words.
* They cannot be used as variable name as their meaning is already reserved.
* If we keep the variable name as any keyword then interpreter will not be able to decide or differentiate between keywords and variable, hence it is advisable not to use the variable name as keyword.
* There are 33 keywords in python.

**Q) How to see the list of keywords in python?**

#Program to see the list of keywords in python

import keyword

print(keyword.kwlist)

**Output:**

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>>

== RESTART: C:/Users/654268/Desktop/Python Programming/3 Keyword import.py ==

['False', 'None', 'True', 'and', 'as', 'assert', 'async', 'await', 'break', 'class', 'continue', 'def', 'del', 'elif', 'else', 'except', 'finally', 'for', 'from', 'global', 'if', 'import', 'in', 'is', 'lambda', 'nonlocal', 'not', 'or', 'pass', 'raise', 'return', 'try', 'while', 'with', 'yield']

>>>

**Few important keywords:**

* True
* False These 3 keywords are the only keywords of python for

Which the first letter is in uppercase.

* None

These keywords are used in the form of values. For example we can keep the value of a variable as true.

**Data Types:**

* **Data type**
* **Numbers and Strings**
* **List of all built in data types.**

**Data type:** Data type is a category of data. In python there are total 14 in built data type.

There are three built in data type in python for numbers:

* Integers(int)
* Floating point numbers (float)
* Complex numbers (complex)

**String:** Strings are marked by quotes.

* Wrapping with single quote. ‘amit’
* Wrapping with double quote. “amit”
* Wrapping with three quotes, using either single quotes or double quotes. ‘’’amit’’’ or “””amit”””

**Boolean:** The data type is bool, it can store one of the two values either true or false.

**List of data types:** int, float, complex, bool, str, bytes, bytearray, range, list, tuple, set, frozenset, dict, NoneType.

**Type conversion and Conversion functions:**

**Q) Why do we need to convert type?**

Suppose we are assigning x as 5 and then y as “123” which is a string type.

As operations if we want to add x and y which is not possible as both the data types are different to each other and if we try to add in python then it will throw error.

Here we see the need of type conversion so that the string can be converted into int type.

**Write a program to show type conversion and also show the error if conversion is not done.**

x=5

y='123'

print (type(x))

print (type(y))

x+y

Output with error:

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>>

= RESTART: C:/Users/654268/Desktop/Python Programming/4. type conversion.py =

<class 'int'>

<class 'str'>

Traceback (most recent call last):

File "C:/Users/654268/Desktop/Python Programming/4. type conversion.py", line 7, in <module>

x+y

TypeError: unsupported operand type(s) for +: 'int' and 'str'

Correct program:

x=5

y='123'

print (type(x))

print (type(y))

print (int(y)+x) #type conversion is done here; syntax is datatype(variable)

Output:

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>>

= RESTART: C:/Users/654268/Desktop/Python Programming/4. type conversion.py =

<class 'int'>

<class 'str'>

128

>>>

**Conversion Functions:**

int() #returns int

float() #returns float

ord() #returns Unicode

str() #returns string

complex()

dict() #convert into dictionary

list()

set()

P.S: all the conversion function we will cover gradually.

**Handling number systems in python**

**Topics**:

* **Number system**
* **How to represent different number system in python script.**
* **How to print values in desired number system**

**Number System:**

Decimal: 0 to 9 (571)10

Binary: 0 and 1 (10100011)2

Octal: 0 to 7 (432)8

Hexadecimal: 0 to 9 and ABCDEF (3FA2)16

**Representation of number system in python:**

Decimal: x=45

Binary: x=0b101 or 0B101

Octal: x=0o42 or 0O42

Hexadecimal: 0x3f or 0X3f or 0x3F or 0X3F

**Python program to show the conversion of numbers into binary:**

''' program to show the number system conversion'''

x=0b10101 #binary to decimal

print(x)

x=0B10101 #binary to decimal(another representation)

print(x)

y=0B1110101 #binary to decimal with another example

print(y)

y=0b1110101

print(y)

z=0o165 #octal to decimal conversion

print(z)

z=0O165

print(z)

z=0o334 #octal to decimal conversion with another example

print(z)

z=0O334

print(z)

w=0x21c #hexa decimal to decimal conversion

print(w)

w=0X21c

print(w)

v=0x16 #hexadecimal to decimal conversion with another example.

print(v)

'''python by default convert any form of numbers into decimal'''

**Output:**

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>>

= RESTART: C:\Users\654268\Desktop\Python Programming\6 number conversion.py =

21

21

117

117

117

117

220

220

540

540

22

>>>

**Python program to show any conversion.**

'''any number to any number system conversion'''

x=45

print(bin(x)) #bin function to convert decimal to binary.

y=0o334

print(bin(y)) #bin function to convert octal to binary.

z=0x21c

print(bin(z)) #bin function to convert hexadecimal to binary.

w=0x21c

print(oct(w)) #oct function to convert hexadecimal to octal.

v=0b1000011100

print(oct(v)) #oct function to convert binary to octal.

u=540

print(hex(u)) #hex function to convert decimal to hexadecimal.

t=0b1000011100

print(hex(t)) #hex function to convert binary to hexadecimal.

s=0o1034

print(hex(s)) #hex function to convert octal to hexadecimal.

'''python by default print any number in decimal format, hence we don’t need to

Use function to convert into decimal'''

'''now suppose if we want to print the exact octal or hexa format then we need to do slicing

Using the indexing concept'''

r=oct(s)

print(r[2::])

**Output:**

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>>

RESTART: C:/Users/654268/Desktop/Python Programming/6.1 number conversion from any to any.py

0b101101

0b11011100

0b1000011100

0o1034

0o1034

0x21c

0x21c

0x21c

1034

>>>

**Operators in Python:**

* Operators are special symbols in python that carry out arithmetic and logical computation.
* Operator requires operands (data) to perform its job.

**List of operators in python:**

* Arithmetic operator
* Relational operator
* Logical operators
* Bitwise operators.
* Assignment operators
* Identity operators
* Membership operators.

**p.s:** There is no conditional operator and increment and decrement operator in python.

Work of conditional operator can be done in python by if-else statement.

**Arithmetic operators:**

1. Exponent operator, which is represented by \*\*

It calculates the power of a number.

For example:

>>> 3\*\*2

9

* It works for the negative power also

For Example:

>>> 3\*\*-2

0.1111111111111111

>>> 4\*\*-2

0.0625

>>> 10\*\*-2

0.01

>>>

* It works with the negative value also

For Example:

>>> -2\*\*3

-8

**P.S**: value will be always negative even if the power is even.

* It work with the power of float or point values

For Example:

>>> 4 0.5

2.0

>>>

* It also works if the value is kept float

>>> 2.5\*\*2

6.25

>>>

* There is no limit of range value; int type can contain large data.

For example:

>>> 10\*\*20

100000000000000000000

>>> 2\*\*128

340282366920938463463374607431768211456

>>>

1. Divide operator:
2. Divide(float) represented by /
3. Divide(floor) represented by //
4. Divide (float) always do the actual mathematical calculation and returns the float value always.

For Example:

>>> 5/2

2.5

>>> 4/2

2.0

>>>

**P.S:** it doesn’t matter even if we take bot the value as integer or float, result will always be in float.

1. Divide (floor) always return the lower value (floor value). If we divide the numbers in which any of them are float then it will return the float value on contrary if both the values are integer then it will return the integer value.

For Example:

>>> 5//2

2

>>> 5.0//2

2.0

>>> 5/2.0

2.5

>>> 5.0//2.0

2.0

>>>

>>> 5//-2

-3

>>> -5//2

-3

>>>

**P.S:** unlike modulas operator here it does not follow the sign of numerator or denominator rule. If any of them is negative then the result will be negative, however the logic will be same.

1. Modulas operator:

Generally it returns the remainder value. However there are some logic when we take negative values.

For example:

>>> 4%2

0

>>> 4%3

1

>>> 4%-3

-2

>>> 11%-7 If the denominator is negative then the result will have

-3 to be negative. So when 11%-7 comes, we cannot multiply

>>> 14%-11 it with 1 as that would result to positive remainder, hence

-8 we need to take a higher value so that the remainder comes

>>> 37%-11 negative.

-7

>>> -4%3

2 similarly when denominator is +ve but the numerator is

>>> -11%7 -ve then the same logic as above is applied, the result

3 should be positive, hence when divided by –ve value we

>>> -14%11 need to divide in such a way that the result is positive.

8

>>> -37%11

7

>>> -4%-3

-1 when both the numerator and denominator is -ve

>>> -11%-7 in that case normal division happens and the

-4 result will be in negative

>>> 3.5%2

1.5 in python 3 modulas operator doesn’t throw error for

>>> 3.5%-2 float values, instead it works with the same logic and

-0.5 delivers the result.

>>> 3.0%-2

-1.0

>>> 3%-2 so magnitude wise the result are same if both the numerator &

-1 denominator is negative or positive.

>>> -3.5%2 same in case if anyone is negative, magnitude wise result is

0.5 same, the difference is of only sign which is according to the

Denominator’s sign.

1. Multiply operator:

It does the normal multiplication of any number integer or float or combination of both.

For Example:

>>> 3\*4

12

>>> 3\*-4

-12

>>> -3\*-4

12

>>> -3\*-4\*-6

-72

>>> -3\*-4\*-5\*-6

360

>>> 3.5\*1.5

5.25

>>> 3.5\*-1.5

-5.25

>>> -3.5\*1.5

-5.25

>>>

One thing special about the \* operator is that it can return the string value if one of the operand is positive integer and the other is string. It repeats the string value when multiplied. However float cannot be taken and both the operand should not be string.

For Example:

>>> 'abc'\*2

'abcabc'

>>> 'abcdefg'\*5

'abcdefgabcdefgabcdefgabcdefgabcdefg'

>>> 'abc'\*2.5

Traceback (most recent call last):

File "<pyshell#11>", line 1, in <module>

'abc'\*2.5

TypeError: can't multiply sequence by non-int of type 'float'

1. Addition and Subtraction:

It does the normal addition and subtraction with any kind of numbers. The key feature of addition operator is that if both the operand is string type then the + operator simply concatenate and returns the string value.

For Example:

>>> 3+'abc'

Traceback (most recent call last):

File "<pyshell#0>", line 1, in <module>

3+'abc'

TypeError: unsupported operand type(s) for +: 'int' and 'str'

>>> 'abc'+'def'

'abcdef'

>>>

**P.S: Precedence of the operator**

**\*\*> / // \* % > + > -**

**If all comes in a single expression then we need to operate it from Left to Right.**

**Relational (Comparison) Operators:**

|  |  |  |
| --- | --- | --- |
| **Operators** | **Expression|Example** |  |
| **>** | 3>2 | **Greater Than** |
| **<** | 10<5 | **Less Than** |
| **>=** | 5>=2 | **Greater than or equal to** |
| **<=** | 6<=5 | **Less than or equal to** |
| **==** | 10==4 | **Equal to** |
| **!=** | 5!=4 | **Not equal to** |

* Relational operator always yields true or false.

For Example:

>>> 3>5

False

>>> 4>=4

True

>>> 10<=7

False

>>> 2==2

True

>>>

* When truth value is converted to int, it becomes 1 or 0.

For Example:

>>> x=True

>>> y=int(x)

>>> y

1

>>> int(False)

0

>>>

Now this was for Boolean to int conversion, when we go for int to Boolean conversion then there is only two concept: zero and non-zero numbers.

So basically every non-zero value is true and zero is false, For Example:

>>> bool(-1)

True

>>> bool(4)

True

>>> bool(0)

False

>>>

* Relational operator can be used for comparison of strings also

For Example:

>>> 'abc'=='def'

False

>>> 'acb'=='adf'

False

>>> 'acb'>='adf'

False

>>> 'acb'>'adf'

False

>>> 'dbc'>'abc'

True

In case of string it only compares the first letter as per the UNICODE which is 97 for a. For complex numbers only two relational operator works that is == and != rest all relational operator throws error.

For Example:

>>> 3+4j>3+5j

Traceback (most recent call last):

File "<pyshell#12>", line 1, in <module>

3+4j>3+5j

TypeError: '>' not supported between instances of 'complex' and 'complex'

>>> 3+4j>=3+5j

Traceback (most recent call last):

File "<pyshell#13>", line 1, in <module>

3+4j>=3+5j

TypeError: '>=' not supported between instances of 'complex' and 'complex'

>>> 3+4j<=3+5j

Traceback (most recent call last):

File "<pyshell#14>", line 1, in <module>

3+4j<=3+5j

TypeError: '<=' not supported between instances of 'complex' and 'complex'

>>> 3+4j<3+5j

Traceback (most recent call last):

File "<pyshell#15>", line 1, in <module>

3+4j<3+5j

TypeError: '<' not supported between instances of 'complex' and 'complex'

>>> 3+4j==3+5j

False

>>> 3+4j!=3+5j

True

>>>

* Relational operator can be used with Boolean type of values and int and float type of values.

For Example:

>>> True+5

6

>>> True+5.5

6.5

>>> True-5

-4

>>> 5-True

4

>>>

It considers the value of true as 1.

* If there are more than one relational operator which is called as operator chaining then in python it shows the result mathematically, however if there is any one false relation then the entire operator chaining will result false.

For Example:

>>> 6>5>4>3>2

True

>>> 6>5>4>3>7

False

>>>

* We cannot use all relational operator for comparing string and integer, we can use only equality operator

For Example:

>>> 'abc'>5

Traceback (most recent call last):

File "<pyshell#24>", line 1, in <module>

'abc'>5

TypeError: '>' not supported between instances of 'str' and 'int'

>>>

**One thing that can be noted is that expression will never throw error for equality operator, equality operator simply checks the content and type on both the sides and shows the result, For Example:**

>>> 3+4j==3+4j

True

>>> 5=='5'

False

As stated assignment operator checks both type and content, hence in the above expression, types were different one was string and the other was integer, hence the result is false.

Suppose we want to compare the character with their UNICODE, in that case we can cannot compare as we use to do in c or c++. Lets try the method of c/c++ once

>>> 'a'==97

False

>>>

But as per c/c++ it should be positive. In python we need to convert the character in UNICODE standards by using the function ord. For Example:

>>> ord('a')==97

True

>>>

**ord**  is a function in python which converts the character to its UNICODE value as shown below:

>>> x=ord('a')

>>> x

97

>>> y=ord('A')

>>> y

65

>>>

**One exception for the equality operator is that it can be compared between int and float values and also between Boolean and int values. Here type is over looked.**

For Example:

>>> 5==5.0

True

>>> True==1

True

>>> False==0

True

>>> True==1.0

True

>>> True<5

True

>>> False>-1

True

>>> relational operator will work between int, float and Boolean.

**Logical Operators:**

|  |  |  |
| --- | --- | --- |
| **Operators** | **Expression|Example** |  |
| and | a>b and a>c | When both operands are true, result will be true. |
| or | a>b or a>c | When both operands are false, result will be false |
| not | Not a>b | Invert the result. |

* “and” and “or” operator is binary operator as it is done between two expression.
* “not” is a unary operator.

**“and” operator**

* True and True = True
* True and False= False
* False and \*(whatever it is) = False

**“or” operator**

* False or False = False.
* False or True= True.
* True or \*(whatever it is)= True.

**P.S: Logical operator will return true or false when the operand’s of “and” and “or” is made by the help of relational operator, however if we apply logical operators without the help of relational operator then the result/output is not true or false.**

**For Example:**

>>> 3>4 and 5>6

False

>>> 3<4 and 5<6

True

>>> 3>4 and 5<6

False

>>> 3>4 or 5>6

False

>>> 3<4 or 5>6

True

>>> 5 and 6

6

>>> 5 and 4

4

>>>

As we see that if we use direct values and execute it, it return values.

Here a logic works in python, that is

* If x and y are non-Boolean then result is also non-boolean
* If x is false then result is x, otherwise y.

For Example:

>>> 3 and 3.5

3.5

>>> 4 and 'ab'

'ab'

>>> 0 and 3

0

>>>

Similarly if we put statement of ‘’or’’ then the logic/rule is

* if x is false then the result is y.

For Example:

>>> 3 or 4

3

>>> 0 or 3

3

Similarly if we put the statement of not then the value will be reverted.

For Example:

>>> 0 or False

False

>>> not 3>2

False

>>> not 0

True

>>> not 1

False

>>>

Hence unlike the ‘and’ ‘or’ operator ‘not’ operator returns the Boolean value.

One more thing to be considered is that all this logical operator work for strings also.

Before that the rule is “” or ‘’ is considered as empty string and ‘a’ or ‘’a’’ is non-empty.

For Example:

>>> "ab" and "cd"

'cd'

>>> "" and "c"

''

>>> "" or "c"

'c'

>>> not "gourav"

False

>>> not ""

True

>>> "ram" and "shyam"

'shyam'

>>> "ram" or "Shyam"

'ram'

>>>

**There are certain expression where we can get error or we can say that we cannot write the expression in that way, also there are certain way which will not give error. Following column has differentiated among them.**

|  |  |
| --- | --- |
| **Error** | **No Error** |
| 3 and x=4  We cannot assign like that. | we can use any other expression other than assignment operator like  3 and 4>2  3 and 3+4  3 and x==4( to use this we need to define x initially, else it will give error) |
| 3 and 5/0  We cannot simply write like that, however it depends which operator we are using.  The reason why 3 or 5/0 will give error because as per the logic if x is false then result is x, here x is true which means it will check the second statement that is y but 5/0 is an error in python script. | 3 or 5/0  It won’t give error because as per the logic if x is false ten result is y, hence the first statement is true and the second statement is not executed at all and it return the value as 3 as output. |
| **Summary: as per the logic if any invalid statement is executed then it will throw error.** | |

For Example:

>>> 3 or 5/0

3

>>> 3 and 5/0

Traceback (most recent call last):

File "<pyshell#19>", line 1, in <module>

3 and 5/0

ZeroDivisionError: division by zero

>>> 3 and x=4

SyntaxError: can't assign to operator

>>> 3 and 4>2

True

>>> 3 and 5/0

Traceback (most recent call last):

File "<pyshell#22>", line 1, in <module>

3 and 5/0

ZeroDivisionError: division by zero

>>> 3 or 5/0

3

>>> 3 and x==4

Traceback (most recent call last):

File "<pyshell#24>", line 1, in <module>

3 and x==4

NameError: name 'x' is not defined

>>> 3 and x=4

SyntaxError: can't assign to operator

>>> 3 and x==4

Traceback (most recent call last):

File "<pyshell#26>", line 1, in <module>

3 and x==4

NameError: name 'x' is not defined

>>> x=5

>>> 3 and x==4

False

>>>

**Bitwise operator:**

Bitwise operators are those operator which works on bits( 0 and 1). It is not the binary numbers. Suppose 1010 is binary number which has 4 different bits, basically each digit in a binary number is known as bits. The operator which works in between bits is known as bitwise operators.

**List of Bitwise Operator and their behavior:**

1. & (and operator) Behavior: 0&0 is 0, 0&1 is 0, 1&0 is 0, 1&1 is 1
2. | (or operator) Behavior: 0|0 is 0, 0|1 is 1, 1|0 is 1, 1|1 is 1
3. ~ (not operator) Behavior: ~0 is 1, ~1 is 0
4. ^ (exor operator) Behavior:0^0 is 0, 0^1 is 1, 1^0 is 1, 1^1 is 0
5. >> (right shift) Behavior:
6. << (left shift) Behavior:

So suppose if we write 5 & 6 then the binary of 5 is 101 and that of 6 is 110 hence the & operator will give the result according to the behavior of & operator

101

110\_\_

100 which is equivalent to 4

Similar calculation is done for |,~, ^ operator as per their respective behavior.

Right Shift Operator:

Suppose we are given as 25>>2 to find the result.

We need to shift the binary value of 25 twice towards right and on the place of shifting a zero will come and the least significant bit will be replaced then and the resultant value after shifting will be the result.

Let us see how to do it.

Step 1: convert the number into binary.

25 = 11001

Step 2: shift the operator.

1st shift: 01100

2nd shift: 00110 which is equivalent to 6

Left Shift Operator:

Suppose we are given 12<<3 to find the result.

We need to shift the binary value of 12 towards left and on the place of shifting a zero will come, however unlike right shift operator least significant bit will not be replaced.

Let us see how to do it.

Step 1: convert the number to binary.

12 = 1100

Step 2: Shift the operator.

1st shift: 11000

2nd shift: 110000

3rd shift: 1100000 which is equivalent to 96

**Program to understand Bitwise Operator:**

>>> 5&6

4

>>> 5 & 6

4

>>> 5|6

7

>>> ~7

-8

>>> 5^6

3

>>> 25>>2

6

>>> 12<<3

96

**Assignment operator:**

First thing which we need to know is that ‘=’ is an assignment operator whereas == is a comparison/equal to operator. X=5 means 5 has been assigned to the block x or else x has an address of the block which contains value 5, we cannot say x=5 and 5=x is same in programming language. Hence = stores the value and == compares the values, also if we are using assignment operator then make sure that the LHS is a variable, else it will throw error.

>>> x=3

>>> 3=x

SyntaxError: can't assign to literal

>>> 3=4

SyntaxError: can't assign to literal

>>>

**Compound assignment operator:**

It is also called as **assignment** operator the only thing is it is combined with another operator.

|  |  |  |
| --- | --- | --- |
| **Operator** | **Statement** | **Meaning** |
| += | X+=3 | X=x+3 |
| \*= | X\*=3 | X=x\*3 |
| /= | x/=3 | X=x/3 |
| //= | x//=3 | X=x//3 |
| %= | X%=3 | X=x%3 |
| &= | X&=3 | X=x&3 |
| |= | X|=3 | X=x|3 |
| ^= | X^=3 | X=x^3 |
| >>= | x>>=3 | X=x>>3 |
| <<= | X<<=3 | X=x<<3 |
| -= | x-=3 | X=x-3 |

To execute the above operator we need to initialize x first, else it will throw error.

>>> x=5

>>> x+=3

>>> x

8

>>> x\*=3

>>> x

24

>>> x/=3

>>> x

8.0

>>> x//=3

>>> x

2.0

>>> x%=3

>>> x

2.0

>>> x&=3

Traceback (most recent call last):

File "<pyshell#11>", line 1, in <module>

x&=3

TypeError: unsupported operand type(s) for &=: 'float' and 'int'

>>> x&=3

Traceback (most recent call last):

File "<pyshell#12>", line 1, in <module>

x&=3

TypeError: unsupported operand type(s) for &=: 'float' and 'int'

>>> x=5

>>> x&=3

>>> x

1

>>> x=6

>>> x|=3

>>> x

7

>>> x^=3

>>> x

4

**P.S: &, |, ^ all these doesn’t works with float values.**

>>> x=8.0

>>> x|=3

Traceback (most recent call last):

File "<pyshell#33>", line 1, in <module>

x|=3

TypeError: unsupported operand type(s) for |=: 'float' and 'int'

>>> x^=3

Traceback (most recent call last):

File "<pyshell#34>", line 1, in <module>

x^=3

TypeError: unsupported operand type(s) for ^=: 'float' and 'int'

>>>

**Identity Operator:**

Before understanding identity operator let us recollect some rules / basic of memory management in python.

* Every variable in python is an object.
* Objects are dynamically created.
* Objects do not have names but references.

We need to know that once we write x=5, x contains the reference value of the block which will contain 5. In C what happens that once we write x=5, a memory block x is created which contains the value 5 but in python it is just the reference value which doesn’t have any name. so next if we write y=5, then in python both x and y will have the same reference value.

For Example:

>>> x=5

>>> print(id(x))

1863607456

>>> y=5

>>> print(id(y))

1863607456

>>>

The identity operator comes into play when we want to know whether the variables are referring to the same memory location or not.

There are two identity operator:

1. is operator
2. is not operator.

As per the above example the syntax to use identity operator is **x is y**

>>> x=5

>>> y=5

>>> x==y

True

>>> x is y

True

>>>

Now the question is what is the difference between equal to operator and ‘is’ operator.

The difference is == operator compares the value 5 and 5 as per the above scenario, however ‘is’ operator compares the reference values.

Now the question is there is no difference in result, both shows True as output so what is the point of using it!

As we know that if x=5 and y=5.0 and if we try to execute x==y then the result will be true. Here the values were compared, however their reference value differs, hence the identity operator will return false value.

>>> x=5

>>> y=5.0

>>> print(id(x))

1863607456

>>> print(id(y))

91516528

>>> x is y

False

>>>x is not y

True

>>>

Hence the other identity operator is ‘is not’ which simply gives reverse value of ‘is’ operator.

**Membership operator:**

* The ‘in’ operator
* Returns True if a sequence with the specified value is present in the object.
* The ‘not in’ operator
* Returns False if a sequence with the specified value is present in the object.

The use of this operator is to check whether the data is available in the variable or not( vfariable should be iterable) , if yes then it will return True else False.

For Example:

>>> x=256

>>> 5 in x

Traceback (most recent call last):

File "<pyshell#20>", line 1, in <module>

5 in x

TypeError: argument of type 'int' is not iterable

>>> x=[10,20,30,40] #list operation

>>> 10 in x

True

>>> x=1,2,3,4 #tupple

>>> 1 in x

True

>>> 5 not in x

True

>>>

One thing that is noticeable is it doesnot work for a single number, we cannot ask weather 2 is present in 256 or not as 256 is an entire single number, however it works for strings.

**Test(question in which I was confused)**

1. What will be the value of x if x=3 and x\*=3+4
2. What will be the value of x after executing the following python statement x=2==’’three’’?
3. Confusion in ‘and’ and ‘or’ operator.

**Taking input from the user**

We write any program for some kind of data processing, for those data processing we need data. Either we presume the data or we take data from the users or we can say that input has been provided by us.

Topics:

* Input method.
* Filtering input data.
* Example Program

**Input() method:**

* The input() method takes a single optional argument.

It means we can keep the parenthesis blank or we can pass one value, this value is a string which will be displayed on the screen and the user can see it.

This method read one line provided by the user and convert it into string. The last enter key pressed by the user while providing the input is removed by the interpreter and returns the remaining as a string.

**Filtering input data:**

* How to input user name?
* How to input an integer?
* How to input a float value?
* How to input a complex number?

One thing to be noted is that input() method always return the value as a string, so when we want to pass any other data type rather than string, we need to do the type conversion so that we can do other operations.

For Example:

x=input('Enter your Name')

print(x)

y=input('Enter a number')

print(y)

z=int(y)

print(type(x),type(y),type(z))

output:

Enter your Name gourav prasad

gourav prasad

Enter a number 47

47

<class 'str'> <class 'str'> <class 'int'>

>>>

As we can see from the above program that by default we input() method returns string, if any other data type is needed then we need to do type conversion.

Similarly we can do the same for float and complex also.

For Example:

x=input('Enter your Name')

print(x)

y=input('Enter a number')

print(y)

z=int(y)

w=input('Enter a floating number')

print(w)

v=float(w)

u=input('Enter a complex number')

print(u)

t=complex(u)

print(type(x),type(y),type(z),type(v),type(t))

Output:

Enter your Name Gourav Prasad

Gourav Prasad

Enter a number 25

25

Enter a floating number25.5

25.5

Enter a complex number3+4j

3+4j

<class 'str'> <class 'str'> <class 'int'> <class 'float'> <class 'complex'>

>>>

**Example Program**

Write a python script to calculate Simple Interest and also show the addition of two numbers:

p=input("Enter principle amount: ")

r=input("Enter rate of interest: ")

t=input("Enter the time: ")

si=float(p)\*float(r)\*float(t)/100

print("simple interest is:", si)

x=input('Enter 1st number: ')

y=input('Enter 2nd number: ')

z=int(x)+int(y)

print('sum of two number is: ',z)

Output:

Enter principle amount: 2000

Enter rate of interest: 3.5

Enter the time: 2

simple interest is: 140.0

Enter 1st number: 250

Enter 2nd number: 378

sum of two number is: 628

>>>

**Use of import, as and from:**

Topics:

* Module.
* Import module.
* Module Alias.
* Keyword From.

**Module:**

* Module is a python file consisting of python code.
* Module can define functions, classes and variables.

**Importing Module:**

* Python code in one module gains access to the code in another module by the process of importing it.
* A local import is when you import a module into local scope. When you do your imports at the top of your Python script file that is importing the module into your global scope, which means that any functions or methods that follow will be able to use it.

Use of import:

>>> import math

>>> math.factorial(5) #proper syntax to use the module of math

120

>>> math.sqrt(25)

5.0

>>> help (math) #to check all the built in functions in math module.

**Module Alias:**

Alias means alternative. As per the previous example, we perform the operation by math.function, now suppose we don’t want to write math always, in that case we can use the keyword ‘as’ like, import math as m, so we can write now m.factorial

>>> import math as m

>>> m.factorial(5)

120

>>> math.factorial(5)

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

math.factorial (5)

NameError: name 'math' is not defined

>>>

Once the alias name is chosen we cannot use default one again.

**from keyword** is used when we don’t want to write math.function and also we don’t want to use as keyword then this keyword come into play.

For example:

>>> from math import factorial

>>> factorial(5)

120

>>>

So we can see that without using ‘as’ and module’s name we can construct the code. Now suppose if we want to call two module or more then we just need to add a comma, like as follow:

>>> from math import factorial,sqrt

>>> factorial(5)

120

>>> sqrt(25)

5.0

>>>

Now if we are not sure which module needs to be accessed then we can call all the module of math by a special syntax as follow:

>>> from math import \*

>>> gcd(4,6)

2

>>> factorial(5)

120

>>>

**Assignment:**

**Topics: input and output |operators | Data type**

1. **Write a python script to add two numbers (integers) taken from user through keyboard.**

'''addition of two numbers'''

x=input('enter first number')

y=input('enter second number')

z=int(x)+int(y)

print('The sum is: ',z)

output:

enter first number25

enter second number35

The sum is: 60

>>>

1. **Write a python script to calculate the area of a circle, radius is taken from the user.**

'''python script to calculate the area of a circle, radius is taken from the user'''

r=float(input('Enter the radius of the circle:'))

area=(r\*\*2)\*3.142

print('area of the circle is: ',area)

'''another way to write the program using the module math'''

from math import pi

r=float(input('Enter the radius of the circle:'))

area=pi\*r\*\*2

print('area of a circle is: ', area)

'''another way to find the area of the circle'''

from math import \*

r=float(input('Enter the radius of the circle:'))

print('The area of the circle with radius' +str(r) +' is: '+str(pi\*r\*\*2))

Output:

Enter the radius of the circle:2.0

area of the circle is: 12.568

Enter the radius of the circle:1.1

area of a circle is: 3.8013271108436504

Enter the radius of the circle:2.1

The area of the circle with radius2.1 is: 13.854423602330987

>>>

1. **Write a python script to calculate the volume of a cuboid, dimensions (float values) taken from user.**

'''program to calculate the volume of a cuboid'''

l=float(input('Enter the length: '))

b=float (input('Enter the breath: '))

h=float (input('Enter the height: '))

area=l\*b\*h

print('Area of the cuboid is: ',area)

Output:

Enter the length: 2.5

Enter the breath: 3.5

Enter the height: 1.1

Area of the cuboid is: 9.625

>>>

1. **Write a python script to calculate the simple interest, data required should be taken from the user.**

'''program to calculate the simple interest'''

p=float(input('Enter the principal amount: '))

r=float(input('Enter the rate: '))

t=float(input('Enter the time: '))

si=(p\*r\*t)/100

print('Simple interest is: ',si)

Output:

Enter the principal amount: 2000

Enter the rate: 3.5

Enter the time: 1

Simple interest is: 70.0

>>>

1. **Write a python script to find the square of a given number (number is taken from user)**

'''program to calculate the square of a number'''

#normal method

n=float(input('Enter the number:'))

s=(n\*n)

print("The square of the number is: ",s)

#using exponent

n=float(input('Enter the number: '))

s=n\*\*2

print('Square of the number is: ',s)

#using pow of math module

from math import \*

n=float(input('Enter the number: '))

s=pow(n,2)

print('Square of the number is: ',s)

Output:

Enter the number:5

The square of the number is: 25.0

Enter the number: 6

Square of the number is: 36.0

Enter the number: 7

Square of the number is: 49.0

>>>

1. **Write a python script to find the area of a triangle. Lengths of the sides are given by the user.**

'''program to calculate the area of a triangle'''

b=float(input('Enter the base: '))

h=float(input('Enter the height: '))

area=0.5\*b\*h

print('Area of the triangle is: ',area)

Output:

Enter the base: 5

Enter the height: 4

Area of the triangle is: 10.0

>>>

**More on output statements:**

**Topics:**

1. **Print variable value**
2. **Print multiple value**
3. **Print multiple values with sep**
4. **Print is like println of java**
5. **End**
6. **Sep and together**
7. **Formatted string**
8. **Replacement operator**

Suppose we want to print a single value then the way to write is simple like as follow:

>>> x=10

>>> print(x)

10

>>>

Now suppose we want to print multiple values then we pass multiple arguments separating by a comma like as follow:

>>> x=10

>>> y=20

>>> print(x,y)

10 20

>>>

As we can see in output that there is a space in between 10 and 20. In python the space is taken by default. If we want a comma in between then we can try a way like as follow:

>>> x=10

>>> y=20

>>> print(x,y)

10 20

>>> print(x,',',y)

10 , 20

>>> here also we can notice that by default there is space before and after comma.

If we don’t want that space also then we need to use a **separator** which is denoted by **‘sep’** in python. The use of **‘sep’** is shown below

>>> x=10

>>> y=20

>>> print(x,y)

10 20

>>> print(x,',',y)

10 , 20

>>> print(x,y,sep=',')

10,20

>>>

We can put colon also for separator or anything with which we want to separate.

>>> print(x,y,sep=',')

10,20

>>> print(x,y,sep=':')

10:20

>>> print(x,y,sep='+')

10+20

>>>

In case of strings also it happens, we can concatenate strings like as follow:

>>> print('AB'+'CD')

ABCD

>>>

Suppose we want to print values in two different lines then the following is the process:

>>> print(x,y,sep='\n')

10

20

>>>

If we print multiple statements or value in python with different print commands then python by default print it in next line which is same as println in java.

For Example:

print('Hello')

print('students')

print('learn python')

Output:

Hello

students

learn python

>>>

Suppose we don’t want the default pattern of next line and we simply want it print in a single line, then the way of doing so is as follows:

print('Hello',end=' ')

print('students',end=' ')

print('learn python')

Output:

Hello students learn python

>>>

Use of end and sep together

x=10

y=20

z=30

print(x,y,z,end='.',sep=':')

Output:

10:20:30.

>>>

**Formatted String: (A string in which format specifiers are used)**

In C programming we use to print some results in the following way:

Printf(“Sum of %d and %d is %d”,a,b,c)

We can do the same thing in python, only the syntax differs a bit like as follows:

Print(‘’Formatted string” %(variable)) or Print(‘’Formatted string” %variable)

Print(‘’Formatted string” %(variable,variable)) #for multiple values

**P.S: No comma is used after formatted string as it was there in case of c.**

For Example:

>>> x=10

>>> print('value of x is %d'%x)

value of x is 10

>>> a=3

>>> b=4

>>> c=a+b

>>> print('sum of %d and %d is %d'%(a,b,c))

sum of 3 and 4 is 7

>>>

**Format specifiers in python:**

>>> x=10

>>> print('x=%a'%x)

x=10

>>>

>>> print('x=%d'%x)

x=10

>>> print('x=%f'%x)

x=10.000000

>>> print('x=%F'%x)

x=10.000000

>>> print('x=%g'%x)

x=10

>>> print('x=%G'%x)

x=10

>>> print('x=%s'%x)

x=10

>>> print('x=%o'%x)

x=12

>>> print('x=%x'%x)

x=a

>>> print('x=%X'%x)

x=A

>>> print('x=%O'%x)

Traceback (most recent call last):

File "<pyshell#18>", line 1, in <module>

print('x=%O'%x)

ValueError: unsupported format character 'O' (0x4f) at index 3

>>> print('x=%i'%x)

x=10

>>> print('x=%c'%x)

x=

>>> print('x=%r'%x)

x=10

>>> print('x=%u'%x)

x=10

>>> print('x=%e'%x)

x=1.000000e+01

>>> print('x=%E'%x)

x=1.000000E+01

>>>

**%d and %i is used to print integer value.**

**%f and %F is used to print float values.**

**%e and %E is used to print Exponential values.**

**%g and %G prints integer value if there is no value after point other than 0, however if there is any non-zero then it prints the float value.**

For Example (%g and %G)

>>> x=10.0

>>> print('x=%g'%x)

x=10

>>> x=10.23

>>> print('x=%g'%x)

x=10.23

>>> print('x=%G'%x)

x=10.23

>>> x=10.0

>>> print('x=%G'%x)

x=10

>>>

So basically the difference between %f and %g in python is that %f by default will return float value only even if the value is integer, however %g can return integer value if there is 0 after (.)

**%o is to print octal value of the number, however if we write %O then it will throw error**

**%x and %X returns the hexa-decimal number of the given value.**

**%r also returns the integer value**

**%u is also returns the integer value but for non-negative values**

**Replacement operator {}**

* Pair of curly braces is knows as Replacement Operator
* Let us understand it via a python program.

x=10

y=3.5

z='Gaurav'

print('Hello, ',z,'x=',x,'y=',y)

'''with the above code we can see that there is a iregular space between hello

and gaurav which is taken by default by the , let us see the work of

replacement operator'''

x=10

y=3.5

z='Gaurav'

print('Hello,{2} x={0} y={1}'.format(x,y,z))

'''The above code uses the indexing format as 0,1 and 2 which is needed to be

specified in curly brackets to print the output in correct format, now what if

we dont mention anything in curly brackets'''

x=10

y=3.5

z='Gaurav'

print('Hello, {} x={} y={}'.format(x,y,z))

'''The result will be printed in the format which is specified, that is in first

curly bracket x's value will come and in the second curly bracket y's value will come and

then at last z's value will come'''

''' we can change the format also in our way like as follow'''

x=10

y=3.5

z='Gaurav'

print('Hello, {} x={} y={}'.format(z,x,y))

'''now suppose we mention the variable in the curly braces itself, then in that

case we need to define the values in format itself, if we define there then

what ever we have initialised earlier will not be taken into consideration by

interpreter'''

x=10

y=3.5

z='Gaurav'

print('Hello, {z} x={x} y={y}'.format(z='GauraV',x=1,y=2))

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/12 replacement operator.py

Hello, Gaurav x= 10 y= 3.5

>>>

RESTART: C:/Users/654268/Desktop/Python Programming/12 replacement operator.py

Hello, Gaurav x= 10 y= 3.5

Hello,Gaurav x=10 y=3.5

>>>

RESTART: C:/Users/654268/Desktop/Python Programming/12 replacement operator.py

Hello, Gaurav x= 10 y= 3.5

Hello,Gaurav x=10 y=3.5

Hello, 10 x=3.5 y=Gaurav

>>>

RESTART: C:/Users/654268/Desktop/Python Programming/12 replacement operator.py

Hello, Gaurav x= 10 y= 3.5

Hello,Gaurav x=10 y=3.5

Hello, 10 x=3.5 y=Gaurav

Hello, Gaurav x=10 y=3.5

>>>

RESTART: C:/Users/654268/Desktop/Python Programming/12 replacement operator.py

Hello, Gaurav x= 10 y= 3.5

Hello,Gaurav x=10 y=3.5

Hello, 10 x=3.5 y=Gaurav

Hello, Gaurav x=10 y=3.5

Hello, GauraV x=1 y=2

>>>

**Flow control statements and Decision control statements.**

Flow control statements means that whatever lines of code we have written, how the flow will be, it decides how the flow will be, which line be executed first and which one will be executed later, basically the sequence of execution of line of code is decided by the flow control statements. We can decide which lines to be executed first or we can decide which one not to execute. This practice of deciding is called as flow control instructions.

Flow control instructions are of two types:

1. Decision control instructions.
2. Iterative control instructions.

Decision control statement:

1. If
2. If-else
3. If elif else

Iterative/loop statement:

1. While
2. For

P.S: switch and do-while is not available in python.

Transfer statements:

* Break
* Continue
* Pass

**If in C/C++/java**

If(condition)

{

-------------- if block

--------------

}

However in python it is little different

In python in case of if block we use indentation, it means each statement written after giving indentation is considered as if block.

Indentation needs to be the same of if block, otherwise whatever statement is written without the proper indentation will be considered as out of if block.

After writing the keyword if we don’t need to give parenthesis in python, we just need to give a space and write the statement and then give a colon (:).

For Example:

'''program to understand if keyword'''

x=int(input('Enter a number'))

if x>0:

print('%d is possitive'%x)

if x<0:

print('%d is non-possitive/negative number'%x)

Output:

Enter a number0

0 is possitive

>>> 5

5

>>> -5

-5

>>>

**If-else in c/c++/java**

If(condition)

{

-------------

------------

}

Else

{

----------------

---------------

}

If-else in python has same indentation logic as that of if

If condition:

---------------

---------------

else:

-------------

-------------

For Example:

'''program to understand if-else keyword'''

x=int(input('Enter a number'))

if x>0:

print('%d is possitive'%x)

else:

print('%d is non-possitive/negative number'%x)

Output:

== RESTART: C:/Users/654268/Desktop/Python Programming/14 ifelse keyword.py ==

Enter a number18

18 is possitive

>>>

== RESTART: C:/Users/654268/Desktop/Python Programming/14 ifelse keyword.py ==

Enter a number-14

-14 is non-possitive/negative number

>>>

Few points to consider:

1. If we give parenthesis in if’s condition then it won’t show any error, it’s our choice to give parenthesis or not.
2. It is not necessary that if block’s indentation should match with else block’s indentation, we can give different indention to different blocks.

**If elif else keyword:**

it is like a ladder of if else

if condition:

----------------

elif condition:

---------------

else

--------------- optional

For Example:

'''program to understand if-elif-else keyword'''

x=int(input('Enter a number'))

if x>0:

print('%d is possitive'%x)

elif x<0:

print('%d is non-possitive/negative number'%x)

else:

print('%d is zero'%x)

Output:

Enter a number0

0 is zero

**Assignment 2**

1. **Write a python script to check whether the given number is even or odd.**

'''Write a python script to check whether the given number is even or odd'''

#using modulas operator:

x=int(input('Enter an integer value greater than zero:'))

if (x%2==0):

print('%d is an even number'%x)

else:

print('%d is an odd number'%x)

#using bitwise operator:

x=int(input('Enter an integer value greater than zero:'))

if (x&1==0):

print('%d is an odd number'%x)

else:

print('%d is an even number'%x)

#Check Even / Odd without using modulus or bitwise operator:

x=int(input('Enter an integer value greater than zero:'))

y=int(x/2)\*2

if(x==y):

print('%d is even'%x)

else:

print('%d is odd'%x)

Output:

Enter an integer value greater than zero:6

6 is an even number

Enter an integer value greater than zero:8

8 is an odd number

Enter an integer value greater than zero:5

5 is odd

>>>

1. **Write a python script to check whether the number is divisible by 5 or not?**

'''python script to check whether the number is divisible by 5 or not'''

#using modulas operator

x=int(input('Enter a possitive integer number: '))

if (x%5==0):

print('%d is divisible by 5'%x)

else:

print('%d is not divisible by 5'%x)

#without using modulas operator

x=int(input('Enter a possitive integer number: '))

y=int(x/5)\*5

if(x==y):

print('%d is divisible by 5'%x)

else:

print('%d is not divisible by 5'%x)

output:

Enter a possitive integer number: 26

26 is not divisible by 5

Enter a possitive integer number: 26

26 is not divisible by 5

>>>

**Need to see by other possible way also after learning about strings, because we can say like if last digit is 5 or 0 then the number is divisible by 5**

1. **Write a python script to check whether the given number is positive negative or zero**

'''python script to check whether the given number is positive negative or zero'''

x=int(input('Enter a number: '))

if x>0:

print('%d is possitive'%x)

elif x<0:

print('%d is negative'%x)

else:

print('Number is zero')

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q3.py ===

Enter a number: 0

Number is zero

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q3.py ===

Enter a number: 5

5 is possitive

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q3.py ===

Enter a number: -1

-1 is negative

>>>

1. **Write a python script to find greatest among three numbers.**

'''python script to find greatest among three numbers'''

x=int(input('Enter the first number: '))

y=int(input('Enter the Second number: '))

z=int(input('Enter the third number: '))

if x>y and x>z:

print('%d is largest'%x)

elif y>x and y>z:

print('%d is largest'%y)

else:

print('%d is largest'%z)

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q4.py ===

Enter the first number: 20

Enter the Second number: 30

Enter the third number: 40

40 is largest

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q4.py ===

Enter the first number: 15

Enter the Second number: 28

Enter the third number: 20

28 is largest

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q4.py ===

Enter the first number: 99

Enter the Second number: 55

Enter the third number: 63

99 is largest

>>>

1. **Write a python script to check if a year is leap year or not.**

'''python script to check if a year is leap year or not'''

year=int(input('Enter a four digit year: '))

if(year%4==0):

print('%d is leap year'%year)

else:

print('%d is a normal year'%year)

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q5.py ===

Enter a four digit year: 2016

2016 is leap year

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q5.py ===

Enter a four digit year: 2017

2017 is a normal year

>>>

1. **Write a python script to take month value in numeric format and display number of days in it.**

'''python script to take month value in numeric format and display number of days in it'''

year=int(input('Enter a four digit year: '))

month=int(input("Enter month's numeric value: "))

d1=31

d2=30

d3=28

d4=29

if month==1:

print('Month is january and has %d days'%d1)

if month==2 and year%4!=0:

print('Year is non-leap and the monthe is febuary and has %d days'%d3)

if month==3:

print('Month is March and has %d days'%d1)

if month==4:

print('Month is April and has %d days'%d2)

if month==5:

print('Month is May and has %d days'%d1)

if month==6:

print('Month is june and has %d days'%d2)

if month==7:

print('Month is july and has %d days'%d1)

if month==8:

print('Month is August and has %d days'%d1)

if month==9:

print('Month is september and has %d days'%d2)

if month==10:

print('Month is october and has %d days'%d1)

if month==11:

print('Month is november and has %d days'%d2)

if month==12:

print('Month is december and has %d days'%d1)

elif month==2 and year%4==0:

print('Year is leap and the month is febuary and has %d days'%d4)

Output:

=== RESTART: C:\Users\654268\Desktop\Python Programming\Assignment 2 q6.py ===

Enter a four digit year: 2016

Enter month's numeric value: 2

Year is leap and the month is febuary and has 29 days

>>>

=== RESTART: C:\Users\654268\Desktop\Python Programming\Assignment 2 q6.py ===

Enter a four digit year: 2017

Enter month's numeric value: 3

Month is March and has 31 days

>>>

=== RESTART: C:\Users\654268\Desktop\Python Programming\Assignment 2 q6.py ===

Enter a four digit year: 2018

Enter month's numeric value: 2

Year is non-leap and the monthe is febuary and has 28 days

>>>

=== RESTART: C:\Users\654268\Desktop\Python Programming\Assignment 2 q6.py ===

Enter a four digit year: 1993

Enter month's numeric value: 12

Month is december and has 31 days

>>>

1. **Write a python script to check nature of roots of a given quadratic equation.**

'''python script to check nature of roots of a given quadratic equation'''

from math import \*

from fractions import Fraction

print ('quadratic equation:(a\*x^2)+b\*x+c')

a=Fraction(input('a: '))

b=Fraction(input('b: '))

c=Fraction(input('c: '))

d=(b\*\*2-4\*a\*c)

if d>0:

Number\_roots = 2

x1=((-b+sqrt(d))/(2\*a))

x2=((-b-sqrt(d))/(2\*a))

print('Nature of roots: Real and Distinct,Roots: %f and %f'%(x1,x2))

elif d==0:

Number\_roots =1

x=(-b)/(2\*a)

print('Nature of roots: Real and Equal,Roots: %f'%x)

else:

Number\_Roots = 0

print('No Roots')

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q7.py ===

quadratic equation:(a\*x^2)+b\*x+c

a: 1

b: -7

c: 12

Nature of roots: Real and Distinct,Roots: 4.000000 and 3.000000

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q7.py ===

quadratic equation:(a\*x^2)+b\*x+c

a: 1

b: 1

c: 1

No Roots

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q7.py ===

quadratic equation:(a\*x^2)+b\*x+c

a: 3

b: -2

c: 1/3

Nature of roots: Real and Equal,Roots: 0.333333

>>>

1. **Write a python script to print a set of three words in dictionary order. Words are given by user.**

'''Arrange three words in dictionary order'''

print('Enter three city names: ')

a,b,c=input(),input(),input()

if a<b<c:

print(a,b,c)

elif a<c<b:

print(a,c,b)

elif b<a<c:

print(b,a,c)

elif b<c<a:

print(b,c,a)

elif c<b<a:

print(c,b,a)

else:

print(c,a,b)

'''Another simple way without using conditional operator'''

print('Enter three city names:')

a,b,c=input(),input(),input()

x=min(a,b,c)

print(x)

if x==a:

print(min(b,c),max(b,c))

elif x==b:

print(min(a,c),max(a,c))

else:

print(min(a,b),max(a,b))

'''Single line pragram'''

[x for x in sorted(input('Enter comma seperated city names: ').split(',')) if print(x,end=' ')]

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q8.py ===

Enter three city names:

bhopal

jaipur

ludhinaya

bhopal jaipur ludhinaya

Enter three city names:

ahmedabad

aurangabad

ranchi

ahmedabad

aurangabad ranchi

Enter comma seperated city names: asia,africa,europe,northamerica,southamerica,antartica,australia

africa antartica asia australia europe northamerica southamerica

>>>

1. **Write a python script to accept one complex number from the user and display the greater number between real and imaginary part.**

c=complex(input('Enter a complex number'))

if c.real>c.imag:

print('%d is greater'%(c.real))

elif c.real<c.imag:

print('%d is greater'%(c.imag))

else:

print('Both are equal')

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q9.py ===

Enter a complex number5-6j

5 is greater

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q9.py ===

Enter a complex number5+5j

Both are equal

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 2 q9.py ===

Enter a complex number3+4j

4 is greater

>>>

**Ternary Operator in python: Single line if expression: Conditional operator**

Ternary operator are the operators which needs three operands to operate.

In C/C++/java Ternary operator was like \_\_\_\_\_?\_\_\_\_\_:\_\_\_\_\_\_

Expression before ‘?’ is considered as condition, if the condition is true then the expression after ‘?’ is executed else the expression after ‘:’ is executed.

Also conditional operator returns some value which can be assigned in some variable, hence that was considered as expression. An expression is something which returns value.

For example

Z=x>y?x:y

The above example is not possible via if-else statement, hence we can say that if else doesn’t return value and is not considered as expression.

Now the question is how to use it in python. one thing is that if-else can be used as an expression in python, however the syntax is little different.

Syntax in python to use if-else as conditional operator is:

**Value1 if condition else value2**

If condition is true then value one will be selected else value 2 will be selected.

Now if we want we can print the value or else we can assign it some variable like

**Variable**= **Value1 if condition else value2**

For Example:

>>> print('Hello' if 3<4 else 'Bye')

Hello

Example via python script:

x=int(input('Enter first number: '))

y=int(input('Enter second number: '))

z=x if x>y else y

print('Greater number is',z)

Output:

Enter first number: 15

Enter second number: 18

Greater number is 18

>>>

**Iterative control statement: loop statement**

* while loop
* for loop
* Decision/selection control statement
* If
* If else
* If elif else

These decides the flow of the statement or program or we can say the sequence

A situation where we want a particular statement to be executed repeatedly then we need to apply loop, that can be achieved by using while loop and for loop

In python there is no do-while loop.

We need to understand properly that when to use ‘for’ loop and when to use ‘while’ loop.

**While loop:**

It is used when we need to apply to some condition to get the output, hence we can say while loop is used in conditional based scenarios.

For Example if I want to print ‘Gourav’ five times, in that case we can write the program as:

'''understanding while loop'''

x=1 #initialization

while x<=5: #condition to get the output repeatedly

print('Gourav')

x=x+1 #flow(deciding statement for flow of the output

#these three are important factor in while loop i.e; initialization, condition and flow

'''another example for while loop: print first 10 natural numbers'''

x=1 #initialization

while x<=10: #condition

print(x)

x=x+1 #flow

Output:

Gourav

Gourav

Gourav

Gourav

Gourav

1

2

3

4

5

6

7

8

9

10

>>>

Now suppose we want to print the natural numbers in reverse from 10 to 1

Then in that case initialization, condition and decision will be changed as follows:

'''printing natural numbers in reverse'''

x=10

while x>=1:

print(x)

x=x-1

Output:

10

9

8

7

6

5

4

3

2

1

**Assignment 3:**

* **Write a python script to print to print first 10 odd natural numbers**

'''print first 10 odd natural numbers'''

x=1

while x<=9:

print(x)

x=x+2

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 3 q3.py ===

1

3

5

7

9

>>>

* **Write a python script to print first 10 even natural numbers**

'''print first 10 even natural numbers'''

x=2

while x<=10:

print(x)

x=x+2

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 3 q4.py ===

2

4

6

8

10

>>>

* **Write a python script to print first N natural number. Value of N is taken from user.**

'''python script to print first N even natural numbers'''

x=1

n=int(input('Enter The number: '))

while x<=n:

print(x,end=' ')

x=x+1

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 3 q5.py ===

Enter The number: 10

1 2 3 4 5 6 7 8 9 10

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 3 q5.py ===

Enter The number: 100

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100

>>>

* **Write a python script to print first n natural number in reverse order. Value of N is taken from user**

'''print first n natural number in reverse order'''

n=int(input('Enter an odd number: '))

while n: #Every non zero value is true and 0 is false

print(n,end=' ')

n=n-1

Output:

Enter an odd number: 20

20 19 18 17 16 15 14 13 12 11 10 9 8 7 6 5 4 3 2 1

>>>

**For Loop:**

c/c++/java Python

for(i=1;i<=10;i++) for element(variable name) in sequence:

{ perform-action

----------------

--------------

}

1. for loop in python doesn’t need a variable to be created to maintain the index this happens because for loop in python is only applied in sequence.
2. for loop is used when we want to access each and every element of a sequence one by one.
3. Sequence examples: string, list, tupple, range, dict, set. All this contains a set of values, For example “Gourav” is a string which has 6 elements. So now suppose we want to execute statements repeatedly equal to the number of elements in sequence then in that case we use for loop.
4. When we use for loop elements take the first sequence and then execute the statements.

**Write a program in which we will input a name and we need to decide how many small a is there.**

x=input("Enter the name: ")

count=0

for e in x:

if e=='a':

count+=1

print("count= ",count)

Output:

Enter the name: gourav

count= 1

**Range() function**

This function generates sequence.

If we pass an integer value in range function then it will generate a sequence.

For example:

>>> print(range(5))

range(0, 5)

>>>

Here it is showing the range simply, but we want to print the value which is generated in sequence, to do so we need to use for loop as such:

for x in range(5):

print(x)

Output:

====== RESTART: C:/Users/654268/Desktop/Python Programming/19 range.py ======

0

1

2

3

4

>>>

Default indexing of Range() function starts from0, however if we want we can print the values leaving 0, like as follow:

for x in range(5):

print(x+1)

Output:

1

2

3

4

5

>>>

Range by default starts from zero. Hence it will generate the sequence from 0 to n-1; n being the input or the number passed in range function.

Range will always generate a sequence of numbers and that also integer value. It cannot be floating value or any character, if we pass a float value in range function, it will throw an error, like as follows:

>>> range(5.6)

Traceback (most recent call last):

File "<pyshell#1>", line 1, in <module>

range(5.6)

TypeError: 'float' object cannot be interpreted as an integer

>>> range('a')

Traceback (most recent call last):

File "<pyshell#2>", line 1, in <module>

range('a')

TypeError: 'str' object cannot be interpreted as an integer

>>>

Hence arguments to be passed in range function should be integer value only and the result will also be the sequence of integers.

**Proper syntax of range() is that we can pass three values:**

range(start, stop, step)

start value by default is always zero(0)

stop value is passed as an argument by the user in range() function, stop value is always excluded when the result is printed.

step by default is 1.

So for example:

range(5)

Means that start is 0 and stop is 5 which will be excluded and the step which is 1 represents that how much the value will be incremented.

Let’s pass some different values in range function.

>>> for x in range(5,10):

print(x)

5

6

7

8

9

>>>

What if we put start value greater than the stop value:

>>> for x in range(10,5):

print(x)

>>>

**No error will come but no output will be displayed, hence we shall always pass lower start value in comparison to stop value, this is possible if we put a negative step value. For example:**

>>> for x in range(10,5,-1):

print(x)

10

9

8

7

6

>>>

Similarly we can get the numbers in reverse sequence by adjusting the step value in our way.

For Example:

>>> for x in range(10,0,-1):

print(x)

10

9

8

7

6

5

4

3

2

1

>>>

**Assignment 4:**

1. **Write a python script to check whether a given number is prime or not.**

'''check whether a given number is prime or not.'''

n=int(input("Enter a number: "))

if n<2:

print("Not a prime number")

else:

for i in range(2,n):

if n%i==0:

print("Not a prime number")

break

else:

print("Number is prime")

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q1.py ===

Enter a number: 6

Not a prime number

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q1.py ===

Enter a number: 113

Number is prime

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q1.py ===

Enter a number: 27

Not a prime number

>>>

1. **Write a python script to find the next prime number of a given number.**

'''python script to find the next prime number of a given number'''

n=int(input("Enter a number: "))

x=n+1

while True:

for i in range(2,x):

if x%i==0:

break

else:

print("Next prime number is: ",x)

break

x+=1

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q2.py ===

Enter a number: 14

Next prime number is: 17

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q2.py ===

Enter a number: 19

Next prime number is: 23

>>>

1. **Write a python script to print first n prime numbers.**

'''print first n prime numbers'''

n=int(input("Enter a natural number: "))

x=2

while n:

for i in range(2,x):

if x%i==0:

break

else:

print(x,end=" ")

n=n-1

x=x+1

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q3.py ===

Enter a natural number: 50

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q3.py ===

Enter a natural number: 10

2 3 5 7 11 13 17 19 23 29

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q3.py ===

Enter a natural number: 100

2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523 541

1. **Write a python script to print all prime numbers between two numbers.**

'''print all prime numbers between two numbers'''

l=int(input("Enter first number: "))

u=int(input("Enter second number: "))

if l>u:

l,u=u,l

if l<1 and u>=2:

l=2

if l<u<2:

pass

else:

for x in range(l+1,u):

for i in range(2,x):

if x%i==0:

break

else:

print(x,end=' ')

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q4.py ===

Enter first number: 10

Enter second number: 20

11 13 17 19

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q4.py ===

Enter first number: 20

Enter second number: 5

7 11 13 17 19

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q4.py ===

Enter first number: -10

Enter second number: -5

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q4.py ===

Enter first number: -5

Enter second number: -10

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q4.py ===

Enter first number: 2

Enter second number: 25

3 5 7 11 13 17 19 23

>>>

1. **Write a python script to check for co-prime numbers.**

'''check for co-prime numbers'''

print("Enter tow numbers: ")

l,u=int(input()),int(input())

h=min(l,u)

while h>=1:

if l%h==0 and u%h==0:

if h==1:

print("Co-Prime Numbers")

else:

print("Not a co-prime numbers")

break

h-=1

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q5.py ===

Enter tow numbers:

3

7

Co-Prime Numbers

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q5.py ===

Enter tow numbers:

3

6

Not a co-prime numbers

>>>

**Write a python script to print N odd natural numbers in reverse, n is input provided by the user:**

'''print first N odd natural numbers in reverse order'''

n=int(input("Enter a number: "))

if n%2==0:

for x in range(n-1,0,-2):

print(x,end=' ')

else:

for x in range(n-2,0,-2):

print(x,end=' ')

Output:

=== RESTART: C:\Users\654268\Desktop\Python Programming\Assignment 4 q6.py ===

Enter a number: 26

25 23 21 19 17 15 13 11 9 7 5 3 1

>>>

=== RESTART: C:\Users\654268\Desktop\Python Programming\Assignment 4 q6.py ===

Enter a number: 23

21 19 17 15 13 11 9 7 5 3 1

>>>

This program is not in assignment. Wrote it only for learning purpose.

1. **Write a python script to print N odd natural numbers in reverse order.**

'''print N odd natural numbers in reverse order'''

n=int(input("Enter a number: "))

for i in range(2\*n-1,0,-2):

print(i,end=' ')

'''Another way of coding'''

n=int(input("Enter a number: "))

for i in range(1,n+1):

print(2\*n-2\*i+1,end=' ')

Output:

Enter a number: 10

19 17 15 13 11 9 7 5 3 1

Enter a number: 5

9 7 5 3 1

>>>

1. **Write a python script to print the first N natural even numbers in reverse order.**

'''print N even natural numbers in reverse order'''

n=int(input("Enter a number: "))

for i in range(2\*n,0,-2):

print(i,end=' ')

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q7.py ===

Enter a number: 5

10 8 6 4 2

>>>

1. **Write a python script to print prime factors of a given number.**

'''python script to print prime factors of a given number'''

n=int(input("Enter a number: "))

x=n

p=2

while x>1:

while x%p==0:

print(p,end=' ')

x//=p

p=p+1

while True:

for i in range(2,p):

if p%i==0:

p=p+1

break

else:

break

Output:Enter a number: 36

2 2 3 3

>>>

=== RESTART: C:\Users\654268\Desktop\Python Programming\Assignment 4 q8.py ===

Enter a number: 98

2 7 7

>>>

=== RESTART: C:\Users\654268\Desktop\Python Programming\Assignment 4 q8.py ===

Enter a number: 21

3 7

1. **Write a python script to find the LCM of two numbers**

'''python script to find the LCM of two numbers'''

print("Enter two numbers: ")

a,b=int(input()), int(input())

L=max(a,b)

while L<=a\*b:

if L%a==0 and L%b==0:

print("LCM is",L)

break

L+=1

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q9.py ===

Enter two numbers:

4

6

LCM is 12

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q9.py ===

Enter two numbers:

100

59

LCM is 5900

1. **Write a python script to find the HCF of two numbers**

'''python script to find the HCF/GCD of two numbers'''

print("Enter two numbers: ")

a,b=int(input()), int(input())

H=min(a,b)

while H>=1:

if a%H==0 and b%H==0:

print("HCF is ",H)

break

H-=1

Output:

== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q10.py ==

Enter two numbers:

4

6

HCF is 2

>>>

== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 4 q10.py ==

Enter two numbers:

3

7

HCF is 1

>>>

**Assignment 5:**

1. **Write a python script to print a table of 5.**

'''python script to print a table of 5'''

n=int(input("Enter the number: "))

x=5

for i in range(x,(n+1)\*5,5):

print(i,end=' ')

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q1.py ===

Enter the number: 10

5 10 15 20 25 30 35 40 45 50

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q1.py ===

Enter the number: 20

5 10 15 20 25 30 35 40 45 50 55 60 65 70 75 80 85 90 95 100

>>>

1. **Write a python script to print a table of user’s choice.**

'''python script to print a table of user’s choice'''

n=int(input('Enter a number for which the mutiple table is needed: '))

i=int(input('Enter a number till which the number table is neede: '))

for x in range(n,(n\*(i+1)),n):

print(x,end=' ')

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q2.py ===

Enter a number for which the mutiple table is needed: 6

Enter a number till which the number table is neede: 15

6 12 18 24 30 36 42 48 54 60 66 72 78 84 90

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q2.py ===

Enter a number for which the mutiple table is needed: 19

Enter a number till which the number table is neede: 10

19 38 57 76 95 114 133 152 171 190

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q2.py ===

Enter a number for which the mutiple table is needed: 23

Enter a number till which the number table is neede: 25

23 46 69 92 115 138 161 184 207 230 253 276 299 322 345 368 391 414 437 460 483 506 529 552 575

>>>

1. **Write a python script to print all Armstrong numbers under 1000**

'''python script to print all Armstrong numbers under 1000'''

for x in range(1,1001):

n=x

s=0

while x:

r=x%10

s=s+r\*\*3

x=x//10

if s==n:

print(n)

'''python script to check whether the inout provided by the user is Armstrong

or not needs clear concept of Functions, hence we will do the same program after

gaining knowledge about functions'''

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q3.py ===

1

153

370

371

407

>>>

1. **Write a python script to print square of numbers from a to b**

'''python script to print square of numbers from a to b'''

a=int(input("Enter the first number: "))

b=int(input("Enter second number: "))

if a>b:

a,b=b,a

for i in range(a,b+1):

print(i\*\*2,end=' ')Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q4.py ===

Enter the first number: 1

Enter second number: 20

1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q4.py ===

Enter the first number: 1

Enter second number: 100

1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400 441 484 529 576 625 676 729 784 841 900 961 1024 1089 1156 1225 1296 1369 1444 1521 1600 1681 1764 1849 1936 2025 2116 2209 2304 2401 2500 2601 2704 2809 2916 3025 3136 3249 3364 3481 3600 3721 3844 3969 4096 4225 4356 4489 4624 4761 4900 5041 5184 5329 5476 5625 5776 5929 6084 6241 6400 6561 6724 6889 7056 7225 7396 7569 7744 7921 8100 8281 8464 8649 8836 9025 9216 9409 9604 9801 10000

>>>

1. **Write a python script to print a sequence of number with given step and boundary value.**

'''python script to print sequence of numbers with given step and boundary value'''

a=int(input("Enter Starting value"))

b=int(input("Enter end value"))

s=int(input("Enter step size"))

if a>b and s<0 or a<b and s>0:

for i in range(a,b+1,s):

print(i,end=' ')

else:

print("Wrong Number")

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q5.py ===

Enter Starting value10

Enter end value20

Enter step size2

10 12 14 16 18 20

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q5.py ===

Enter Starting value12

Enter end value16

Enter step size3

12 15

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q5.py ===

Enter Starting value10

Enter end value5

Enter step size-2

10 8

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 5 q5.py ===

Enter Starting value10

Enter end value5

Enter step size2

Wrong Number

>>>

**Transfer statements in python:  
  
Break, continue and pass**

Before jumping into Transfer statements let us understand the **infinite loop**, sometimes by mistake we will make infinite loop.

'''program to understand infinite loop'''

x=5

while x>0:

print("Hello")

x=x-1

'''The above program doesnot have infinite loop and will run smooth'''

'''as we can see that x=x-1 is inside the while loop'''

'''so whenever the condition is checked 'x' operation is also getting executed,

however if we keep the "x=x-1" outside the while loop then the operation will

never be executed and the code will stuck in infinite loop as in that case

while x>0 is always true because x is initialised at value 5 and it is less

than 0, hence the value will always be true for while condition and it will be

infinite loop'''

'''lets see that byu doing'''

x=5

while x>0:

print("Hello")

x=x-1

We can stop the infinite loop by pressing **Ctrl+C**

**Break:**

With the break statement we can stop the loop even if the while condition is true.

Suppose we are writing a program which is running the loop 10 times, however our goal is achieved in the 5th run itself and don’t want the loop to be running, in that scenario we use break such that even if the while or loop condition is true the control will go outside the loop.

Let us understand that by a simple program:

'''Game program to enter even number in three chance'''

#idea is to understand the concept of break

chance=1

while chance<=3:

x=int(input("Enter an Even Number: "))

if x%2==0:

break

chance+=1

if chance==4:

print("You lost")

else:

print("you Win")

'''as per the code's logic, 3 chance will be given to the player to enter the

correct even number, if the 3 chance is exceeded then the player looses'''

'''we can see that the chance will run only three times, however what if the

player gives the right answer in first attempt, in that case we need to apply

break in the middle of the execution of the loop, this is the scenario where

the role of 'break' keyword comes''' see program number 25.

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/21 understanding break keyword.py

Enter an Even Number: 15

Enter an Even Number: 11

Enter an Even Number: 13

You Lost

>>>

RESTART: C:/Users/654268/Desktop/Python Programming/21 understanding break keyword.py

Enter an Even Number: 2

you Win

>>>

RESTART: C:/Users/654268/Desktop/Python Programming/21 understanding break keyword.py

Enter an Even Number: 11

Enter an Even Number: 113

Enter an Even Number: 115

You lost

>>>

**Continue:**

With the continue statement we can stop the current iteration, and continue with the next.

'''Program to understand the continue keyword'''

x=1

while x<=24:

if x%5==0:

x=x+1 #if we dont put this then it will print only till 4

continue

print(x)

x=x+1

#what if we use break insted of continue

x=1

while x<=24:

if x%5==0:

break

print(x)

x=x+1

'''if we use continue then the control goes back to the while condition loop,

whereas in break it goes out of the loop'''

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/22 understanding continue keyword.py

1 2 3 4 6 7 8 9 11 12 13 14 16 17 18 19 21 22 23 24 1 2 3 4

>>>

**Pass:**

'''Program to understand the pass keyword'''

'''if we are running a statement and we want that if the condition of 'if' is

Executed then no operations is needed to be performed then there comes the role of

pass keyword'''

'''Now the issue is if we code this in C/C++/Java then we can simply put the curly

brackets and print the next statement, however this is not possible in python

as python understands indentation, lets see how we can do that'''

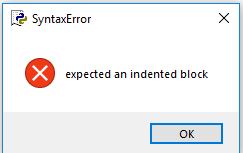
x=0

if x==0:

print("Hello") '''print statement doesnot depend on if condition, but there is

no indentation for if block, hence it will throw error as

expected and indented block as shown below'''



'''so for that reason we will use the keyword pass, lets see how it goes through

after using the pass keyword'''

#hence we will use pass keyword when we need to make empty block.

#it will be treated as there is nothing in the block and the program will be passed

x=0

if x==0:

pass

print("Hello")

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/24 understanding pass keyword.py

Hello

>>>

**Use of Else with loop:**

* while-else
* for-else

In C/C++/Java we use to use else only with if keyword, however in python we can use else with both while loop and for loop.

Hence in python we can use else with the following:

* if else
* if elif else
* Single line if else which acts as ternary operator or conditional operator.
* While else
* For else

**While-else:**

* Else condition will be used when the while loop’s condition will be false.
* Now the question comes that when we were using while loop and if there was a break keyword then anyway the statements after break (out of while loop) were getting executed, so what is the use of else.
* Else is meaningful when break keyword is used in while loop
* Important point to note is that, when a break keyword is encountered then the control goes out of the loop, in-fact out of else loop also, so if the loop’s condition is false then the control comes to else block.
* If the loop is ended/terminated due to break then the control doesn’t goes to else’s block.
* While loop can be terminated under two circumstances:

1. If the condition is false.
2. Break is executed.

* Hence if the condition is false then else block will be executed and if the break is executed then else block will not be executed.

Let us go through a program to understand the above statements:

'''Game program to enter even number in three chance'''

#idea is to understand the concept of while-else

i=1

while i<=3:

x=int(input("Enter an Even Number: "))

if x%2==0:

print("You Win")

break

i=i+1

else:

print("you lost")

Output:

==== RESTART: C:/Users/654268/Desktop/Python Programming/25 while else.py ====

Enter an Even Number: 11

Enter an Even Number: 13

Enter an Even Number: 15

you lost

>>>

==== RESTART: C:/Users/654268/Desktop/Python Programming/25 while else.py ====

Enter an Even Number: 2

You Win

>>>

**For-else:**

* The concept is same as while-else
* Else is meaningful when break keyword is used in for loop
* If the sequence ends then the else’s loop is executed.
* When break is executed else will not work
* Else work when condition of while is false.

Let us understand the concept by a simple program. Program is that we will take two inputs from the user and in between that two numbers we will exclude the second number will print the immediate prime number after the first number.

#program to print the prime numbers between a and b

a=int(input("Enter smaller number: "))

b=int(input("Enter greater number: "))

s=range(a,b)

for x in s:

for num in range(2,x):

if x%num==0:

break

else:

print(x,"is prime in range")

break

'''if the last break is not used then the program will print all the prime-

numbers in the range of a and b, however if it is used then it will print

the first prime number in range of a and b

Output:

===== RESTART: C:/Users/654268/Desktop/Python Programming/26 for else.py =====

Enter smaller number: 10

Enter greater number: 20

11 is prime in range

>>>

===== RESTART: C:/Users/654268/Desktop/Python Programming/26 for else.py =====

Enter smaller number: 3

Enter greater number: 11

3 is prime in range

>>>

--------------------------------------------------End of Basics------------------------------------------------------------

**Sequences:**

**List in python:**

**Part1:**

As in C/C++/Java we have the concept of arrays, similarly we have sequences in python, however it is the improvised version of array.

* In C/C++/Java as we have seen that we cannot expand the array, however there was a concept of dynamic array if the coding is done explicitly, in case of python list is expandable and we can add values while coding itself.
* In list we can keep heterogeneous data, we can keep different types of values, it can contain as many variables we want.
* Lists can also be iterated over in a very simple manner.

Let us understand an overview (not deep dive) of list by a simple program.

'''program to understand the concept of list part 1: '''

l=[]

l.append(10)

l.append(20)

l.append(30)

'''the above statement l.append is the basic way to store the value in empty

list by using the function append'''

print(l)

#above way is one method to print the list, there are other ways also

print(l[0],l[1],l[2])

#above way is via indexing method.

'''The difference between normal print method and indexing print method is that

Normal print method will show the output in square bracket with a comma and space,

However in indexing method it is printed normally with a space in between.'''

for x in l:

print(x)

#above method is the iterable method.

'''in this case it will print each value in different lines'''

'''This program gives the basic idea how we can create a list and how to store

values in it and print in different ways.'''

l=[10,20,30]

print(l[0],l[1],l[2])

for x in l:

print(x)

Output:

[10, 20, 30]

10 20 30

10

20

30

10 20 30

10

20

30

>>>

**Part 2:**

We are going to deep dive on list topic.

1. Create list:
2. l=[] Empty list
3. l1=[10,20,30]
4. l2=[“Bhopal”, “Delhi”, “Mumbai”]
5. l3=[10,25.5,”Bhopal”] Heterogeneous data.

In C/C++ if we cross the upper index in the array then there will illegal memory allocation which there is no mechanism to check the upper bound limits, however in java it will throw exception which will happen in case python also.

The above statement will be cleared in the below shell code:

>>> l1=[10,20,30]

>>> print(l1[3])

Traceback (most recent call last):

File "<pyshell#1>", line 1, in <module>

print(l1[3])

IndexError: list index out of range

>>>

Now suppose we have created an empty list and we are trying to store 10 in index 0 , then it will throw an error. Lets see the below shell code:

>>> l2=[] #empty lsit

>>> l2[0]=10

Traceback (most recent call last):

File "<pyshell#4>", line 1, in <module>

l2[0]=10

IndexError: list assignment index out of range

>>> #it is showing error because the list is empty and there is no index

So we can conclude that we cannot access the index when there is no value in list, however we can use the append function to keep the value in list.

We can keep the value in list like the below way:

>>> l2=[]

>>> l2=[10,20]

>>> l2 # here it will print the values of the list.

[10, 20]

>>>

Now suppose we write one more line and assign the value as 1,2 and 3, in this case it will modify the old list and it will show the new list value as shown in shell code below:

>>> l2=[]

>>> l2=[10,20]

>>> l2

[10, 20]

>>> l2=[1, 2, 3]

>>> l2

[1, 2, 3]

>>>

Hence the above method can be used to remove the older data and assign new data. So if we want to append new values to the existing list then there is a different way for that:

>>> a=[]

>>> a.append(10)

>>> a.append(20)

>>> a

[10, 20]

>>> a.insert(2,30)

>>> a

[10, 20, 30]

>>> ''' append always adds value from the end, on the other hand insert function is the customized one, as we can pass the value by mentioning the index number as shown above'''

>>> '''Now suppose we want to expand the list and add value 40 at index 3, but

by mistake we have provided the value 40 at index 4, in this case it will not

throw any error, but the value will actually be stored in index 3 only, let

us see that from the following code'''

>>> a.insert(4,40)

>>> a

[10, 20, 30, 40]

>>> a[3]

40

>>> a[4]

Traceback (most recent call last):

File "<pyshell#26>", line 1, in <module>

a[4]

IndexError: list index out of range

>>> '''hence we can see that even if we provide higher index value, it is stored

in index 3 sequentially only'''

'hence we can see that even if we provide higher index value, it is stored in index 3 sequentially only'

>>> '''insert option can insert values in between also, now the question is

what will happen to the value which is replaced, it will actually be shifted let us understand that from the following code'''

'insert option can insert values in between also, now the question is what will happen to the value which is replaced, it will actually be shifted let us understand that from the following code'

>>> a.insert(1,50)

>>> a

[10, 50, 20, 30, 40]

>>> a[4]

40

>>> '''what if we put negative value of index, it will not throw any error, it

Will count the index value from right to left in contrary to positive value

Which gets assigned as per left to right, let us see that in the following code'''

>>> a.insert(-1,60)

>>> a

[10, 50, 20, 30, 60, 40]

>>> a.insert(-3,100)

>>> a

[10, 50, 20, 100, 30, 60, 40]

>>> #insert function works on negative values too

>>> '''we can append two different list by using + operator as shown in the

following code'''

>>> A=[1,2,3]

>>> B=[4,5,6]

>>> C=A+B

>>> C

[1, 2, 3, 4, 5, 6]

>>> '''if we want to add a single value like A+4, it will throw error as scalar

value cannot be added, however we can code it as A+[10]'''

>>> A+4

Traceback (most recent call last):

File "<pyshell#56>", line 1, in <module>

A+4

TypeError: can only concatenate list (not "int") to list

>>> A+[10]

[1, 2, 3, 10]

>>> '''one thing here is that the value 10 has been appended to the list A, it is not assigned to list A'''

>>> A(3)

Traceback (most recent call last):

File "<pyshell#59>", line 1, in <module>

A(3)

TypeError: 'list' object is not callable

>>> A

[1, 2, 3]

>>> **'''if we want to append it in A then we need to use compound assignment**

**operator like as follows:'''**

'if we want to append it in A then we need to use compound assignment operator like as follows:'

>>> A+=[10]

>>> A

[1, 2, 3, 10]

>>> '''We can repeat the list by using \*(Multiply operator), it will not multiply with the values, but it will repeat the values as shown below'''

>>> D=[10,20]

>>> D\*3

[10, 20, 10, 20, 10, 20]

>>> '''hence \* doesnt work as multiplication operator insted it works as an repeating operator'''

>>> '''However there will be no changes in list D, we again need to use compound assignment operator as shown below:'''

>>> D

[10, 20]

**>>> D\*=3**

>>> D

[10, 20, 10, 20, 10, 20]

>>>

**Part 3:**

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>> '''Modify List'''

>>> mylist=[10,20,30]

>>> mylist[0]=100

>>> mylist

[100, 20, 30]

>>> '''we can mention the index number and modify it'''

>>> '''we cannot put index which doesn't exist'''

>>> '''now suppose we want to remove any one value the we need to use remove function'''

>>> mylist.remove(20)

>>> mylist

[100, 30]

>>> '''now the question comes, what will happen if there is repeating values

in that case it will remove the first occurrence element'''

>>> list=[10,20,10,20,30]

>>> list

[10, 20, 10, 20, 30]

>>> list.remove(20)

>>> list

[10, 10, 20, 30]

>>> '''now if we remove 20 again then the second occurrence element will be moved'''

>>> '''element which doesn’t exist and if we try to do any manipulation with that then it will throw error'''

>>> '''if we want to arrange the list in ascending order then we need to use the function sort'''

>>> list.sort()

>>> list

[10, 10, 20, 30]

>>> e=[5,4,3,2,1]

>>> e.sort()

>>> e

[1, 2, 3, 4, 5]

>>> '''if we want to remove all the value/elements from the list then we need to use the function clear'''

>>> e.clear()

>>> e

[]

>>> e=[1,2,3,4]

>>> e

[1, 2, 3, 4]

>>> '''if we want to reverse then we need to use the function reverse'''

>>> e.reverse()

>>> e

[4, 3, 2, 1]

>>> f=[7,5,6,3,12]

>>> f.reverse()

>>> f

[12, 3, 6, 5, 7]

>>> **'''pop function by default removes the last element and returns it'''**

'pop function by default removes the last element and returns it'

>>> d.pop()

Traceback (most recent call last):

File "<pyshell#38>", line 1, in <module>

d.pop()

NameError: name 'd' is not defined

>>> f.pop()

7

>>> '''now the question is, whether the last element is deleted from the list, lets see by printing the list'''

>>> f

[12, 3, 6, 5]

>>> '''hence the last element is removed by using the pop function'''

>>> '''Difference in remove and pop function is

a) pop returns value but remove doesn't

b) pop returns value even if we don’t pass any value but remove needs value to be passed

c) in remove we pass the value which is needed to be removed, however in pop we can pass index which needs to be deleted'''

>>> x=f.pop(0)

>>> x

12

>>> f

[3, 6, 5]

>>> A=[10,20,30,40]

>>> A.remove(20)

>>> A

[10, 30, 40]

>>> A.pop(2)

40

>>> A

[10, 30]

>>> '''Now Suppose we want to know the index of a particular value then we use the function index'''

>>> A.index(30)

1

>>> A.index(70)

Traceback (most recent call last):

File "<pyshell#58>", line 1, in <module>

A.index(70)

ValueError: 70 is not in list

>>> '''Now suppose we are serching for the index which is repeating in list, then lets see below outcome what happens'''

>>> A=[10,20,30,20,40,50,60,70,70]

>>> A.index(20)

1

>>> '''we can also mention the index number from where the search needs to be started'''

>>> A.index(20,1)

1

>>> A.index(20,2)

3

>>> '''we can also pass the end index value to restrict the search till there'''

>>> A.index(20,0,3)

1

>>> '''now suppose we want to know how many times a value is repeated then we need to use the count function and pass the value which is repeating'''

>>> A.count(20)

2

>>> '''we can see the entire function available by help function'''

>>> help(A)

Help on list object:

class list(object)

| list(iterable=(), /)

|

| Built-in mutable sequence.

|

| If no argument is given, the constructor creates a new empty list.

| The argument must be an iterable if specified.

|

| Methods defined here:

|

| \_\_add\_\_(self, value, /)

| Return self+value.

|

| \_\_contains\_\_(self, key, /)

| Return key in self.

|

| \_\_delitem\_\_(self, key, /)

| Delete self[key].

|

| \_\_eq\_\_(self, value, /)

| Return self==value.

|

| \_\_ge\_\_(self, value, /)

| Return self>=value.

|

| \_\_getattribute\_\_(self, name, /)

| Return getattr(self, name).

|

| \_\_getitem\_\_(...)

| x.\_\_getitem\_\_(y) <==> x[y]

|

| \_\_gt\_\_(self, value, /)

| Return self>value.

|

| \_\_iadd\_\_(self, value, /)

| Implement self+=value.

|

| \_\_imul\_\_(self, value, /)

| Implement self\*=value.

|

| \_\_init\_\_(self, /, \*args, \*\*kwargs)

| Initialize self. See help(type(self)) for accurate signature.

|

| \_\_iter\_\_(self, /)

| Implement iter(self).

|

| \_\_le\_\_(self, value, /)

| Return self<=value.

|

| \_\_len\_\_(self, /)

| Return len(self).

|

| \_\_lt\_\_(self, value, /)

| Return self<value.

|

| \_\_mul\_\_(self, value, /)

| Return self\*value.

|

| \_\_ne\_\_(self, value, /)

| Return self!=value.

|

| \_\_repr\_\_(self, /)

| Return repr(self).

|

| \_\_reversed\_\_(self, /)

| Return a reverse iterator over the list.

|

| \_\_rmul\_\_(self, value, /)

| Return value\*self.

|

| \_\_setitem\_\_(self, key, value, /)

| Set self[key] to value.

|

| \_\_sizeof\_\_(self, /)

| Return the size of the list in memory, in bytes.

|

| append(self, object, /)

| Append object to the end of the list.

|

| clear(self, /)

| Remove all items from list.

|

| copy(self, /)

| Return a shallow copy of the list.

|

| count(self, value, /)

| Return number of occurrences of value.

|

| extend(self, iterable, /)

| Extend list by appending elements from the iterable.

|

| index(self, value, start=0, stop=2147483647, /)

| Return first index of value.

|

| Raises ValueError if the value is not present.

|

| insert(self, index, object, /)

| Insert object before index.

|

| pop(self, index=-1, /)

| Remove and return item at index (default last).

|

| Raises IndexError if list is empty or index is out of range.

|

| remove(self, value, /)

| Remove first occurrence of value.

|

| Raises ValueError if the value is not present.

|

| reverse(self, /)

| Reverse \*IN PLACE\*.

|

| sort(self, /, \*, key=None, reverse=False)

| Stable sort \*IN PLACE\*.

|

| ----------------------------------------------------------------------

| Static methods defined here:

|

| \_\_new\_\_(\*args, \*\*kwargs) from builtins.type

| Create and return a new object. See help(type) for accurate signature.

|

| ----------------------------------------------------------------------

| Data and other attributes defined here:

|

| \_\_hash\_\_ = None

>>>

**Assignment 6:**

1. **Write a python script to sort a list given by user**

'''Write a python script to sort a list given by user'''

l=[int (e) for e in input("Enter numbers seperated by comma: ").split(',')]

'''input function will return the string value and split function will

sepearate the single string into several small strings on the basis of

comma'''

'''also we need to transfer the string into int, hence we will be using int

'''

print(l)

sorted(l)

print(l)

'''sorted function returns sorted list but there will be no changes in the list'''

l=[int (e) for e in input("Enter numbers seperated by comma: ").split(',')]

print(l)

m=sorted(l)

print(m)

l=[int (e) for e in input("Enter numbers seperated by comma: ").split(',')]

print(l)

l.sort()

print(l)

'''Sorted function can be applied for any sequence whereas sort is used for

list purpose only'''

Output:

Enter numbers seperated by comma: 10,6,2,9,7

[10, 6, 2, 9, 7]

[10, 6, 2, 9, 7]

Enter numbers seperated by comma: 10,6,2,9,7

[10, 6, 2, 9, 7]

[2, 6, 7, 9, 10]

Enter numbers seperated by comma: 10,6,2,9,7

[10, 6, 2, 9, 7]

[2, 6, 7, 9, 10]

>>> A=[10,6,2,9,7]

>>> sorted(A)

[2, 6, 7, 9, 10]

>>> A

[10, 6, 2, 9, 7]

>>> A.sort

<built-in method sort of list object at 0x054AD0F8>

>>> A.sort()

>>> A

[2, 6, 7, 9, 10]

>>> A

[2, 6, 7, 9, 10]

>>>

1. **Write a python script to find max value in a list of integers.**

'''Write a python script to find greatest number in the list of integers given

by user'''

#input list

l=[int(e) for e in input("Enter numbers seperated by comma: ").split(',')]

#finding maximum value in the list

m=max(l)

#Display result

print("Greatest number in the list is",m)

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 6 q2.py ===

Enter numbers seperated by comma: 10,20,30,40,50,60,70,90,100,780

Greatest number in the list is 780

>>>

1. **Write a python script to calculate sum of all the integers of the list given by user.**

print("Enter a list of integers seperated by comma: ")

list=[int(x) for x in input().split(',')]

sum=0

for e in list:

sum+=e

print("sum is",sum)

Output:

Enter a list of integers seperated by comma:

10,20,30,40,50

sum is 150

>>>

**Write a python script to calculate sum of all the integers of the list given by user.**

list=[int(e) for e in input("Enter numbers seperated by comma: ").split(',')]

'''print("Enter a list of integers seperated by comma: ")

list=[int(x) for x in input().split(',')]'''

sum=0

for f in list:

sum+=f

print("sum is",sum)

Output:

Enter numbers seperated by comma: 10,20

sum is 30

>>>

1. **Write a python script to create a list of first N prime numbers. Values of N is given by user.**

'''python script to create a list of first N prime numbers.

Values of N is given by user.'''

n=int(input("Enter a natural number"))

x=2

l=[]

while n:

if all (x%i!=0 for i in range(2,x)):

l+=[x] #to append the prime numbers in the list(compound assignment)

n-=1

x+=1

'''all is a function which checks multiple

conditions, if all the conditions are true then all function will return true

otherwise false'''

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 6 q4.py ===

Enter a natural number10

>>> l

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29]

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 6 q4.py ===

Enter a natural number15

>>> l

[2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, 41, 43, 47]

>>>

1. **Write a python script to find to create a list of squares of list of first N natural numbers**

'''create a list of squares of list of first N natural numbers'''

l=[e\*\*2 for e in range(1,int(input("Enter a natural number"))+1)]

print(l)

Output:

Enter a natural number9

[1, 4, 9, 16, 25, 36, 49, 64, 81]

>>>

1. **Write a python script to print indices of all occurrences of a given element in list.**

'''python script to print indices of all occurrences of a given element in list'''

l=[eval (x) for x in input("Enter list elements").split(',')]

#eval funtion converts ths string into int type

element=eval(input("Enter element value"))

index=0

while index<len(l):

if l[index]==element:

print(index,end=' ')

index+=1

Output:

Enter list elements1,2,3,31,1,2,5,1,45,47,1

Enter element value1

0 4 7 10

>>>

**P.S: we can write int also instead of eval as bot can be used to convert the string data type into integer data type.**

1. **Write a python script to print distinct list elements along with their frequency of occurrence in the list.**

'''print distinct list elements along with their frequency of

occurrence in the list '''

l=[int (x) for x in input("Enter list elements").split(',')]

i=0

for e in l:

if l.index(e)==i:

print(e,"-----",l.count(e))

i+=1

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 6 q7.py ===

Enter list elements1,1,2,2,3,3,4,4

1 ----- 2

2 ----- 2

3 ----- 2

4 ----- 2

>>>

1. **Write a python script to print the sum of all even numbers and sum of all odd numbers in the list.**

'''print the sum of all even numbers and sum of all odd numbers in the list '''

print("Enter list of integers seperated by comma")

list1=[int(x) for x in input().split(',')]

sum\_of\_even,sum\_of\_odd=0,0

for e in list1:

if e%2==0:

sum\_of\_even+=e

else:

sum\_of\_odd+=e

print("Sum of all even elements is",sum\_of\_even)

print("sum of all odd elements is",sum\_of\_odd)

Output:

Enter list of integers seperated by comma

2,3,4,5,6,1,7,8,9,10

Sum of all even elements is 30

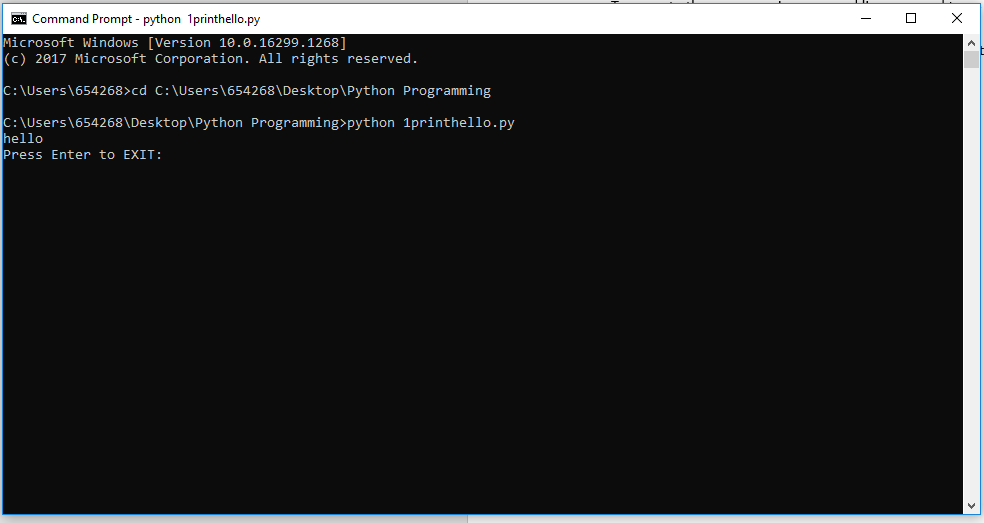
sum of all odd elements is 25

>>>

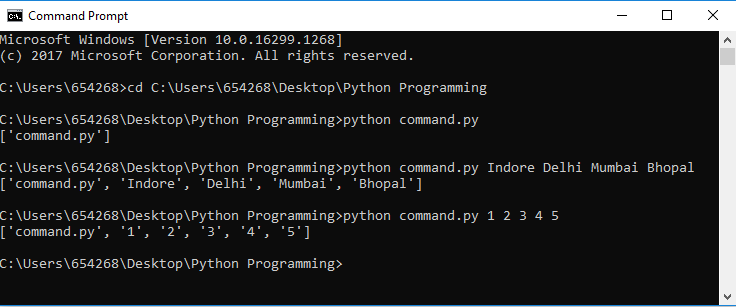
**Command line arguments:**

**Use of else with loop:**

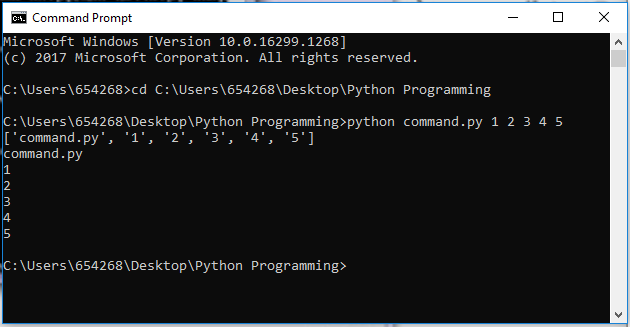
* When we execute a program by writing the program in command prompt is known as command line, we can pass argument through command line.
* To execute the program via command line we need to go to that path by the command cd path name
* Once we enter into the path we need to call the interpreter by typing python and then the program name



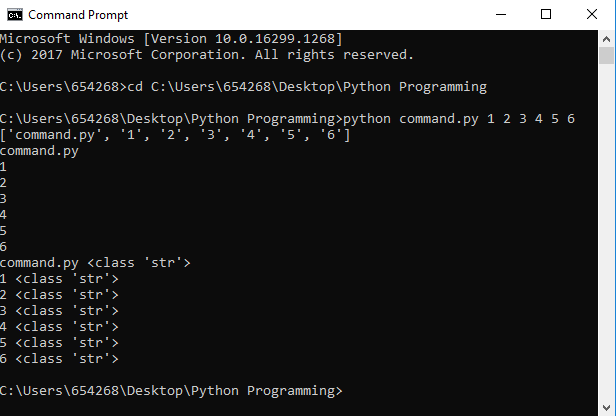
* We can pass arguments in command line also, like if we see the previous example we can pass arguments like Indore Bhopal Delhi which will be separated by space.
* However even if we pass argument it will print hello only because we haven’t done any coding in the program which will handle the arguments provided in command line, so we will have to understand how it can be done.\
* There is a module named as sys. It’s a predefined module which has a variable named as **argv** which is a short form of argument.
* This variable doesn’t needs to be created because it’s a predefined variable for the module.
* To use that predefined variable we need to write “from sys import argv “
* This predefined variable “argv” is of type list
* Hence when we pass the argument in command line it stores the value in argv, so the next line after “from sys import argv “ we need to print it.
* Let’s execute the below in command prompt.



* As we can see that the argument which we pass returns in string type with the file name in beginning.
* So the first argument will be the file name itself in the output.
* We can perform actions with the argument, we can apply for loop in the code but in that case as the print statement will run according to the loop so it will show the output in different lines as shown below:



* This output is also string type, to prove it we will have to change the code and need to print the type of x also as shown in the command screen below.



* So we can conclude that argv is a list of string.
* Now the code used for each command line is shown below:

'''Command line argument'''

from sys import argv

print(argv)

'''using for loop'''

for x in argv:

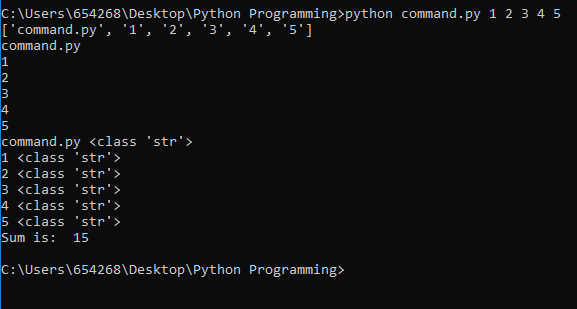
print(x)

'''type of x'''

for x in argv:

print(x, type(x))

* Now suppose we want to perform some sum operation with the argument passed, so for that first thing which we need to keep in mind is that the first element will be the file name and we will have to skip that.



**The code behind the sum operation is:**

'''perform sum operation with the arguments passed'''

#command line argument

from sys import argv

y=0

s=0

for x in argv:

if y==0:

y=1

else:

s=s+int(x)

print("Sum is: ",s)

**Strings in python: Part1**

Strings are bits of text. They can be defined as anything between quotes.

There are several ways to represent a string, those are as follows:

1. **‘GouraV’ single quotes**
2. **“GouraV” Double quotes**
3. **‘’’GouraV’’’ Single Quotes three times**
4. **“””GouraV””” Double quotes three times**

Suppose we want to print:

Don’t avoid practice

Here we can notice that in string there is a character (‘) which can be conflicting as (‘) defines the start and end of the string so here we will use double quotes, if we don’t use it then it will throw error as we can see in the below code:

>>> s1="Don't avoid practice"

>>> s1

"Don't avoid practice"

>>> print(s1)

Don't avoid practice

>>> s2='Don't avoid practice'

SyntaxError: invalid syntax

>>>

Now suppose we want to print [www.google.com](http://www.google.com) with double quotes i.e. we want to see double quotes also in data, we can do that by putting the string in double quote and we will use single quote to mark it as start data and end data.

>>> s3= '"www.google.com"'

>>> s3

'"www.google.com"'

>>> print(s3)

"www.google.com"

>>>

Now suppose we want to print Teacher’s day is “My Day”, in this case we need to use triple quote so that we can get both single quote and double quote in the data.

>>> s4='''Teacher's day is "My Day"'''

>>> print(s4)

Teacher's day is "My Day"

>>>

Triple quote has one more advantage that if we want to make multi line string then we can make it like as follows:

>>> s5='''MySirg.com is

run by

Saurabh Shukla'''

>>> print(s5)

MySirg.com is

run by

Saurabh Shukla

>>>

We can check the length of the string by using a predefined function ‘len’

>>> s6="mysirg educational services"

>>> len(s6)

27

>>>

It takes the space also as a character and counts it.

We can also find the index of the string as :

>>> s6="mysirg educational services"

>>> len(s6)

27

>>> s7="mysirg education services"

>>> s7.index('s')

2

>>> s7.index('e')

7

>>> s7.index('education')

7

>>>

It always print the first occurrence of the element.

We can also print the number of occurrence of the character as:

>>> s7="mysirg education services"

>>> s7.count('e')

3

>>>

**Strings in python: part 2**

As we know that in python indexing starts from 0 and increases sequentially, also a space in between word is considered as a character and hence has an index value.

We are also aware of negative indexing that starts from -1 and is counted from Right to Left.

>>> s8="Gourav Prasad"

>>> s8[-2]

'a'

>>>

Slicing Operator:

[Beginning:end:step] #syntax

All the three are optional, we can simply put empty.

Suppose we want to pick a particular items from a particular index to certain part the we can mention the beginning index which is included and the end index which is excluded in the output and then we can put step as 1 or 2 which will increase the value of index by 1 or 2 respectively.

For Example:

>>> s1="mysirg education services"

>>> s1[7:16]

'education'

>>> print(s1[7:16])

education

>>>

If we simply do slicing then the output will be in quotes, if we want without quotes then we will have to print the statement.

One important thing to note is that beginning value is included and end value is always excluded, hence if we put the beginning value as 7 and end value as 16 then it will print from index 7 to 15.

We can put the value as negative also in end value as shown in example below:

>>> s1[2:-2]

'sirg education servic'

>> print(s1[2:-2])

sirg education servic

>>> s1[-5:23]

'vic'

When the slicing happens it happens from left to right only, now suppose we are doing the slicing as [-5:8] in these case the slicing need to be done from right to left, in such scenario we will put negative value to step so that slicing happens in Right to Left directions.

>>> s1[-5:8:-1]

'vres noitacu'

>>>

This will print in complete reverse format from the string passed.

Now let’s see an example with different step values:

>>> s1[2:23:2]

'sr dcto evc'

>>

Step value decides how many indexes gap needs to be taken.

**Now the question is how to reverse a string?**

First thing to note is that there is no inbuilt method or function in python to reverse a string.

The only possible way to do is to do via slicing method.

>>> s1

'mysirg education services'

>>> s1[25:0:-1]

'secivres noitacude grisy'

>>> s1[24::-1]

'secivres noitacude grisym'

>>> s1[len(s1)::-1]

'secivres noitacude grisym'

>>> s1[::-1]

'secivres noitacude grisym'

>>>

We can do the slicing in different ways.

In first way we can see that m is missing as the end value assigned is 0 and it will be excluded, so if we want the m also then we can ignore the end value as we did in second way, in such format it will take the end value till the string is present.

We can also use the function len which is used in third way.

We can keep the beginning and end value blank and simply put the step as -1 then it will reverse the whole string by taking the range of the string as it is.

There are several other functions in strings as:

>>> s1

'mysirg education services'

>>> s1.upper()

'MYSIRG EDUCATION SERVICES'

>>> s1.lower()

'mysirg education services'

>>> s1.startswith("my")

True

>>> s1.endswith("services")

True

>>> s1.startswith("abc")

False

>>>

The function startswith and endswith return the value as true or false.

It check whether it starts and endswith the argument passed in it or not.

There is another special function that is split, which separates the data on the basis of space

>>> s1

'mysirg education services'

>>> s1.split(' ')

['mysirg', 'education', 'services']

>>>

Hence it is cleared that split function returns the list of strings.

>>> help(str)

Help on class str in module builtins:

class str(object)

| str(object='') -> str

| str(bytes\_or\_buffer[, encoding[, errors]]) -> str

|

| Create a new string object from the given object. If encoding or

| errors is specified, then the object must expose a data buffer

| that will be decoded using the given encoding and error handler.

| Otherwise, returns the result of object.\_\_str\_\_() (if defined)

| or repr(object).

| encoding defaults to sys.getdefaultencoding().

| errors defaults to 'strict'.

|

| Methods defined here:

|

| \_\_add\_\_(self, value, /)

| Return self+value.

|

| \_\_contains\_\_(self, key, /)

| Return key in self.

|

| \_\_eq\_\_(self, value, /)

| Return self==value.

|

| \_\_format\_\_(self, format\_spec, /)

| Return a formatted version of the string as described by format\_spec.

|

| \_\_ge\_\_(self, value, /)

| Return self>=value.

|

| \_\_getattribute\_\_(self, name, /)

| Return getattr(self, name).

|

| \_\_getitem\_\_(self, key, /)

| Return self[key].

|

| \_\_getnewargs\_\_(...)

|

| \_\_gt\_\_(self, value, /)

| Return self>value.

|

| \_\_hash\_\_(self, /)

| Return hash(self).

|

| \_\_iter\_\_(self, /)

| Implement iter(self).

|

| \_\_le\_\_(self, value, /)

| Return self<=value.

|

| \_\_len\_\_(self, /)

| Return len(self).

|

| \_\_lt\_\_(self, value, /)

| Return self<value.

|

| \_\_mod\_\_(self, value, /)

| Return self%value.

|

| \_\_mul\_\_(self, value, /)

| Return self\*value.

|

| \_\_ne\_\_(self, value, /)

| Return self!=value.

|

| \_\_repr\_\_(self, /)

| Return repr(self).

|

| \_\_rmod\_\_(self, value, /)

| Return value%self.

|

| \_\_rmul\_\_(self, value, /)

| Return value\*self.

|

| \_\_sizeof\_\_(self, /)

| Return the size of the string in memory, in bytes.

|

| \_\_str\_\_(self, /)

| Return str(self).

|

| capitalize(self, /)

| Return a capitalized version of the string.

|

| More specifically, make the first character have upper case and the rest lower

| case.

|

| casefold(self, /)

| Return a version of the string suitable for caseless comparisons.

|

| center(self, width, fillchar=' ', /)

| Return a centered string of length width.

|

| Padding is done using the specified fill character (default is a space).

|

| count(...)

| S.count(sub[, start[, end]]) -> int

|

| Return the number of non-overlapping occurrences of substring sub in

| string S[start:end]. Optional arguments start and end are

| interpreted as in slice notation.

|

| encode(self, /, encoding='utf-8', errors='strict')

| Encode the string using the codec registered for encoding.

|

| encoding

| The encoding in which to encode the string.

| errors

| The error handling scheme to use for encoding errors.

| The default is 'strict' meaning that encoding errors raise a

| UnicodeEncodeError. Other possible values are 'ignore', 'replace' and

| 'xmlcharrefreplace' as well as any other name registered with

| codecs.register\_error that can handle UnicodeEncodeErrors.

|

| endswith(...)

| S.endswith(suffix[, start[, end]]) -> bool

|

| Return True if S ends with the specified suffix, False otherwise.

| With optional start, test S beginning at that position.

| With optional end, stop comparing S at that position.

| suffix can also be a tuple of strings to try.

|

| expandtabs(self, /, tabsize=8)

| Return a copy where all tab characters are expanded using spaces.

|

| If tabsize is not given, a tab size of 8 characters is assumed.

|

| find(...)

| S.find(sub[, start[, end]]) -> int

|

| Return the lowest index in S where substring sub is found,

| such that sub is contained within S[start:end]. Optional

| arguments start and end are interpreted as in slice notation.

|

| Return -1 on failure.

|

| format(...)

| S.format(\*args, \*\*kwargs) -> str

|

| Return a formatted version of S, using substitutions from args and kwargs.

| The substitutions are identified by braces ('{' and '}').

|

| format\_map(...)

| S.format\_map(mapping) -> str

|

| Return a formatted version of S, using substitutions from mapping.

| The substitutions are identified by braces ('{' and '}').

|

| index(...)

| S.index(sub[, start[, end]]) -> int

|

| Return the lowest index in S where substring sub is found,

| such that sub is contained within S[start:end]. Optional

| arguments start and end are interpreted as in slice notation.

|

| Raises ValueError when the substring is not found.

|

| isalnum(self, /)

| Return True if the string is an alpha-numeric string, False otherwise.

|

| A string is alpha-numeric if all characters in the string are alpha-numeric and

| there is at least one character in the string.

|

| isalpha(self, /)

| Return True if the string is an alphabetic string, False otherwise.

|

| A string is alphabetic if all characters in the string are alphabetic and there

| is at least one character in the string.

|

| isascii(self, /)

| Return True if all characters in the string are ASCII, False otherwise.

|

| ASCII characters have code points in the range U+0000-U+007F.

| Empty string is ASCII too.

|

| isdecimal(self, /)

| Return True if the string is a decimal string, False otherwise.

|

| A string is a decimal string if all characters in the string are decimal and

| there is at least one character in the string.

|

| isdigit(self, /)

| Return True if the string is a digit string, False otherwise.

|

| A string is a digit string if all characters in the string are digits and there

| is at least one character in the string.

|

| isidentifier(self, /)

| Return True if the string is a valid Python identifier, False otherwise.

|

| Use keyword.iskeyword() to test for reserved identifiers such as "def" and

| "class".

|

| islower(self, /)

| Return True if the string is a lowercase string, False otherwise.

|

| A string is lowercase if all cased characters in the string are lowercase and

| there is at least one cased character in the string.

|

| isnumeric(self, /)

| Return True if the string is a numeric string, False otherwise.

|

| A string is numeric if all characters in the string are numeric and there is at

| least one character in the string.

|

| isprintable(self, /)

| Return True if the string is printable, False otherwise.

|

| A string is printable if all of its characters are considered printable in

| repr() or if it is empty.

|

| isspace(self, /)

| Return True if the string is a whitespace string, False otherwise.

|

| A string is whitespace if all characters in the string are whitespace and there

| is at least one character in the string.

|

| istitle(self, /)

| Return True if the string is a title-cased string, False otherwise.

|

| In a title-cased string, upper- and title-case characters may only

| follow uncased characters and lowercase characters only cased ones.

|

| isupper(self, /)

| Return True if the string is an uppercase string, False otherwise.

|

| A string is uppercase if all cased characters in the string are uppercase and

| there is at least one cased character in the string.

|

| join(self, iterable, /)

| Concatenate any number of strings.

|

| The string whose method is called is inserted in between each given string.

| The result is returned as a new string.

|

| Example: '.'.join(['ab', 'pq', 'rs']) -> 'ab.pq.rs'

|

| ljust(self, width, fillchar=' ', /)

| Return a left-justified string of length width.

|

| Padding is done using the specified fill character (default is a space).

|

| lower(self, /)

| Return a copy of the string converted to lowercase.

|

| lstrip(self, chars=None, /)

| Return a copy of the string with leading whitespace removed.

|

| If chars is given and not None, remove characters in chars instead.

|

| partition(self, sep, /)

| Partition the string into three parts using the given separator.

|

| This will search for the separator in the string. If the separator is found,

| returns a 3-tuple containing the part before the separator, the separator

| itself, and the part after it.

|

| If the separator is not found, returns a 3-tuple containing the original string

| and two empty strings.

|

| replace(self, old, new, count=-1, /)

| Return a copy with all occurrences of substring old replaced by new.

|

| count

| Maximum number of occurrences to replace.

| -1 (the default value) means replace all occurrences.

|

| If the optional argument count is given, only the first count occurrences are

| replaced.

|

| rfind(...)

| S.rfind(sub[, start[, end]]) -> int

|

| Return the highest index in S where substring sub is found,

| such that sub is contained within S[start:end]. Optional

| arguments start and end are interpreted as in slice notation.

|

| Return -1 on failure.

|

| rindex(...)

| S.rindex(sub[, start[, end]]) -> int

|

| Return the highest index in S where substring sub is found,

| such that sub is contained within S[start:end]. Optional

| arguments start and end are interpreted as in slice notation.

|

| Raises ValueError when the substring is not found.

|

| rjust(self, width, fillchar=' ', /)

| Return a right-justified string of length width.

|

| Padding is done using the specified fill character (default is a space).

|

| rpartition(self, sep, /)

| Partition the string into three parts using the given separator.

|

| This will search for the separator in the string, starting at the end. If

| the separator is found, returns a 3-tuple containing the part before the

| separator, the separator itself, and the part after it.

|

| If the separator is not found, returns a 3-tuple containing two empty strings

| and the original string.

|

| rsplit(self, /, sep=None, maxsplit=-1)

| Return a list of the words in the string, using sep as the delimiter string.

|

| sep

| The delimiter according which to split the string.

| None (the default value) means split according to any whitespace,

| and discard empty strings from the result.

| maxsplit

| Maximum number of splits to do.

| -1 (the default value) means no limit.

|

| Splits are done starting at the end of the string and working to the front.

|

| rstrip(self, chars=None, /)

| Return a copy of the string with trailing whitespace removed.

|

| If chars is given and not None, remove characters in chars instead.

|

| split(self, /, sep=None, maxsplit=-1)

| Return a list of the words in the string, using sep as the delimiter string.

|

| sep

| The delimiter according which to split the string.

| None (the default value) means split according to any whitespace,

| and discard empty strings from the result.

| maxsplit

| Maximum number of splits to do.

| -1 (the default value) means no limit.

|

| splitlines(self, /, keepends=False)

| Return a list of the lines in the string, breaking at line boundaries.

|

| Line breaks are not included in the resulting list unless keepends is given and

| true.

|

| startswith(...)

| S.startswith(prefix[, start[, end]]) -> bool

|

| Return True if S starts with the specified prefix, False otherwise.

| With optional start, test S beginning at that position.

| With optional end, stop comparing S at that position.

| prefix can also be a tuple of strings to try.

|

| strip(self, chars=None, /)

| Return a copy of the string with leading and trailing whitespace remove.

|

| If chars is given and not None, remove characters in chars instead.

|

| swapcase(self, /)

| Convert uppercase characters to lowercase and lowercase characters to uppercase.

|

| title(self, /)

| Return a version of the string where each word is titlecased.

|

| More specifically, words start with uppercased characters and all remaining

| cased characters have lower case.

|

| translate(self, table, /)

| Replace each character in the string using the given translation table.

|

| table

| Translation table, which must be a mapping of Unicode ordinals to

| Unicode ordinals, strings, or None.

|

| The table must implement lookup/indexing via \_\_getitem\_\_, for instance a

| dictionary or list. If this operation raises LookupError, the character is

| left untouched. Characters mapped to None are deleted.

|

| upper(self, /)

| Return a copy of the string converted to uppercase.

|

| zfill(self, width, /)

| Pad a numeric string with zeros on the left, to fill a field of the given width.

|

| The string is never truncated.

|

| ----------------------------------------------------------------------

| Static methods defined here:

|

| \_\_new\_\_(\*args, \*\*kwargs) from builtins.type

| Create and return a new object. See help(type) for accurate signature.

|

| maketrans(x, y=None, z=None, /)

| Return a translation table usable for str.translate().

|

| If there is only one argument, it must be a dictionary mapping Unicode

| ordinals (integers) or characters to Unicode ordinals, strings or None.

| Character keys will be then converted to ordinals.

| If there are two arguments, they must be strings of equal length, and

| in the resulting dictionary, each character in x will be mapped to the

| character at the same position in y. If there is a third argument, it

| must be a string, whose characters will be mapped to None in the result.

>>>

**Assignment 7:**

1. **Write a python script to calculate the length of the string. String is given by user.**

'''python script to calculate the length of the string. String is given by user'''

s=input("Enter the string: ")

print("Length of the string is: ",len(s))

Output:

Enter the string: my educational services

Length of the string is: 23

>>

Enter the string: Gourav Prasad Nunia

Length of the string is: 19

>>>

1. **Write a python script to count vowels in a given string.**

'''python script to count vowels in a given string'''

s=input("Enter the string: ")

l=len(s)

v="aeiouAEIOU"

count=0

for e in s:

if e in v: #we cannot write v in e, as that will check complete aeiouAEIOU in the string

count+=1

print("Total vowels:",count)

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q2.py ===

Enter the string: gourav

Total vowels: 3

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q2.py ===

Enter the string: goUAav

Total vowels: 4

>>>

1. **Write a python script to reverse a given string.**

'''Write a python script to reverse a given string'''

s=input("Enter a string: ")

print("The reverse is: ",s[::-1])

output:

Enter a string: My name is Gourav Prasad Nunia

The reverse is: ainuN dasarP varuoG si eman yM

1. **Write a python script to count the occurrence of a character.**

''''python script to count the occurrence of a character'''

s=input("Enter a string: ")

t=input("Enter the character you want to search the occurence for: ")

count=0

for e in s:

if e in t:

count+=1

print("Number of occurence is: ",count)

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q4.py ===

Enter a string: gourav

Enter the character you want to search the occurence for: g

Number of occurence is: 1

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q4.py ===

Enter a string: gourav prasad

Enter the character you want to search the occurence for: a

Number of occurence is: 3

>>>

1. **Write a python script to count words in a given string.**

'''python script to count words in a given string'''

s=input("Enter a string: ")

print("Total words is", s.count(' ')+1)

#the above program is not robust

'''because if a user types one word and give 4 spaces and the type the

second word and gives 5 spaces, in that scenario program will not show

actual results, we need to make a robust program'''

s=input("Enter the string: ").strip() #strip function will remove the spaces from front and back

print("total words: ", len([i for i in range(0,len(s)) if s[i]==' ' and s[i+1]!=' '])+1)

output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q5.py ===

Enter a string: dsadsa asdsad asdsad\

Total words is 5

Enter the string: mysirg is great and best

total words: 5

>>>

1. **Write a python script to arrange words of a given string in alphabetical order.**

for s in sorted(input("Enter a string: ").split()):

print(s,end=' ')

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q6.py ===

Enter a string: mjhjh gbhj

gbhj mjhjh

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q6.py ===

Enter a string: gourav prasad nunia

gourav nunia prasad

>>>

1. **Write a python script to check whether the given string is palindrome or not?**

'''python script to check whether the given string is palindrome or not'''

s=input("Enter a string: ")

if s==s[::-1]:

print("String is pallindrome")

else:

print("string is not pallindrome")

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q7.py ===

Enter a string: papa

string is not pallindrome

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q7.py ===

Enter a string: abcba

String is pallindrome

>>>

1. **Write a python script to remove duplicate characters from a given string.**

If a particular string comes multiple times then we will have to remove that and keep that only once.

We need to one thing about ‘set’ is that we cannot give same entries multiple times however in ‘list’ we can.

Hence we need to apply the concept of set here.

'''python script to remove duplicate characters from a given string'''

names=[]

n=int(input("How many names you want to enter?: "))

for i in range(n):

print(i+1,"Enter name: ") #to print it sequentially from 1

names.append(input())

s=set(names)

names=list(s)

for x in s:

print(x)

output:

How many names you want to enter?: 2

1 Enter name:

gourav

2 Enter name:

gourav

gourav

>>>

1. **Write a python script to find a pattern in a given string.**

'''python script to find a pattern in a given string'''

s=input("Enter a string:")

p=input("Enter a pattern to search in the string: ")

if p in s:

print(p,"is in the string",s)

else:

print(p,"is not in the string",s)

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q9.py ===

Enter a string:Gourav Prasad Nunia

Enter a pattern to search in the string: Prasad

Prasad is in the string Gourav Prasad Nunia

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q9.py ===

Enter a string:gourav prasad nunia

Enter a pattern to search in the string: Prasad

Prasad is not in the string gourav prasad nunia

>>>

1. **Write a python script to check the count occurrence of a given pattern in a given string.**

'''python script to check the count occurrence of a given pattern in a given string'''

s=input("Enter a string:")

p=input("Enter a pattern to search in the string:")

print("Total occurences of",p,"in",s,s.count(p))

Output:

== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 7 q10.py ==

Enter a string: Gourav is learning python to progress in python

Enter a pattern to search in the string:python

Total occurences of python in Gourav is learning python to progress in python 2

>>>

**Tuple - part1:**

* Like list, tuple is also a list of sequence.
* Unlike square bracket in list we use parenthesis or round bracket for tuple

List []

Tuple ()

* Similar to list it is also indexed from 0
* There are significant differences in between list and tuple:

1. List is Mutable (Changes can be made) i.e. we can add modify or remove elements.
2. Tuple is Immutable (Changes cannot be made).

* Create Tuple:

>>> t=()

>>> type(t)

<class 'tuple'>

>>> t

()

>>> t=(10)

>>> type(t)

<class 'int'>

>>> '''if we write t=(), it will not be treated as tuple, rather it will show

as int data type as shown above, hence the correct method is t=(10,)'''

>>> t=(10,)

>>> type(t)

<class 'tuple'>

>>> '''Hence we can create tuple by simply keeping parenthesis or if we want a

single value then comma should be there in the end'''

>>> '''However in case if we are assigning multiple values then we don't need

comma at the end'''

>>> t=(1,2,3)

>>> type(t)

<class 'tuple'>

>>> '''we can also create tuple by simply assigning value without parenthesis'''

>>> t=1,2,3,4,5

>>> type(t)

<class 'tuple'>

>>> 'we cannot pass value in parenthesis and expect to be converted into tuple'

>>> t=tuple(10,20,30,40)

Traceback (most recent call last):

File "<pyshell#24>", line 1, in <module>

t=tuple(10,20,30,40)

TypeError: tuple expected at most 1 arguments, got 4

>>> '''if we want to do that then we need to use two parenthesis'''

>>> t=tuple((10,20,30,40))

>>> type(t)

<class 'tuple'>

>>> '''like list we can keep heterogeneous data also'''

>>> t=(10,2.5,"ABC")

>>> type(t)

<class 'tuple'>

>>> '''As the tupple works on indexing concept also we cann access the value by passing the index value'''

>>> t[0]

10

>>> t[1]

2.5

>>> t[2]

'ABC'

>>> '''we cannot go out of index as in list'''

>>> t[3]

Traceback (most recent call last):

File "<pyshell#36>", line 1, in <module>

t[3]

IndexError: tuple index out of range

>>> '''we can access the index via several ways:1:via loop(while and for)

2:via slicing, we will see all process one by one'''

>>> #loop(while)

>>> t=(10,20,30,40)

>>> i=0

>>> while i<len(t):

print(t[i])

i+=1

10

20

30

40

>>> for x in t:

print(x)

10

20

30

40

>>> #slicing

>>> t

(10, 20, 30, 40)

>>> t[0::]

(10, 20, 30, 40)

>>> '''we can add into tuple as:'''

>>> t=t+(1,2,3,4)

>>> t

(10, 20, 30, 40, 1, 2, 3, 4)

>>> t[0::2] #if we want only alternate numbers

(10, 30, 1, 3)

>>> t[2:5]

(30, 40, 1)

>>>

**Tuple-Part 2:**

'Tuple (Packing)'

>>> '''if we have 3 variable and values are assigned to it and

if we want it to be assigned in a single entity then we need to do

packing of different variables, process is as follows:'''

>>> a=1

>>> b=2

>>> c=3

>>> t=a,b,c

>>> t

(1, 2, 3)

>>> type(t)

<class 'tuple'>

‘Tuple(unpacking)’

>>> '''Similarly on contrary to packing there is unpacking, if we

want to assign the value in tuple to different variable then we need

to use unpacking concept shown as below:'''

>>> t=(10,20,30)

>>> a,b,c=t

>>> a

10

>>> b

20

>>> c

30

>>> type(a)

<class 'int'>

>>>

'Concatenation of tuple'

>>> a=1,2,3

>>> b=10,20

>>> a+b

(1, 2, 3, 10, 20)

>>> a

(1, 2, 3)

>>> b

(10, 20)

>>> '''there is no change in actual value of tuple a and b'''

'Repetition operator'

>>> t=1,2,3

>>> t\*2

(1, 2, 3, 1, 2, 3)

>>> t

(1, 2, 3)

>>> '''there is no change in actual value of tuple t'''

>>> '''if we want to make changes in t also then we need to use

Compound assignment operator'''

>>> t\*=2

>>> t

(1, 2, 3, 1, 2, 3)

>>>

**Tuple-Part3:**

Input values to tuple:

2 ways:

1. X=tuple(input())
2. Y=eval(input())

#eval is a function in which we can write any expression and that expression is evaluated by eval function, whatever expression we give should be valid python statement.

Eval function operates only when the expression is valid as per the python standards, hence we can say it’s a kind of program inside a program.

If we pass value in eval like eval(1,2,3) it will convert it into tuple, whereas if we pass the value as eval[1,2,3] it will convert it into list.

In case of eval converting into tuple, parenthesis is not mandatory to provide, however in case of list conversion square bracket is necessary.

Hence it is better to use eval function when we want to input data from the user.

All the above explanation is shown in shell below:

>>> x=input("Enter a tuple")

Enter a tuple1,2,3

>>> x

'1,2,3'

>>> type(x)

<class 'str'>

>>> x=tuple(input("Enter a tuple"))

Enter a tuple1,2,3

>>> x

('1', ',', '2', ',', '3')

>>> type(x)

<class 'tuple'>

>>> '''if we dont want the commas then we have to use following code'''

>>> eval("5+2")

7

>>> x=eval(input("Enter a tuple: "))

Enter a tuple: 1,2,3

>>> x

(1, 2, 3)

>>> type(x)

<class 'tuple'>

>>> y=eval(input("Enter a list:"))

Enter a list:[1,2,3]

>>> y

[1, 2, 3]

>>> type(y)

<class 'list'>

>>>

**A simple program on tuple:**

'''Sum of the values in tuple'''

x=eval(input("Enter a tuple"))

s=0

for e in x:

s=s+e

print("Sum is",s)

**Output:**

Enter a tuple1,2,3,4,5,6

Sum is 21

>>>

**Few Functions in Tuple:**

1. len: to find the number of elements in tuple

>>> x=(1,2,3,4)

>>> len(x)

4

>>>

1. count: to find the occurrence of a particular element

>>> x=(1,2,3,4)

>>> len(x)

4

>>> x=10,20,30,10,10,20,10

>>> x.count(10)

4

>>> x.count(20)

2

>>>

1. index: to find the value of a particular index

>>> x=10,20,30,40,10

>>> x.index(10)

0

>>> x.index(20)

1

>>>it return the index value of first occurrence if there are repeated values.

>>> x.index(100)

Traceback (most recent call last):

File "<pyshell#24>", line 1, in <module>

x.index(100)

ValueError: tuple.index(x): x not in tuple

>>> '''to avoid the above error we can use the following code'''

>>> if 100 in x:

print("Index is",x.index(100))

>>> if 10 in x:

print("Index is",x.index(10))

Index is 0

>>>

1. max
2. min

>>> x=10,20,30,40,50

>>> max(x)

50

>>> min(x)

10

>>>

We can see all functions under tuple as:

>>> help(x)

Help on tuple object:

class tuple(object)

| tuple(iterable=(), /)

|

| Built-in immutable sequence.

|

| If no argument is given, the constructor returns an empty tuple.

| If iterable is specified the tuple is initialized from iterable's items.

|

| If the argument is a tuple, the return value is the same object.

|

| Methods defined here:

|

| \_\_add\_\_(self, value, /)

| Return self+value.

|

| \_\_contains\_\_(self, key, /)

| Return key in self.

|

| \_\_eq\_\_(self, value, /)

| Return self==value.

|

| \_\_ge\_\_(self, value, /)

| Return self>=value.

|

| \_\_getattribute\_\_(self, name, /)

| Return getattr(self, name).

|

| \_\_getitem\_\_(self, key, /)

| Return self[key].

|

| \_\_getnewargs\_\_(self, /)

|

| \_\_gt\_\_(self, value, /)

| Return self>value.

|

| \_\_hash\_\_(self, /)

| Return hash(self).

|

| \_\_iter\_\_(self, /)

| Implement iter(self).

|

| \_\_le\_\_(self, value, /)

| Return self<=value.

|

| \_\_len\_\_(self, /)

| Return len(self).

|

| \_\_lt\_\_(self, value, /)

| Return self<value.

|

| \_\_mul\_\_(self, value, /)

| Return self\*value.

|

| \_\_ne\_\_(self, value, /)

| Return self!=value.

|

| \_\_repr\_\_(self, /)

| Return repr(self).

|

| \_\_rmul\_\_(self, value, /)

| Return value\*self.

|

| count(self, value, /)

| Return number of occurrences of value.

|

| index(self, value, start=0, stop=2147483647, /)

| Return first index of value.

|

| Raises ValueError if the value is not present.

|

| ----------------------------------------------------------------------

| Static methods defined here:

|

| \_\_new\_\_(\*args, \*\*kwargs) from builtins.type

| Create and return a new object. See help(type) for accurate signature.

>>>

**Assignment 8:**

1. **Write a python script to calculate the average of tuple values. Assuming values are of int type only.**

'''calculate the average of tuple values. Assuming values are of int type only'''

'''n=int(input("Enter the number of values you want to enter: "))'''

t=eval(input("Enter the values: "))

s=0

for e in t:

s+=e

avg=s/len(t)

print("Average of tuple values is",avg)

Output:

Enter the values: 10,20,30,40,50

Average of tuple values is 30.0

>>>

1. **Write a python script to sort a tuple.**

'''sort a tuple'''

t=eval(input("Enter the values: "))

m=tuple(sorted(t))

print(m)

Output:

Enter the values: 10,9,8,7,6,5,4,3,2,1

(1, 2, 3, 4, 5, 6, 7, 8, 9, 10)

>>> type(t)

<class 'tuple'>

>>> type(m)

<class 'tuple'>

>>>

1. **Write a python script to merge two sorted tuples.**

'''merge two sorted tuples.'''

t=eval(input("Enter the values of first tuple: "))

m=tuple(sorted(t))

u=eval(input("Enter the values of second tuple: "))

n=tuple(sorted(u))

print("sorted list of two tuple values is",m+n)

Output:

Enter the values of first tuple: 4,3,2,1

Enter the values of second tuple: 1,2,3,4

sorted list of two tuple values is (1, 2, 3, 4, 1, 2, 3, 4)

1. **Write a python script to reverse a tuple**

Before starting the program please keep in mind that in list we have reverse function which will change the value of list as it is mutable, on the other hand tuple is immutable.

List is a kind of constructor which can keep any kind of sequence.

'''reverse a tuple'''

t=eval(input("Enter the tuple values to be reversed: "))

l=list(t)

l.reverse()

t=tuple(l)

print("Reverse of tuple is:",t)

Output:

Enter the tuple values to be reversed: 1,2,3,4

Reverse of tuple is: (4, 3, 2, 1)

>>>

1. **Write a python script to count elements in the tuple.**

'''count elements in the tuple'''

t=eval(input("Enter the values of tuple: "))

print("Number of elements in the tuple is: ",len(t))

Output:

Enter the values of tuple: 1,2,3,4,5,6,7,8

Number of elements in the tuple is: 8

>>>

1. **Write a python script to find the sum of tuple elements.**

'''sum of tuple elements'''

t=eval(input("Enter the tuple elements:"))

s=0

for e in t:

s=s+e

print("Sum of tuple elements",t, "is",s)

Output:

Enter the tuple elements:10,20,30,40,50

Sum of tuple elements (10, 20, 30, 40, 50) is 150

>>>

1. **Write a python script to create tuples with homogenous elements from a tuple containing heterogeneous elements.**

'''create tuples with homogenous elements from a tuple containing heterogeneous elements'''

t=eval(input("Enter the tuple elements:"))

l=list(t)

for i in l:

if type(i) is str:

l.remove(i)

t=tuple(l)

print("Elements excluding string type is:",t)

Output:

Enter the tuple elements:10,2.5,"hello"

Elements excluding string type is: (10, 2.5)

>>>

1. **Write a python script to compare two tuples, whether they contain the same element in any order or not.**
2. **Write a python script to compare two tuples, whether they contain the same element in same order or not.**

'''compare two tuples, whether they contain the same element in same order or not'''

t1=eval(input("Enter the first tuple: "))

t2=eval(input("Enter the second tuple: "))

if t1==t2:

print("Tuples are same and are in same order")

else:

print("Tuples are not same")

Output:

.

Enter the first tuple: 10,20,30,40

Enter the second tuple: 10,20,30,40

Tuples are same and are in same order

>>>

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 8 q9.py ===

Enter the first tuple: 10,20,30,40

Enter the second tuple: 10,20,30,40,50

Tuples are not same

>>>

1. **Write a python script to find the greatest number from a tuple of int values.**

'''find the greatest number from a tuple of int values'''

t=eval(input("Enter the first tuple: "))

print("Maximum value in the entered tuple is", max(t))

Output:

Enter the first tuple: 10,20,30,40

Maximum value in the entered tuple is 40

1. **Write a python script to check whether a tuple is a subset of another tuple or not.**

'''check whether a tuple is a subset of another tuple or not'''

t1=eval(input("Enter the first tuple: "))

t2=eval(input("Enter the second tuple: "))

print("t2 is a subset of t1" if all(e in t1 for e in t2) else "t1 is subset of t2" if all(e in t2 for e in t1) else "none of the tuple is the subset of anyother tuple")

Output:

== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 8 q11.py ==

Enter the first tuple: 1,2,3,4,5

Enter the second tuple: 2,3,6

none of the tuple is the subset of anyother tuple

>>>

== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 8 q11.py ==

Enter the first tuple: 1,2,3

Enter the second tuple: 1,2

t2 is a subset of t1

>>>

== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 8 q11.py ==

Enter the first tuple: 1,2

Enter the second tuple: 1,2,3,4

t1 is subset of t2

>>>

1. **Write a python script to calculate the frequency of elements of tuple.**

'''calculate the frequency of elements of tuple'''

t=eval(input("Enter the first tuple: "))

i=0

for e in t:

if t.index(e)==i:

print("frequency of",e,"is",t.count(e))

i+=1

Output:

Enter the first tuple: 1,2,3,4,1,1,3,2,3

frequency of 1 is 3

frequency of 2 is 2

frequency of 3 is 3

frequency of 4 is 1

>>>

**Set : Part1**

* We have studied list, string, tuple, and range in which we can keep repetitive data.
* In set we cannot keep repetitive data or duplicate values, however if we assign multiple duplicate values then it will keep only one value.
* Representation: s={10,20,30}
* As we have seen in list, string, tuple the values use to get assigned as per the order or in sequence of index, in case of set there is no concept of index, value can be assigned randomly in any index or we can say that it doesn’t maintain any order, hence there is no use of subscript operator. The subscript operator is defined as square brackets []
* We cannot use slicing operator too.
* It is mutable i.e. we can modify the values of sets as we use to do in list.
* If we create an empty set like s={}, it will not throw error but that is actual dict type, that is not a set and it is not a correct method to represent set.
* We can create empty set by using the set function as s=set()
* Using the set function we can pass only one argument in set.
* We cannot pass one argument using set function as s=(10), it will throw error because set is such a function that we can pass only sequence in the form of arguments. We can do that as s=([10,20,30])
* If we create a list like l=[10,20,30.10,30] and want this to be converted to set then we can do that but as the values of list is mutable, in that case the duplicate values will be eliminated keeping only one of them.
* Below is the shell script for the above point:

>>> s={10,20,30}

>>> type(s)

<class 'set'>

>>> s={10,20,30,10,10,30}

>>> s

{10, 20, 30}

>>> s[0]

Traceback (most recent call last):

File "<pyshell#4>", line 1, in <module>

s[0]

TypeError: 'set' object is not subscriptable

>>> s[1:3]

Traceback (most recent call last):

File "<pyshell#5>", line 1, in <module>

s[1:3]

TypeError: 'set' object is not subscriptable

>>> s={}

>>> type(s)

<class 'dict'>

>>> s=set{}

SyntaxError: invalid syntax

>>> s=set()

>>> type(s)

<class 'set'>

>>> s=set(10,20,30)

Traceback (most recent call last):

File "<pyshell#11>", line 1, in <module>

s=set(10,20,30)

TypeError: set expected at most 1 arguments, got 3

>>> s=set([10,20,30])

>>> s

{10, 20, 30}

>>> type(s)

<class 'set'>

>>> s=set("mysirg.com")

>>> s

{'s', 'c', 'y', 'g', 'i', 'r', 'm', '.', 'o'}

>>> #here the data is stored randomly, it can be in sequence as we saw

>>> #in s=set([10,20,30])

>>> l=[10,20,30,10,30]

>>> s=set(l)

>>> s

{10, 20, 30}

>>> s=set("abababa")

>>> s

{'b', 'a'}

>>>

**Set : Part2**

* Add: we can add similar or heterogeneous data in set.
* We will not get error if we add a value which is already existing, instead we will get distinct values as an output.
* There is one function in set which is update. It does the union of two sequences, those sequences can be of any type.
* Discard function deletes the value which is asked for
* Remove function also does the same.
* Difference between discard and remove is that discard doesn’t gives any error if the item passed to delete is not available in sequence, whereas remove method throws an error.
* We cannot combine list, tuple and elements all together in set, it will throw unhashable type error. List cannot be in the form of elements in set
* We can combine two tuple and elements in set and then we can perform discard operation
* S3qOne more function is intersection, it gives the common value as output.
* Clear function will make the set empty.
* Union function does the union of two sets.
* Issubset and issuperset: to understand that we need to understand subset and superset, for example: s1={1,2,3,5,8} and s2={2,3}, here s2 is subset of s1, on contrary s1 is superset of s2.
* Pop function will remove any element from the set and it will be a random element, pop function cannot work in empty set.

>>> l=[10,20,30,10,30]

>>> s=set(l)

>>> s

{10, 20, 30}

>>> s=set("abababa")

>>> s

{'b', 'a'}

>>> s={1,2,3}

>>> s.add(4)

>>> s

{1, 2, 3, 4}

>>> s.add(3.5)

>>> s

{1, 2, 3.5, 3, 4}

>>> s.add("HEllo")

>>> s

{1, 2, 3.5, 3, 4, 'HEllo'}

>>> s.add(3.5)

>>> s

{1, 2, 3.5, 3, 4, 'HEllo'}

>>> s={1,2,3}

>>> l=[1,3,5,7]

>>> s.update(l)

>>> s

{1, 2, 3, 5, 7}

>>> s.update([3,5,9],(5,6,7,8))

>>> s

{1, 2, 3, 5, 6, 7, 8, 9}

>>> s.discard(5)

>>> s

{1, 2, 3, 6, 7, 8, 9}

>>> s.discard(4)

>>> s.remove(4)

Traceback (most recent call last):

File "<pyshell#42>", line 1, in <module>

s.remove(4)

KeyError: 4

>>> s={1,2,[3,4],(5,6)}

Traceback (most recent call last):

File "<pyshell#43>", line 1, in <module>

s={1,2,[3,4],(5,6)}

TypeError: unhashable type: 'list'

>>> s={1,2,(3,4),(5,6)}

>>> s

{(5, 6), 1, 2, (3, 4)}

>>> s.discard((3,4))

>>> s

{(5, 6), 1, 2}

>>> s1={1,2,3,5,8}

>>> s2={2,3,4,6}

>>> s1.intersection(s2)

{2, 3}

>>> 1.clear

SyntaxError: invalid syntax

>>> s1.clear

<built-in method clear of set object at 0x06139AF8>

>>> s1

{1, 2, 3, 5, 8}

>>> s1.clear()

>>> s1

set()

>>> s1={1,2,3,5,8}

>>> s1.union(s2)

{1, 2, 3, 4, 5, 6, 8}

>>> s1={1,2,3,5,8}

SyntaxError: unexpected indent

>>> s2={2,3}

>>> s1={1,2,3,5,8}

>>> s2.issubset(s1)

True

>>> s1.issuperset(s2)

True

>>> s1.pop()

1

>>> s1

{2, 3, 5, 8}

>>> s=set()

>>> s.pop

<built-in method pop of set object at 0x06139C60>

>>> s.pop()

Traceback (most recent call last):

File "<pyshell#67>", line 1, in <module>

s.pop()

KeyError: 'pop from an empty set'

>>> s1=s2.copy()

>>> s1

{2, 3}

>>> s2

{2, 3}

>>> id(s1)

101949232

>>> id(s2)

101948752

>>>

**Assignment 9:**

1. **Write a python script to find the common elements in the set**

'''find the common elements in the set'''

print("Enter comma seperated elements for the first set:")

s1={int(e) for e in input().split(',')}

print("Enter comma seperated elements for the second set:")

s2={int(e) for e in input().split(',')}

print("Common elements are: ")

for e in s1.intersection(s2):

print(e,end=' ')

'''other way'''

s1=set(input("Enter the values of first set: ").split(','))

s2=set(input("Enter the values of second set: ").split(','))

s3=set(s1.intersection(s2))

print(s3,end=' ')

Output:

Enter comma seperated elements for the first set:

1,2,3,4,5

Enter comma seperated elements for the second set:

1,3,5

Common elements are:

1 3 5 Enter the values of first set: 1,2,3,4,5

Enter the values of second set: 1,3,5

{'1', '3', '5'}

>>> type(s3)

<class 'set'>

>>>

1. **Write a python script to print the set of first N prime numbers.**

'''Write a python script to print the set of first N prime numbers'''

n=int(input("Enter the number till which prime number is needed:"))

x=2

s=set() #creating an empty set

while n: #loop statements until n is zero

if all(x%i!=0 for i in range(2,x)): #check x for prime numbers

s.add(x)

n-=1

x+=1

print("Set of prime number is: ",s)

Output:

Enter the number till which prime number is needed:5

Set of prime number is: {2, 3, 5, 7, 11}

>>>

1. **Write a python script to take union of two sets.**

'''union of two sets'''

s1=set(input("Enter the values of first set: ").split(','))

s2=set(input("Enter the values of second set: ").split(','))

print("The union of the sets are: ",(s1.union(s2)),end=' ')

#the above code will give output with commas and single inverted commas

#the below code will give output without commas and single inverted commas

print("Enter comma seperated elements for the firsrt set")

s1={int(e) for e in input().split(',')}

print("Enter comma seperated elements for the second set")

s2={int(e) for e in input().split(',')}

print("union of two sets are: ")

for e in s1.union(s2):

print(e,end=' ')

Output:

Enter the values of first set: 1,2,3,4,5

Enter the values of second set: 67,89,99

The union of the sets are: {'3', '2', '5', '89', '4', '67', '1', '99'} Enter comma seperated elements for the firsrt set

1,2,3,4,5

Enter comma seperated elements for the second set

6,7,8,9,10

union of two sets are:

1 2 3 4 5 6 7 8 9 10

>>>

1. **Write a python script to count the elements of the set.**

'''count the elements of the set'''

print("Enter comma seperated set elements: ")

s1={int(e) for e in input().split(',')}

print("Total elements in the set are",len(s1))

output:

Enter comma seperated set elements:

1, 2, 3 , 4

Total elements in the set are 4

1. **Write a python script to count distinct elements in a list using set.**

'''count distinct elements in a list using set'''

print("Enter comma seperated list elements:")

l=[int(e) for e in input().split(',')]

s=set(l)

print("Total distinct elements in the list",l, "are: ",len(s),"and those elements are",s)

Output:

Enter comma seperated list elements:

1,1,2,2,3,3,4,4

Total distinct elements in the list [1, 1, 2, 2, 3, 3, 4, 4] are: 4 and those elements are {1, 2, 3, 4}

>>>

1. **Write a python script to create a power set of a given set.**

Before jumping into the program let us understand powerset first.

Power set is P(s) of a set s is the set of all subsets of s. For example s={a,b,c} , then P(s)= {{},{a},{b},{c},{a,c},{b,c},{a,b,c}}

If s has n elements in it then P(s) will have 2^n elements.

'''create a power set of a given set'''

print("Enter comma seperated set elements: ")

s1={int(e) for e in input().split(',')}

l1=list(s1)

n=len(s1)

powerset=set()

for i in range(2\*\*n):

t=tuple({l1[j] for j in range(n) if (i&(1<<j))})

powerset.add(t)

print(powerset)

output:

Enter comma seperated set elements:

1,2,3

{(1, 2), (1, 3), (1,), (2,), (3,), (1, 2, 3), (), (2, 3)}

>>>

1. **Write a python script to check whether the given set is subset of another given set or not.**

'''check whether the given set is subset of another given set or not.'''

s1={1,2,3,4,5}

s2={1,2,3}

s3={4,5,6,7}

if s2.issubset(s1):

print("s2 is subset of s1")

elif s3.issubset(s1):

print("s3 is a subset of s1")

Output:

=== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 9 q7.py ===

s2 is subset of s1

>>>

**Dictionary (dict)**

As we have seen in List, tuple and set, we use to store homogenous or heterogeneous collection of values. Dictionary works in association which means we can associate a key to the value.

For example, if we want to know the names of student in a class with their roll number, then in that case we use dictionary, it can store value/s as a pair.

So each element in List, tuple and set is an object, however in dictionary each element is a pair of key-value.

**Few important points on Dictionary:**

1. Key’s value cannot be repeated, hence key must be unique.
2. Repetition of value can be repeated.
3. It is not necessary that key and value should of same type. It can be any type’s combination.
4. Indexing is not possible in Dictionary like in list, set and tuple. As indexing is not possible, slicing operator doesn’t work.
5. The pair can be in any sequence. Cannot guarantee of order of storage of key-values.
6. Dictionary is mutable, which means we can edit, add, delete or can change the value.
7. The above points concludes that its memory can dynamically grow and shrink.

**Creating Dictionary:**

>>> d={}

>>> type(d)

<class 'dict'>

>>> d

{}

>>> d={100:'Rahul',101:'Ajay',102:'Samar'}

>>> d

{100: 'Rahul', 101: 'Ajay', 102: 'Samar'}

>>> d

{100: 'Rahul', 101: 'Ajay', 102: 'Samar'}

>>> d={'A':'Apple','B':'Ball'}

>>> d

{'A': 'Apple', 'B': 'Ball'}

>>> #the above example shows the dict type initialization

>>> #when use dict then the colon process is not followed which we will see next

>>> **d=dict(one='Bhopal’, two='Indore')**

>>> d

{'one': 'Bhopal', 'two': 'Indore'}

>>> d['one']

'Bhopal'

>>> d={1:'A',2:'B',3:'C',4:'D'}

>>> #NOW WE WANT TO ADD A VALUE IN d

>>> d[5]='E'

>>> d

{1: 'A', 2: 'B', 3: 'C', 4: 'D', 5: 'E'}

>>> #as we know we cannot duplicate value but if we do so it will not

>>> #throw error but will edit the value, lets see that

>>> d[1]='a'

>>> d

{1: 'a', 2: 'B', 3: 'C', 4: 'D', 5: 'E'}

**Hence when we use dict constructor we use equal to not colon and while using dict we need to provide the key directly, not as any string. It will be internally treated as string. The keys cannot be expression and neither numbers.**

**How to access elements of dict?**

>>> #now we want access all elements of d

>>> d={1:'A',2:'B',3:'C',4:'D',5:'E'}

>>> for k in d:

print("key",k, "value",d[k])

key 1 value A

key 2 value B

key 3 value C

key 4 value D

key 5 value E

>>>

The memory allocation of key-value in dict is random, hence we cannot guarantee any particular sequence to be followed like set.

We will now understand how to take input from user in dict.

words={}

l=['a','b','c','d']

for x in l:

print("Enter word for letter",x)

words[x]=input()

print(words)

Output:

Enter word for letter a

apple

Enter word for letter b

ball

Enter word for letter c

cat

Enter word for letter d

dog

{'a': 'apple', 'b': 'ball', 'c': 'cat', 'd': 'dog'}

>>>

Now we will see how to remove data.

>>> words

{'a': 'apple', 'b': 'ball', 'c': 'cat', 'd': 'dog'}

>>> del('c')

SyntaxError: can't delete literal

>>> del(words['c'])

'if we want to delete the key value from dictionary then only key will not work, we need to provide the dictionary name and then the key in square bracket'

>>> words

{'a': 'apple', 'b': 'ball', 'd': 'dog'}

>>>

Some useful functions while dealing with dict:

>>> words

{'a': 'apple', 'b': 'ball', 'd': 'dog'}

>>> words.clear()

>>> words

{}

>>> '''clear function removes all the items of dictionary'''

'clear function removes all the items of dictionary'

>>>

>>> d={1:"A",2:'B'}

>>> d1=d.copy()

>>> d

{1: 'A', 2: 'B'}

>>> d1

{1: 'A', 2: 'B'}

>>> d is d1

False

>>> '''False value shows that the id of d and d1 is different'''

>>> '''also it is pointing to two different objects'''

>>> '''This copy is known as shallow copy'''

>>> '''Now suppose we want to use the value of list as key in dictionary'''

>>> l=[1,2,3]

>>> d.fromkeys(l,"ABC")

{1: 'ABC', 2: 'ABC', 3: 'ABC'}

>>> '''This doesn't make any change in d, however we can assign it in d'''

>>> d2=d.fromkeys(l,"ABC")

>>> d2

{1: 'ABC', 2: 'ABC', 3: 'ABC'}

>>> d

{1: 'A', 2: 'B'}

>>> d.get(2)

'B'

>>> d[2]

'B'

>>> '''both give the same value but there is difference, we cannot put invalid key in d[] as it will give key error, however the get method will not throw any error'''

>>> d.get(3)

>>> x=d.get(3)

>>> x

>>> type(x)

<class 'NoneType'>

>>> d2

{1: 'ABC', 2: 'ABC', 3: 'ABC'}

>>> y=d2.pop(2)

>>> y

'ABC'

>>> d2

{1: 'ABC', 3: 'ABC'}

>>> '''pop function returns the value of dict but not the key'''

>>> d

{1: 'A', 2: 'B'}

>>> d1

{1: 'A', 2: 'B'}

>>> d2

{1: 'ABC', 3: 'ABC'}

>>> d[3]="C"

>>> d[4]="D"

>>> d

{1: 'A', 2: 'B', 3: 'C', 4: 'D'}

>>> t=d.popitem()

>>> d

{1: 'A', 2: 'B', 3: 'C'}

>>> type(t)

<class 'tuple'>

>>> t

(4, 'D')

>>> t[0]

4

>>> t[1]

'D'

>>> help(dict)

Help on class dict in module builtins:

class dict(object)

| dict() -> new empty dictionary

| dict(mapping) -> new dictionary initialized from a mapping object's

| (key, value) pairs

| dict(iterable) -> new dictionary initialized as if via:

| d = {}

| for k, v in iterable:

| d[k] = v

| dict(\*\*kwargs) -> new dictionary initialized with the name=value pairs

| in the keyword argument list. For example: dict(one=1, two=2)

|

| Methods defined here:

|

| \_\_contains\_\_(self, key, /)

| True if the dictionary has the specified key, else False.

|

| \_\_delitem\_\_(self, key, /)

| Delete self[key].

|

| \_\_eq\_\_(self, value, /)

| Return self==value.

|

| \_\_ge\_\_(self, value, /)

| Return self>=value.

|

| \_\_getattribute\_\_(self, name, /)

| Return getattr(self, name).

|

| \_\_getitem\_\_(...)

| x.\_\_getitem\_\_(y) <==> x[y]

|

| \_\_gt\_\_(self, value, /)

| Return self>value.

|

| \_\_init\_\_(self, /, \*args, \*\*kwargs)

| Initialize self. See help(type(self)) for accurate signature.

|

| \_\_iter\_\_(self, /)

| Implement iter(self).

|

| \_\_le\_\_(self, value, /)

| Return self<=value.

|

| \_\_len\_\_(self, /)

| Return len(self).

|

| \_\_lt\_\_(self, value, /)

| Return self<value.

|

| \_\_ne\_\_(self, value, /)

| Return self!=value.

|

| \_\_repr\_\_(self, /)

| Return repr(self).

|

| \_\_setitem\_\_(self, key, value, /)

| Set self[key] to value.

|

| \_\_sizeof\_\_(...)

| D.\_\_sizeof\_\_() -> size of D in memory, in bytes

|

| clear(...)

| D.clear() -> None. Remove all items from D.

|

| copy(...)

| D.copy() -> a shallow copy of D

|

| get(self, key, default=None, /)

| Return the value for key if key is in the dictionary, else default.

|

| items(...)

| D.items() -> a set-like object providing a view on D's items

|

| keys(...)

| D.keys() -> a set-like object providing a view on D's keys

|

| pop(...)

| D.pop(k[,d]) -> v, remove specified key and return the corresponding value.

| If key is not found, d is returned if given, otherwise KeyError is raised

|

| popitem(...)

| D.popitem() -> (k, v), remove and return some (key, value) pair as a

| 2-tuple; but raise KeyError if D is empty.

|

| setdefault(self, key, default=None, /)

| Insert key with a value of default if key is not in the dictionary.

|

| Return the value for key if key is in the dictionary, else default.

|

| update(...)

| D.update([E, ]\*\*F) -> None. Update D from dict/iterable E and F.

| If E is present and has a .keys() method, then does: for k in E: D[k] = E[k]

| If E is present and lacks a .keys() method, then does: for k, v in E: D[k] = v

| In either case, this is followed by: for k in F: D[k] = F[k]

|

| values(...)

| D.values() -> an object providing a view on D's values

|

| ----------------------------------------------------------------------

| Class methods defined here:

|

| fromkeys(iterable, value=None, /) from builtins.type

| Create a new dictionary with keys from iterable and values set to value.

|

| ----------------------------------------------------------------------

| Static methods defined here:

|

| \_\_new\_\_(\*args, \*\*kwargs) from builtins.type

| Create and return a new object. See help(type) for accurate signature.

|

| ----------------------------------------------------------------------

| Data and other attributes defined here:

|

| \_\_hash\_\_ = None

>>>

**Assignment 10:**

1. **Write a python script to print dict items (KEY,VALUE), each in one line**

'''python script to print dict items (KEY,VALUE), each in one line'''

d={100:"Arav",101:"Arnav",103:"Bishal"}

for k in d:

print("key",k,"Value",d[k])

Output:

== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 10 q1.py ==

key 100 Value Arav

key 101 Value Arnav

key 103 Value Bishal

>>>

1. **Write a python script to create dictionary in which each item is a pair of roll number(as key) and Student Name (as Value)**

'''python script to create dictionary in which each item is a pair of roll number(as key) and Student Name (as Value)'''

d={}

ch='y'

while ch!='n':

rno=int(input("Enter roll number:"))

sname=input("Enter student's name:")

ch=input("Do you want to continue? (y/n)")

d[rno]=sname

print('list of students')

print(d)

Output:

== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 10 q2.py ==

Enter roll number:1

Enter student's name:anulekha

Do you want to continue? (y/n)y

Enter roll number:2

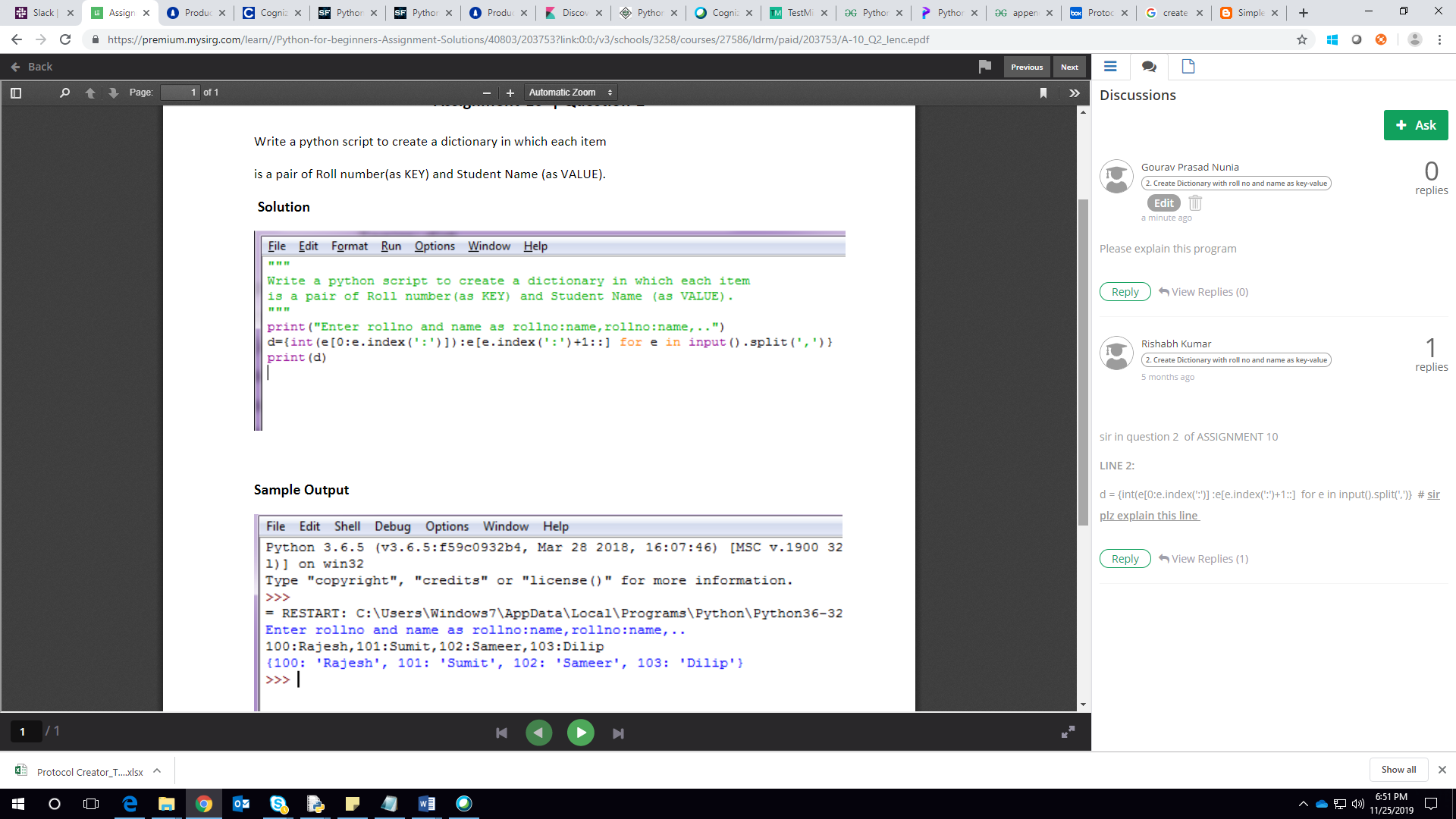
Enter student's name:Bishal

Do you want to continue? (y/n)n

list of students

{1: 'anulekha', 2: 'Bishal'}

>>>



1. **Write a python script to sort a dict according to key.**

Before understanding this program we need to understand that there is way to sort dictionary as the memory allocation doesn’t have any order.

**Lambda Function**

This are the functions which are defined without names, that’s why this are known as anonymous functions.

In python we define functions in following syntax:

def functionname():

function body

but in case of Lambda function, it doesn’t carry any def keyword, the syntax is as follows:

lamda arguments:expression

so the general way of writing function in python is

**def add(x,y):**

**return x+y**

However when we use lambda, we use it as

**lambda x,y:x+y**

**when lambda can be used?**

1. It can be used for short period of time.
2. It can be used as an arguments in the higher order function
3. Higher order functions are the functions which can use another functions as arguments.
4. So, in that case we can use lambda functions as the arguments in the higher order functions.
5. We can see the advantage of lambda function when it is used with the built in function such as map, reduce and filter.

**Map:**

This function is used to apply a function to all the elements of the sequence.

Syntax:

Map(function,sequence)

For example if we want to find the square of each elements in the list, in that case we need to apply the function to each element of the sequence, in this scenario map is used.

we will see use of map function with lambda and without lamda also.

We can use map function with multiple lists also.

**Filter and Reduced function:**

Filter function will filters the elements of the itterables based on some function. This is basically to filter out the unwanted element.

Syntax;

filter(function, iterables)

when the

function will ret-

urn true for any

value of iterables, that

value will be returned by the

filter function.

**Reduce Function:**

Reduce function will reduce the iterables into a single element, hence the output from the reduce function is a single element. Suppose we want to find the sum or product of all element in the list then we can use the reduce function:

Syntax:

Reduce(function, iterables)

Python 3.7.3 (v3.7.3:ef4ec6ed12, Mar 25 2019, 21:26:53) [MSC v.1916 32 bit (Intel)] on win32

Type "help", "copyright", "credits" or "license()" for more information.

>>> list1=[7,5,6,3,4,2,3,1]

>>> tuple1=((3,8),(2,9),(1,10),(4,7))

>>> d1={3:"e",2:"a",1:"c",7:"b",5:"d"}

>>> sorted(list1)

[1, 2, 3, 3, 4, 5, 6, 7]

>>> sorted(list1,reverse=True)

[7, 6, 5, 4, 3, 3, 2, 1]

>>> sorted(tuple1)

[(1, 10), (2, 9), (3, 8), (4, 7)]

>>> sorted(tuple1,reverse=True)

[(4, 7), (3, 8), (2, 9), (1, 10)]

>>> sorted(d1)

[1, 2, 3, 5, 7]

>>> sorted(d1.values())

['a', 'b', 'c', 'd', 'e']

>>> sorted(d1.items())

[(1, 'c'), (2, 'a'), (3, 'e'), (5, 'd'), (7, 'b')]

>>> sorted(d1.items(),key=lambda x:x[1])

[(2, 'a'), (7, 'b'), (1, 'c'), (5, 'd'), (3, 'e')]

>>> help(sorted)

Help on built-in function sorted in module builtins:

sorted(iterable, /, \*, key=None, reverse=False)

Return a new list containing all items from the iterable in ascending order.

A custom key function can be supplied to customize the sort order, and the

reverse flag can be set to request the result in descending order.

>>> a=[1,2,3,4]

>>> def square(x):

return x\*x

>>> map(square,a)

<map object at 0x014F9910>

>>> list(map(square,a))

[1, 4, 9, 16]

>>> list(map(lambda x:x\*x,a))

[1, 4, 9, 16]

>>> #map with multiple list

>>> a

[1, 2, 3, 4]

>>> b=[1,1,1,1]

>>> b

[1, 1, 1, 1]

>>> #we can use map to add 2 lists but the length and data type should be same

>>> tuple(map(lambda x,y:x+y,a,b))

(2, 3, 4, 5)

>>> #we can use list and tuple also, we can get the output in the same way

>>> #filter function

>>> for x in range(1,11):

if(x%2==0):

print(x)

2

4

6

8

10

>>> #the same thing can be acheived by using filter function:

>>> filter(lambda x:x%2==0,range(1,11))

<filter object at 0x014E6F10>

>>> list(filter(lambda x:x%2==0,range(1,11)))

[2, 4, 6, 8, 10]

>>> #filter function can be applied only on one iterable:

>>> #we will get filter error if we pass two list like as follows:

>>> a=[1,2,3,4]

>>> b=[2,3,4,5]

>>> filter(lambda x:x>2,a,b))

SyntaxError: invalid syntax

>>> filter(lambda x:x>2,a,b)

Traceback (most recent call last):

File "<pyshell#39>", line 1, in <module>

filter(lambda x:x>2,a,b)

TypeError: filter expected 2 arguments, got 3

>>> #reducce function

>>> import functools

>>> num=[1,2,3,4]

>>> functools.reduce(lambda x,y:x+y,num)

10

>>> #reduce function is kept in functool module, hence we need to import it first.

>>> #reduce function can be used only in one iterable.

>>> #hence both filter and reduce function can be applied only on 1 iterable

>>> tuple1=[(1,"a"),(4,"s"),(3,"z"),(2,"r"))]

SyntaxError: invalid syntax

>>> tuple1=[(1,"a"),(4,"s"),(3,"z"),(2,"r")]

>>> def secondvalue(element):

return element[1]

>>> sorted(tuple1,key=secondvalue)

[(1, 'a'), (2, 'r'), (4, 's'), (3, 'z')]

>>>

**Write a python script to sort a dict according to the key.**

**More on Input: Part 1**

We are about to learn more input instruction in this topic.

1. Reading Single value

As we know that, if we want to take input from user then we use the following method:

x=input("Enter a number:")

The above method will always return the value as string by default as the return type of input function is string as shown below in python shell.

>>> x=input("Enter a number:")

Enter a number:123

>>> type(x)

<class 'str'>

>>>

Now suppose if we want the function to return integer value then we will do the type conversion and will write the code as:

>>> x=int(input("Enter a number:"))

Enter a number:123

>>> type(x)

<class 'int'>

>>>

What if we enter a float value, any sequence type value or complex number, in this case it will throw error because the int function will not convert the value as shown below:

>>> x=int(input("Enter a number:"))

Enter a number:3.5

Traceback (most recent call last):

File "<pyshell#4>", line 1, in <module>

x=int(input("Enter a number:"))

ValueError: invalid literal for int() with base 10: '3.5'

>>> x=int(input("Enter a number:"))

Enter a number:3+4j

Traceback (most recent call last):

File "<pyshell#5>", line 1, in <module>

x=int(input("Enter a number:"))

ValueError: invalid literal for int() with base 10: '3+4j'

>>> x=int(input("Enter a number:"))

Enter a number:[1,2,3]

Traceback (most recent call last):

File "<pyshell#6>", line 1, in <module>

x=int(input("Enter a number:"))

ValueError: invalid literal for int() with base 10: '[1,2,3]'

>>>

Now the question is what if the user wants to enter any kind of value and the program should return the exact type as per the values entered. This can be achieved by using the function: eval

eval function doesn’t do the type conversion but do the evaluation of the literals passed internally and return the exact type. We can enter any float value, any sequence type value or complex number, it will return the exact type of the value as shown in following python shell.

>>> x=eval(input("Enter a number:"))

Enter a number:3.5

>>> x=eval(input("Enter a number:"))

Enter a number:3+4j

>>> x

(3+4j)

>>> type(x)

<class 'complex'>

>>> x=eval(input("Enter a number:"))

Enter a number:[1,2,3]

>>> x

[1, 2, 3]

>>> type(x)

<class 'list'>

>>> x=eval(input("Enter a number:"))

Enter a number:{1,2,3,4,5,1,2,3,4,5}

>>> x

{1, 2, 3, 4, 5}

>>> type(x)

<class 'set'>

>>> #set cannot keep repeated value hence the eval function does that job too to eliminate the value

>>> x=eval(input("Enter a number:"))

Enter a number:{1:'a',2:'b',3:'c'}

>>> x

{1: 'a', 2: 'b', 3: 'c'}

>>> type(x)

<class 'dict'>

>>>

**P.S: The eval function is only for taking a single value from the user. We will see for multiple values in next chapter.**

**More on input: part2**

We are about to learn more input instruction in this topic.

1. Reading Multiple value

Suppose we want to enter two values then we will write the code as follow:

>>> x,y=input("Enter first number:"),input("Enter second number:")

Enter first number:25

Enter second number:35

>>> x

'25'

>>> y

'35'

>>>

The above method will definitely return the value as string. So one of the method to enter the multiple value is writing the input function multiple times. It is a tedious process if we want to enter 10 values or more than 2 values.

The problem can be overcome in a way if we use the split function.

>>> x,y=input("Enter two number:").split()

Enter two number:one two

>>> x

'one'

>>> y

'two'

>>> '''the input function returns string type and the split function will split the string on the basis of space as no argument is passed in it'''

The entire “one two” is a string which is split on the basis of space.

It is not mandatory that the string needs to be split on the basis of space only, we can split on the basis of any symbol.

>>> x,y=input("Enter two number:").split(',')

Enter two number:one,two

>>> x

'one'

>>> y

'two'

>>>

The entire “one, two” is a string which is split on the basis of comma

The code syntax input().split() will return the **list of strings,** lets prove the same with the code:

>>> x=input("Enter few words:").split()

Enter few words:one two three four

>>> x

['one', 'two', 'three', 'four']

>>> d=input("Enter date:").split('/')

Enter date:25/12/2019

>>> d

['25', '12', '2019']

Hence we can see that input().split() will return the **list of strings.**

>>> a,b,c=input("Enter date:").split('/')

Enter date:25/12/2019

>>> a

'25'

>>> b

'12'

>>> c

'2019'

>>>

In the above code it is returning as string type, now if we want to convert it into int type, then the code will be different.

We cannot write it as:

>>> a,b,c=int(input("Enter date:").split('/'))

Enter date:25/12/2019

Traceback (most recent call last):

File "<pyshell#44>", line 1, in <module>

a,b,c=int(input("Enter date:").split('/'))

TypeError: int() argument must be a string, a bytes-like object or a number, not 'list'

>>>

It throws error because the return type of input().split() is list of strings and the int function is not able to convert the string type into integer.

The correct of doing so is as:

>>> a,b,c=[int(x) for x in input("Enter date:").split('/')]

Enter date:25/12/2019

>>> a

25

>>> b

12

>>> c

2019

>>> a,b,c

(25, 12, 2019)

>>>

For loop will run for each element of the list[].

**More on input: part3**

We are about to learn more input instruction in this topic.

1. Reading Multiple value of different type

>>> a,b,c=[eval(x) for x in input("Enter three values:").split(',')]

Enter three values:12,'abc',3.5

>>> a

12

>>> b

'abc'

>>> c

3.5

>>> type(a)

<class 'int'>

>>> type(b)

<class 'str'>

>>> type(c)

<class 'float'>

>>>

Now suppose we want the value entered to be returned as list or tuple or set or dict. All has different code to achieve that as follow:

>>> l=[eval(x) for x in input("Enter three values:").split(',')]

Enter three values:11,22,33

>>> l

[11, 22, 33]

>>> type(l)

<class 'list'>

Now suppose we want tuple kind of value then if we put the code in parenthesis then it will return generator not tuple type as shown below:

>>> t=(eval(x) for x in input("Enter three values:").split(','))

Enter three values:1,2,3

>>> t

<generator object <genexpr> at 0x034B07B0>

>>>

So if we are adamant to return the type as tuple then we can write the code as:

>>> t=tuple([eval(x) for x in input("Enter three values:").split(',')])

Enter three values:10,20,30

>>> t

(10, 20, 30)

>>> type

<class 'tuple'>

>>>

Similarly we cannot put the curly bracket to return set type, it won’t as it will throw error:

>>> t={[eval(x) for x in input("Enter three values:").split(',')]}

Enter three values:11,22,33

Traceback (most recent call last):

File "<pyshell#21>", line 1, in <module>

t={[eval(x) for x in input("Enter three values:").split(',')]}

TypeError: unhashable type: 'list'

>>>

We can use the set constructor as:

>>> t=set([eval(x) for x in input("Enter three values:").split(',')])

Enter three values:11,22,33

>>> t

{33, 11, 22}

>>> type(t)

<class 'set'>

>>>

>>> t=set([eval(x) for x in input("Enter three values:").split(',')])

Enter three values:11,22,33,44,11,22,33

>>> t

{33, 11, 44, 22}

>>> type(t)

<class 'set'>

>>>

For returning dict type, it is little tricky to do that, there are several ways to do so:

>>> d=dict(input().split('-') for \_ in range(3))

1-A

2-B

3-C

>>> d

{'1': 'A', '2': 'B', '3': 'C'}

>>> type(d)

<class 'dict'>

>>>

We can see that the pair key and value is returning as string.

Now suppose we want that the key is there and value needs to be entered by the user.

We can do that without using the constructor dict.

>>> d={x:input() for x in range(1,5)}

Ajay

Bishal

Chandan

Delta

>>> d

{1: 'Ajay', 2: 'Bishal', 3: 'Chandan', 4: 'Delta'}

>>> type(d)

<class 'dict'>

Suppose we want both the values to be entered by user:

>>> d1={input("Key:"):input("Value:") for x in range(3)}

Value:Amit

Key:100

Value:Bishal

Key:101

Value:Ajay

Key:102

>>> d1

{'100': 'Amit', '101': 'Bishal', '102': 'Ajay'}

"One important thing to note that if two input function is passed then the input function after colon will be executed first. as per the above results the pair has been returned as string type, now we want the key's input to be returned as integer only, then the following code works:"

>>> d1={int(input("Key:")):input("Value:") for x in range(3)}

Value:Amit

Key:100

Value:Bishal

Key:101

Value:Chandan

Key:103

>>> d1

{100: 'Amit', 101: 'Bishal', 103: 'Chandan'}

>>>

**Functions in python.**

Function is a block of code which we can reuse in our program.

**How to create function in python?**

Def function\_name():

Statement1

Statement2

----------------

We need to take care of colon which is put after the function name, also indentation needs to be maintained.

**How to call or use the function in python?**

The code written in function will be executed once called. If it is not called then it will not run those code.

We can call the function by simply typing the function name.

Sample program in which it is shown how function is defined and after that it is called twice:

def add():

a=int(input("Enter first number:"))

b=int(input("Enter second number:"))

c=a+b

print("Sum is:",c)

add()

add()

Output:

Enter first number:10

Enter second number:20

Sum is: 30

Enter first number:100000025

Enter second number:25000011

Sum is: 125000036

>>>

We can modify the above program in a certain way like:

def add():

a=int(input("Enter first number:"))

b=int(input("Enter second number:"))

c=a+b

print("Sum is:",c)

print("This is First line")

add()

print("This is the line after first call to the function add:")

add()

Output:

This is First line

Enter first number:98

Enter second number:89

Sum is: 187

This is the line after first call to the function add:

Enter first number:597

Enter second number:795

Sum is: 1392

>>>

There are 2 types of function:

1. Pre-defined functions: which are already inbuilt like print, eval, dict, sorted etc.
2. User-defined function: which are made or customized by the user like add in above example.

**Ways to define function:**

1. Takes Nothing, Returns Nothing.
2. Takes Something, Returns Nothing.
3. Takes Nothing, Returns Something.
4. Takes Something, Returns Something.

**Takes Nothing, Returns Nothing**

Takes nothing can be denoted by empty parenthesis () and returns nothing can be understood when no return keyword is used in the function.

**Takes Something, Returns Nothing**

'''Takes Something returns nothing function'''

def add(a,b):

c=a+b

print("Sum is:",c)

add(10,20)

'''while defining the function the variables which we pass is known as formal

Argument and the value we pass while calling the function is known as actual argument'''

'''actual arguments should be same with formal arguments'''

Output:

Sum is: 30

**Takes Nothing, Returns Something**

'''Takes Nothing, Returns Something'''

def add():

a=int(input("Enter first number:"))

b=int(input("Enter second number:"))

c=a+b

return c

x=add()

print("x=",x)

output:

Enter first number:12

Enter second number:13

x= 25

>>>

It is not mandatory that return will return single value but we can return multiple value, list, tuple, set.

**Takes Something, Returns Something**

'''Takes Something, Returns Something'''

def add(a,b):

s=a+b

return s

x=add(10,20)

print("Sum is: ",x)

Output:

Sum is: 30

>>>

**Functions: Part 3**

'''None Type'''

def f1():

print("Hello")

x=f1()

print(x,"-",type(x))

'''in c,c++ the concept was known as void but here it is a None type, so basically when the function doesn't

return anything in python it returns None type'''

it means there is nothing in x which indicates it as None type.

Output:

Hello

None - <class 'NoneType'>

>>>

**Default Argument:**

'''Default Argument'''

def sum(a,b):

s=a+b

print("sum is:",s)

sum(10,20)

sum(10,20,30)

#First line will get executed but not the second one as it will throw error

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/40 Functions Default Argument 1.py

sum is: 30

Traceback (most recent call last):

File "C:/Users/654268/Desktop/Python Programming/40 Functions Default Argument 1.py", line 6, in <module>

sum(10,20,30)

TypeError: sum() takes 2 positional arguments but 3 were given

>>>

**'''Default Argument'''**

def sum(a,b):

s=a+b

print("sum is:",s)

sum(10,20,30)

sum(10,20)

#First line itself will give error because more arguments has been passed

RESTART: C:/Users/654268/Desktop/Python Programming/40 Functions Default Argument 2.py

Traceback (most recent call last):

File "C:/Users/654268/Desktop/Python Programming/40 Functions Default Argument 2.py", line 5, in <module>

sum(10,20,30)

TypeError: sum() takes 2 positional arguments but 3 were given

>>>

**'''Default Argument'''**

def sum(a,b,c):

s=a+b

print("sum is:",s)

sum(10,20)

sum(10,20,30)

#First line itself will give error because more arguments has been passed.

Output:

Traceback (most recent call last):

File "C:/Users/654268/Desktop/Python Programming/40 Functions Default Argument 2.py", line 5, in <module>

sum(10,20)

TypeError: sum() missing 1 required positional argument: 'c'

Suppose we want the lines to be executed then we can use the concept of default argument.

When we define formal argument we can assign a value to it as shown in the below program:

'''Default Argument'''

def sum(a,b,c=0):

s=a+b+c

print("sum is:",s)

sum(10,20)

sum(10,20,30)

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/40 Functions Default Argument 3.py

sum is: 30

sum is: 60

>>>

'''Default Argument'''

def sum(a=0,b=0,c=0):

s=a+b+c

print("sum is:",s)

sum()

sum(10)

sum(10,20)

sum(10,20,30)

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/40 Functions Default Argument 4.py

sum is: 0

sum is: 10

sum is: 30

sum is: 60

if we are passing a default value in formal argument then we will have to set all the default values.

Once a default argument is set for a formal argument then after that whatever arguments is passed, default value needs to be set or assigned.

Actual arguments can also be known as **Positional arguments** because the order in which the arguments are placed gets assigned to the arguments passed while defining the function.

**Keyword Argument:**

def f1(a,b):

print("a=",a,"b=",b)

f1(10,20) #Positional argument

f1(b=20,a=20) #keyword argument where order doesn't matter like positional argument

Output:

a= 10 b= 20

a= 20 b= 20

>>>

Hence we can say that by using keyword argument we can assign value to any formal argument by mentioning the variable name.

We can write the program by mixing keyword arguments and positional arguments but that has a certain way to do which will see below:

def f1(a,b):

print("a=",a,"b=",b)

f1(10,20)

f1(10,b=20) #combination of keyword and positional argument

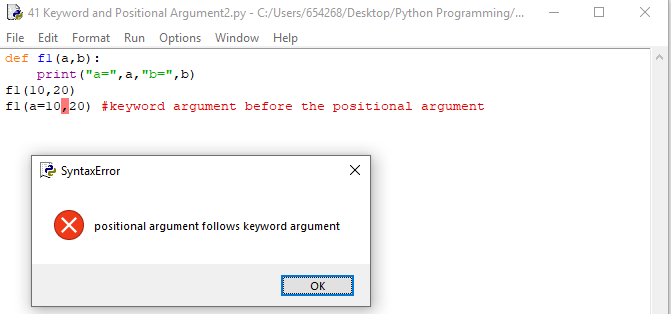
Output:

a= 10 b= 20

a= 10 b= 20

>>>

Here we can note a point that the keyword argument is placed after positional argument and the program gives output without any error. What if we put the keyword argument before the positional argument! Lets see that below;



So, if we try to execute that it will throw an error as: positional argument follows keyword argument. Hence it is mandatory that if an argument is made keyword then all the arguments after that should be keyword argument or we can say that the positional argument cannot come after keyword argument.

Now let us understand what will happen if we pass the values to the same formal argument:

def f1(a,b):

print("a=",a,"b=",b)

f1(10,a=10)

Output:

Traceback (most recent call last):

File "C:/Users/654268/Desktop/Python Programming/41 Keyword and Positional Argument3.py", line 3, in <module>

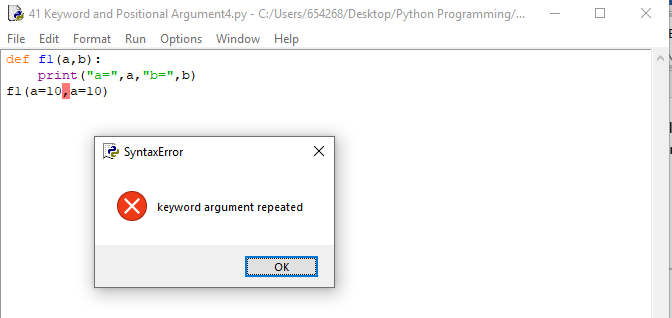
f1(10,a=10)

TypeError: f1() got multiple values for argument 'a'

>>>

So it will throw multiple value argument error.

Let us modify the code and mention the same variable name as a=10 and a=10 again



**Variable Length Argument:**

This come into existence when we don’t know what may be the count of inputs.

Suppose we are finding average of 3 numbers then we pass 3 formal arguments but what if we want to decide how many values needs to be put to find an average, hence the concept of variable length argument comes into play. Let us see how we can code that.

'''Variable length argument'''

def avg(\*n):

s=0

for x in n:

s=s+x

return s/len(n)

x=avg(10,20,30,40,50,60)

print("Average is:",x)

Output:

Average is: 35.0

>>>

This is very useful if we want to pass as many number of actual arguments without using default arguments.

\*n type is tuple, \* doesn’t represent pointer as in C/C++. It takes the actual argument as tuple as proved in the following code:

def avg2(\*n):

print(n,type(n))

avg2(10,20,30)

output:

(10, 20, 30) <class 'tuple'>

>>>

What if we don’t pass any argument, so there will be no elements in the tuple and the operation will be like 0/0 which will basically give error, lets us see that:

def avg(\*n):

s=0

for x in n:

s=s+x

return s/len(n)

x=avg()

print("Average is:",x)

output:

Traceback (most recent call last):

File "C:/Users/654268/Desktop/Python Programming/42 Variable length Argument.py", line 19, in <module>

x=avg()

File "C:/Users/654268/Desktop/Python Programming/42 Variable length Argument.py", line 18, in avg

return s/len(n)

ZeroDivisionError: division by zero

>>>

This doesn’t mean that there is any mistake in code, the mathematical operation 0/0 is not possible, and hence it is throwing ZeroDivisionError.

It can be overcome by applying a condition that if there is no elements then it should show the output accordingly.

def avg(\*n):

s=0

for x in n:

s=s+x

if len(n)!=0:

return s/len(n)

else:

return "No Elements"

x=avg()

print("Average is:",x)

Output:

Average is: No Elements

>>>

We can also combine fixed variable and variable length arguments which is also useful many times, suppose we want to calculate the total points of a player then we can use this combo as:

def f1(playername,\*points):

print(playername,end=' ')

s=0

for x in points:

s=s+x

print("Total points is:",s)

f1("Ajay",10,20,30,56)

f1("Amit",11,13,57,15)

f1("Anurag")

Output:

Ajay Total points is: 116

Amit Total points is: 96

Anurag Total points is: 0

>>>

def f1(playername,\*points):

print(playername,end=' ')

s=0

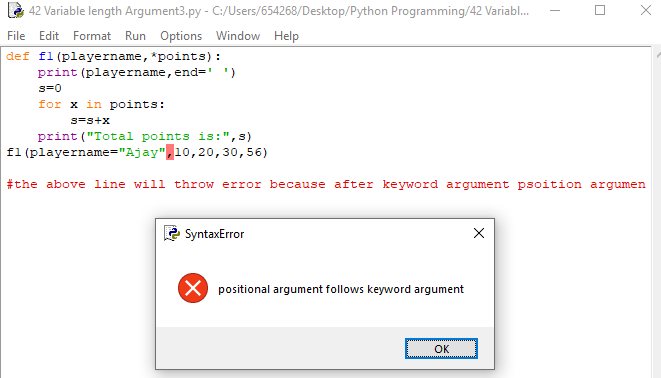
for x in points:

s=s+x

print("Total points is:",s)

f1(playername="Ajay",10,20,30,56)

#the above line will throw error because after keyword argument position argument is invalid.



The above error can be overcome if we change the order of formal arguments while defining.

def f1(\*points,playername):

print(playername,end=' ')

s=0

for x in points:

s=s+x

print("Total points is:",s)

f1("Ajay",10,20,30,56)

However the above code will throw error because it will take all the arguments under points, it won’t be able to understand what to take under playername and what to take under points, by default as the point comes first so it will take all the argument under points.

Output:

Traceback (most recent call last):

File "C:/Users/654268/Desktop/Python Programming/42 Variable length Argument4.py", line 7, in <module>

f1("Ajay",10,20,30,56)

TypeError: f1() missing 1 required keyword-only argument: 'playername'

>>>

**If we use variable length argument and normal argument then it is mandatory that we use keyword argument while calling the function that also for non-variable length argument.**

def f1(\*points,playername):

print(playername,end=' ')

s=0

for x in points:

s=s+x

print("Total points is:",s)

f1(10,20,30,56,playername="Ajay")

Output:

Ajay Total points is: 116

>>>

**Variable length keyword arguments:**

Now suppose we want to store the values of a particular person but with different parameters. So earlier we use to use the concept of tuple but that cannot be applied here because we have to maintain the keyword arguments also which is not possible in tuple. So here we need to appl the concept of dictionary which is represented by \*\*

def f1(\*\*k):

print("Person Information")

for key,value in k.items():

print(key,"-",value)

f1(name="Sameer",age=22)

f1(name="Rahukl",marks=87,age=23)

f1(name="Ajay",empid=125,salry=35000.0)

**output:**

Person Information

name - Sameer

age - 22

Person Information

name - Rahukl

marks - 87

age - 23

Person Information

name - Ajay

empid - 125

salry - 35000.0

>>>

**Assignment 11:**

1. **write a python function to calculate LCM of two numbers.(Takes something, Return Something)**

'''write a python function to calculate LCM of two numbers.

(Takes something, Return Something)'''

def LCM(a,b):

L=max(a,b)

while L<=a\*b:

if L%a==0 and L%b==0:

return L

L+=1

x=LCM(4,6)

print("LCM is:",x)

Output:

LCM is: 12

>>>

1. **write a python function to print reverse of binary representation of a given number (Takes something, returns nothing)**

'''python function to print reverse of binary representation of a

given number(Takes something, returns nothing)'''

def rbin(n):

while n>0:

print(n%2,end='')

n=n//2

OutPut:

>>> rbin(13)

1011

1. **write a python function to print first n prime numbers.(Takes something returns nothing)**

def prime(n):

x=2

while n:

for i in range(2,x):

if x%i==0:

break

else:

print(x,end=' ')

n-=1

x+=1

Output:

>>> prime(5)

2 3 5 7 11

>>>

1. **write a python function to count words in a given string.(Takes something returns something)**

'''python function to count words in a given string.

(Takes something returns something)'''

def countWords(string):

total=1

for i in range(len(string)):

if(string[i]==' ' or string[i]=='\n' or string[i]=='\t'):

total=total+1

return total

sentence=input('please enter your own string:')

leng=countWords(sentence)

print("Total number of words in this string=",leng)

Output:

== RESTART: C:/Users/654268/Desktop/Python Programming/Assignment 11 q4.py ==

please enter your own string:gourav prasad nunia

Total number of words in this string= 3

>>>

1. **write a python function to calculate the sum of all odd numbers and even numbers from a given list of int values.(Takes something Returns something)**

**'''python function to calculate the sum of all odd numbers and even numbers**

**from a given list of int values.(Takes something Returns something)'''**

def sum(l):

s1,s2=0,0

for e in l:

if e%2==0:

s1+=e

else:

s2+=e

return {'sum of even':s1,'sum of odd':s2}

Output:

>>> d=sum([1,2,3,4,5,6,7,8])

>>> d

{'sum of even': 20, 'sum of odd': 16}

>>>

1. **write a python function to find all possible combinations one can make from n items of a given set when two elements are selected at a time.**
2. **Write a python function to find Nth term of Fibonacci series (Takes something, Return Something)**

def fibonacci(n):

if n==1:

return 1

elif n==2:

return 1

elif n>2:

return fibonacci(n-1)+fibonacci(n-2)

for n in range(1,101):

print(n,":",fibonacci(n))

Output:

1 : 1

2 : 1

3 : 2

4 : 3

5 : 5

6 : 8

7 : 13

8 : 21

9 : 34

>>>

The above program will slow down and will increase the PC utilization for the program as it is consistently recurring the value each time the function is running. Hence we need a robust program so that if a higher range of value is given then it should show the result promptly. This can be achieved if we store the previous value as cache which can be done by putting the value in dictionary as cache. The following will explain it.

fibonacci\_cache={}

def fibonacci(n):

if n in fibonacci\_cache:

return fibonacci\_cache[n]

if n==1:

value=1

elif n==2:

value=1

elif n>2:

value=fibonacci(n-1)+fibonacci(n-2)

fibonacci\_cache[n]=value

return value

for n in range(1,101):

print(n,":",fibonacci(n))

Output:

RESTART: C:\Users\654268\Desktop\Python Programming\Assignment 11 q7 robust program.py

1 : 1

2 : 1

3 : 2

4 : 3

5 : 5

6 : 8

7 : 13

8 : 21

9 : 34

10 : 55

11 : 89

12 : 144

13 : 233

14 : 377

15 : 610

16 : 987

17 : 1597

18 : 2584

19 : 4181

20 : 6765

21 : 10946

22 : 17711

23 : 28657

24 : 46368

25 : 75025

26 : 121393

27 : 196418

28 : 317811

29 : 514229

30 : 832040

31 : 1346269

32 : 2178309

33 : 3524578

34 : 5702887

35 : 9227465

36 : 14930352

37 : 24157817

38 : 39088169

39 : 63245986

40 : 102334155

41 : 165580141

42 : 267914296

43 : 433494437

44 : 701408733

45 : 1134903170

46 : 1836311903

47 : 2971215073

48 : 4807526976

49 : 7778742049

1. **Write a python function to find the next prime number of a given number(Takes Something, Return Something).**

def nextPrime(n):

while True:

n+=1

for i in range(2,n):

if n%i==0:

break

else:

return n

Output:

>>> nextPrime(13)

17

>>>

**Recursion:**

A function calling itself is known as recursion.

def f1 ():

------------------

------------------

f1()

--------

--------

* Solution of the problem is defined with the help of result of simpler version of the same problem then the solution is recursive.
* In simple words, we can say that the complexity of the recursive function for which the code is designed will be less complex or will will be used to solve less complex problem than the actual function.
* At certain point recursion needs to be stopped else it will keep on calling the function repeatedly, so a point where the recursion is stopped by applying condition is known as base case.

def sum(n):

if (n==1): #base case

return 1

else

return n+sum(n-1) #recursive case

* The best approach to use recursion is that we need to first define function and then we need to know the recursive case and then we need to figure out the base case where the function needs to be stopped.

**Lambda Expression/ Anonymous Function**

* Anonymous function means a function without name.
* When we make function such way, it is known as Lambda expression.

s=lambda a,b:a+b

r=s(10,20)

print("sum is",r)

####second way####

r=(lambda a,b:a+b)(10,20)

print("sum is",r)

##Maximum of two numbers##

t=(lambda a,b:max(a,b))(10,20)

print("greater number is",t)

##Other way####

print("Enter two numbers")

r=(lambda a,b:a if a>b else b)(int(input()),int(input()))

print("Greater number is",r)

Output:

sum is 30

sum is 30

greater number is 20

Enter two numbers

20

34

Greater number is 34

>>>

**Function Decorators:**

To understand Decorators we need to understand few thing Prior:

1. Namespace and Variable scope
2. LEGB rule
3. Closures
4. Decorators

**Namespace and Variable scope:**

* We use names in python to identify objects so it is called as identifiers
* We use various names like class name, method name, variable name etc.
* So Namespace is basically a system which will control all the names which we use in our program.
* It will assure that whatever names we use is unique and won’t led to any conflict.
* It also allows to reuse the name in the program.
* A python program will have several namespace:
* Built-in namespace : When we start the python interpreter it will create its own built-in namespace
* Global name-space: when we import any module it will create global namespace
* Local namespace: when we call a function it will create its own namespace
* Part of the program where variable is accessible is known as variable scope.
* There are four types of variable scope: Local, global, enclosed and built-in
* We cannot access all the variables from any part of the program, it has its own lifetime and its own scope.
* We will understand the scope of a variable with a small program as follow:

print("initially:",dir()) **#this will print the built in variable**

num=20

def f1():

n=10

print("inside the function:",dir()) **##this will print only n variable name as it is inside a function**

f1()

print("Outside the function:",dir()) **#this will print the builtins and function name**

**Output:**

RESTART: C:/Users/654268/Desktop/Python Programming/49 Namespce dir part 2.py

initially: ['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_']

inside the function: ['n']

Outside the function: ['\_\_annotations\_\_', '\_\_builtins\_\_', '\_\_doc\_\_', '\_\_file\_\_', '\_\_loader\_\_', '\_\_name\_\_', '\_\_package\_\_', '\_\_spec\_\_', 'f1', 'num']

>>>

**LEGB Rule:**

There are four types of scope of a variable:

1. Local: Contains name defined inside the function. Lifetime ends when the function ends
2. Global: Contains name defined at the top level of the script or module. Life time ends when the program ends.
3. Enclosed: Contains name defined inside any all enclosed function, if we define a function inside another function then it is called nested function and the variables are called as enclosed.
4. Built-in: Contains names built in to the python language

y=10 #global variable, can be accessed both inside and outside the function.

def inner(): #this is a global function, this can be called anywhere in code

x=4 #this is local variable

y=5

print("x value inside the function",x)

print("y value Inside the function",y)

print("y:",y) #this will print the global variable value

inner()

Output:

y: 10

x value inside the function 4

y value Inside the function 5

>>>

We cannot modify the global variable inside the local scope, it will throw unbound local error Let us understand this by a program:

y=10 #global variable, can be accessed both inside and outside the function.

def inner(): #this is a global function, this can be called anywhere in code

x=4 #this is local variable

y=y+1

print("x value inside the function",x)

print("y value Inside the function",y)

print("y:",y) #this will print the global variable value

inner()

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/49 Namespace dir part 4.py

y: 10

Traceback (most recent call last):

File "C:/Users/654268/Desktop/Python Programming/49 Namespace dir part 4.py", line 8, in <module>

inner()

File "C:/Users/654268/Desktop/Python Programming/49 Namespace dir part 4.py", line 4, in inner

y=y+1

**UnboundLocalErro**r: local variable 'y' referenced before assignment

>>>

If we want to modify the global variable inside the local scope then we need to use a keyword: “global”

y=10

def inner():

x=4

global y

y=y+1

print("x:",x)

print("inside the function y:",y)

print("y:",y)

inner()

Output:

y: 10

x: 4

inside the function y: 11

>>>

#if the same code is written but print("y:",y) is given after inner function is called then the outputwill differ as:

y=10

def inner():

x=4

global y

y=y+1

print("x:",x)

print("inside the function y:",y)

inner()

print("y:",y)

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/49 Namespace dir part 5.py

y: 10

x: 4

inside the function y: 11

x: 4

inside the function y: 11

y: 11

>>>

**Enclosed Variable:**

y=10

def outer():

z=4 #non-local variable, it has scope both inside and outer function but is not global, hence named as enclosed variable

def inner():

x=4

print("x:",x)

print("inside the function z:",z)

inner()

print("z:",z)

outer()

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/49 Namespace dir part 6 Enclosed variable.py

x: 4

inside the function z: 4

z: 4

>>>

#similar to global keyword we can modify the value of enclosed variable by using nonlocal

y=10

def outer():

z=4 #non-local variable, it has scope both inside and outer function but is not global, hence named as enclosed variable

def inner():

x=4

nonlocal z

z=z+1

print("x:",x)

print("inside the function z:",z)

inner()

print("z:",z)

outer()

Output:

x: 4

inside the function z: 5

z: 5

>>>

Now what is LEGB rule.

Suppose all the variable name is same in local, global, enclosed and built in then how python will understand what to print. Let us understand that by the a program:

#1st Priority

x=5

def function():

x=10

def inner():

x=15 # local scope will be printed as per the scope priority

print("x:",x)

inner()

function()

#2nd priority

x=5

def function():

x=10 # enclosed scope will be printed 2nd as per the scope priority

def inner():

#x=15

print("x:",x)

inner()

function()

#3rd priority

x=5 #global scope will be printed 3rd as per the scope priority

def function():

#x=10

def inner():

#x=15

print("x:",x)

inner()

function()

Output:

==== RESTART: C:/Users/654268/Desktop/Python Programming/49 LEGB rule.py ====

x: 15

x: 10

x: 5

>>>

**Closures:**

To understand closures we need to understand 2 things that is:

1. Nested Functions
2. Functions are first class objects.

Let us understand Nested function with simple program:

def outer():

x=3

def inner():

print(x)

inner()

outer()

Output: 3

We cannot call inner function outside the function body as inner function is local to the outer function body. If we try that it will throw name error as name not defined.

def outer():

x=3

def inner():

print(x)

inner()

outer()

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/50 closures Nested Function.py

Traceback (most recent call last):

File "C:/Users/654268/Desktop/Python Programming/50 closures Nested Function.py", line 5, in <module>

inner()

NameError: name 'inner' is not defined

>>>

Now the next concept is functions are objects.

We can assign functions to a variable.

>>> def f():

print("hi")

>>> f()

hi

>>> f

<function f at 0x03D78420>

>>> g=f

>>> g

<function f at 0x03D78420>

>>> g()

hi

>>>if we call a function without parenthesis, it will give the reference value of the function.

As we know that We cannot call inner function outside the function body as inner function is local to the outer function body. If we try that it will throw name error as name not defined.

However there is technique by which we can achieve that and that technique is known as closure.

Let us see that by program:

def outer():

x=3

def inner():

y=3

result=x+y

return result

return inner()

a=outer()

print(a)

Output:

>> 6

It means we have executed the inner function body outside its scope.

**We can define closure as Function object that remembers values in the enclosing scope even if they are not present in the memory.**

#2nd example

def outer():

msg="hello"

def inner():

print(msg)

return inner

a=outer()

a()

Output:

Hello

Now the question is:

1. What are the criteria to create closure in python.

Answer: it must have a nested function and the nested function must refer to a value defined in the enclosing function and the enclosing function must return nested function.

1. What are the advantages of using closure?

Answer: we can avoid global values, as we saw that we called the local function outside its scope and we printed the local variable outside its scope, hence we avoided the local names using closure.

Data hiding and it allows us to implement decorators which we will see in next chapter.

**Decorators:**

Any callable python object that is used to modify a function or a class.

There are two type of decorator:

1. Function decorator
2. Class decorator.

Function Decorators:

Before understanding function decorator we need to be clear on the concept of nested function, functions can return functions, if we give just function name it will be reference to that function and functions can be used a parameter.

All of those we have covered but functions as parameter is not seen. We will see that and then jump into decorators:

We will try to understand the concept of passing function as parameter by a simple program:

def function1():

print("I am function1")

def function2(func):

print("I am function 2 now i will call function1")

func()

function2(function1)

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/51 Function as parameter.py

I am function 2 now i will call function1

I am function1

>>>

Now we will deep dive into decorators:

def print\_str():

return "goog morning"

print(print\_str())

'''Now suppose we want the message in upper case, then we need to decorate it'''

def str\_upper(func):

def inner():

str1=func()

return str1.upper()

return inner

def print\_str():

return "goog morning"

print(print\_str())

d=str\_upper(print\_str)

print(d())

'''Ways to use decorator for the above program'''

def str\_upper(func):

def inner():

str1=func()

return str1.upper()

return inner

@str\_upper

def print\_str():

return "goog morning"

print(print\_str())

'''with parameters'''

def div\_decorator(func):

def inner(x,y):

if y==0:

return "give proper input"

return func(x,y)

return inner

@div\_decorator

def div(a,b):

return a/b

print(div(4,0))

Output:

= RESTART: C:/Users/654268/Desktop/Python Programming/52 Decorator part 1.py =

goog morning

goog morning

GOOG MORNING

GOOG MORNING

give proper input

>>>

While defining decorators we need to keep 3 things in mind:

1. Need to take a function as parameter
2. Add functionality to the function
3. Function needs to return another function

Now we will learn how to use **multiple decorators in single function** and **decorators with parameter** and lastly how to use a **single decorator in multiple functions**.

Let us understand Multiple decorators in single function using a simple program:

def upper\_d(func):

def inner():

str1=func()

return str1.upper()

return inner

def split\_d(func):

def wrapper():

str2=func()

return str2.split()

return wrapper

@split\_d

@upper\_d

def ordinary():

return "good morning"

print(ordinary())

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/52 Multiple Decorators in single function.py

['GOOD', 'MORNING']

>>>

Now we need to be careful about the order of @split\_d and @upper\_d as if we interchange it in the program it will throw error because if we put upper\_d first then the program will executed the split function first and will return the string in the form of list which it will not be able to convert it in uppercase.

Another Example:

def upper\_d(func):

def inner():

return "first "+func()+" first"

return inner

def split\_d(func):

def wrapper():

return "second "+func()+" second"

return wrapper

@upper\_d

@split\_d

def ordinary():

return "good morning"

print(ordinary())

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/52 Multiple Decorators in single function 2.py

first second good morning second first

>>>

Now we will understand decorators with parameters:

If we want to pass a parameter to the decorator function then we need to take a function, inside which we need to define the decorator function as shown in the below program:

def outer(expr):

def upper\_d(func):

def inner():

return func() + expr

return inner

return upper\_d

@outer(" Gourav")

def ordinary():

return "good morning"

print(ordinary())

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/52 Decorators with parameters.py

good morning Gourav

>>>

Now we will see how to make general decorators so that we can use it in multiple function.

def div\_decorator(func):

def inner(\*args):

list1=[]

list1=args[1:] #parameter present after 1st argument shouldnot be 0

for i in list1:

if i==0:

return "Give proper input!!"

return func(\*args)

return inner

@div\_decorator

def div1(a,b):

return a/b

@div\_decorator

def div2(a,b,c):

return a/b/c

print(div1(10,5))

print(div2(10,0,5))

Output:

RESTART: C:/Users/654268/Desktop/Python Programming/52 Decorators in multiple function..py

2.0

Give proper input!!

>>>

'''Decorators hides many data, to unhide it we need to use functools.

first we will understand how it hides and then we will see how to unhide it'''

def decorator(func):

def inner():

str1=func()

return str1.upper()

return inner

@decorator

def greet():

return "good moring"

print(greet.\_\_name\_\_)

output:

inner

as by output we can see that it is called by inner function which is actually called y greet function, here the data is hidden. Now we will see how to unhide the data and get the actual function name:

import functools

def decorator(func):

@functools.wraps(func)

def inner():

str1=func()

return str1.upper()

return inner

@decorator

def greet():

return "good moring"

print(greet.\_\_name\_\_)

Output:

Greet

So far we have seen applying decorators on functions. We can apply decorator function on methods also.

To understand that we will go through class , objects and methods concept first.

Let us begin with Class

Classes allows us to logically group our data and function in a way that it is easy to reuse and also easy to build upon when needed.

Data and functions associated to a specific class is called as attributes and Methods, so when we say **methods it means the function associated to the class**.

To understand classes in python we can take an example as employee records in a company. Each employee can have specific set of attributes as name, email id, pay and access provided.

We also need to understand the difference between class and instances of class: A class is basically a blue print to create instances. So each user name or employee we create will be an instance of the class.

Instance variable contains data that is unique to each instances.

class Employee:

pass

emp\_1=Employee()

emp\_2=Employee()

print(emp\_1)

print(emp\_2)

emp\_1.first='Corey'

emp\_1.last='Schafer'

emp\_1.email='corey.schafer@company.com'

emp\_1.pay=50000

emp\_2.first='Test'

emp\_2.last='User'

emp\_2.email='Test.User@company.com'

emp\_2.pay=60000

print(emp\_1.email)

print(emp\_2.email)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/53 Understanding Class.py

<\_\_main\_\_.Employee object at 0x03C2C580>

<\_\_main\_\_.Employee object at 0x03C49298>

corey.schafer@company.com

Test.User@company.com

>>>

Now we can see that email Id is created for each of the employee, however this is cumbersome to add manually each time, hence this method of using class will be tough. So to make it automated we need to going to use special init method. when we create method within a class, they receive the instance as first argument automatically. By convention we should call the instance self, After self we can specify what are the arguments we want to accept.

class Employee:

def \_\_init\_\_(self,first, last, pay):

self.first=first #we can give any name as self.fname=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

emp\_1=Employee('corey','schafer',50000) #init method wil run automatically

emp\_2=Employee('Test','User',60000) #so emp\_1 & emp\_2 will be passed as self

#and then it will set all the attributes

'''when we create method within a class,

they receive the instance as first argument automatically'''

'''Now within our init method we are going to set the instance variables '''

'''when we create an instance of employee class which is emp\_1 and emp\_2

we can now pass the values that we specified in our init method.

Now the init method takes the instance which we call self and the first, last

and pay as arguments.'''

##print(emp\_1)

##print(emp\_2)

'''so now that we have init method in place we can delete the manual assignments

as shown below which i wrote at the beginning, here i will comment the codes written below'''

##emp\_1.first='Corey'

##emp\_1.last='Schafer'

##emp\_1.='corey.schafer@company.com'

##emp\_1.pay=50000

##

##emp\_2.first='Test'

##emp\_2.last='User'

##emp\_2.email='Test.User@company.com'

##emp\_2.pay=60000

print(emp\_1.email)

print(emp\_2.email)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/53 Understanding Class.py

corey.schafer@company.com

Test.User@company.com

>>>

If we want to get the full name we can do that in 2 ways:

1. By defining the method
2. By defining function under the class

The first method, we will, have to add a line as: self.fname=first+ " "+last

So the whole program looks like:

class Employee:

def \_\_init\_\_(self,first, last, pay):

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

self.fname=first+ " "+last

emp\_1=Employee('corey','schafer',50000)

emp\_2=Employee('Test','User',60000)

print(emp\_2.fname)

print(emp\_1.fname)

Output:

= RESTART: C:\Users\Dell\Desktop\python\Let-us-learn-python-master\53 Understanding Class.py

Test User

corey schafer

>>>

The second way which is by defining the function under a class, the code will look like as:

class Employee:

def \_\_init\_\_(self,first, last, pay):

self.first=first #we can give any name as self.fname=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

def fname(self):

return'{} {}'.format(self.first,self.last)

emp\_1=Employee('corey','schafer',50000) #init method wil run automatically

emp\_2=Employee('Test','User',60000)

print(emp\_2.fname())

print(emp\_1.fname())

Output:

= RESTART: C:\Users\Dell\Desktop\python\Let-us-learn-python-master\53 Understanding Class.py

Test User

corey schafer

>>>

We can call the methods using class names and also by using instances, in the above example we have used calling by instance name, however, in case of calling by class name it doesn’t know what instance it needs to be run on that method, hence we will have to pass the instance as an argument while calling by function as shown in the code below:

class Employee:

def \_\_init\_\_(self,first, last, pay):

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

def fname(self):

return'{} {}'.format(self.first,self.last)

emp\_1=Employee('corey','schafer',50000)

emp\_2=Employee('Test','User',60000)

print(emp\_1.fname())

print(Employee.fname(emp\_1))

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/53 understanding class part 2.py

corey schafer

corey schafer

>>>

We will get to know about the class variables now:

Class variables are the variables that are shared among all the instances of a class. So while instance variable should be unique for each instance like name, email, pay whereas class variable should be same for each instance. Like the annual raise for each employee. Whenever we access the class variable we need to access it using the class variable itself or instance itself.

If it is accessed by the instances then it will check whether the instance has those attributes or not, if it doesn’t the it will look for the class or the class it inherits from contain those attributes or not.

It is better to call via instance as we can change the constant value if we want for any specific employee.

class Employee:

raise\_amount=1.04

def \_\_init\_\_(self,first,last,pay):

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

def fullnamme(self):

return'{}{}'.format(self.first,self.last)

def apply\_raise(self):

self.pay=int(self.pay \* self.raise\_amount)

emp\_1=Employee('corey','schafer',50000)

emp\_2=Employee('Tst','User',50000)

emp\_1.raise\_amount=1.05

print(Employee.raise\_amount)

print(emp\_1.raise\_amount)

print(emp\_2.raise\_amount)

print(emp\_1.pay)

emp\_1.apply\_raise()

print(emp\_1.pay)

print(emp\_2.pay)

emp\_2.apply\_raise()

print(emp\_2.pay)

Output:

= RESTART: C:\Users\Dell\Desktop\python\Let-us-learn-python-master\54 Understanding class variables.py

1.04

1.05

1.04

50000

52500

50000

52000

>>>

We can also see an example where using instances to call doesn’t make sense. Suppose we want to keep a track for number of employees then we don’t need to use instances to call as under no case we need to change value of any particular instance.

class Employee:

num\_of\_emps=0

raise\_amount=1.04

def \_\_init\_\_(self,first,last,pay):

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

Employee.num\_of\_emps +=1

def fullnamme(self):

return'{}{}'.format(self.first,self.last)

def apply\_raise(self):

self.pay=int(self.pay \* self.raise\_amount)

emp\_1=Employee('corey','schafer',50000)

emp\_2=Employee('Tst','User',50000)

print(Employee.num\_of\_emps)

Output:

= RESTART: C:\Users\Dell\Desktop\python\Let-us-learn-python-master\54 Understanding class variables.py

2

>>>

Now we will learn regular methods class methods and static methods:

class Employee:

num\_of\_emps=0

raise\_amount=1.04

def \_\_init\_\_(self,first,last,pay):

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

Employee.num\_of\_emps +=1

def fullnamme(self):

return'{}{}'.format(self.first,self.last)

def apply\_raise(self):

self.pay=int(self.pay \* self.raise\_amount)

@classmethod

def set\_raise\_amt(cls,amount):

cls.raise\_amt=amount

@classmethod

def from\_string(cls,emp\_str):

first,last,pay=emp\_str.split('-')

return cls(first, last, pay)

emp\_1=Employee('corey','schafer',50000)

emp\_2=Employee('Tst','User',50000)

emp\_str\_1='john-Doe-70000'

emp\_str\_2='Steve-Smith-30000'

emp\_str\_3='Jame-Doe-90000'

new\_emp\_1=Employee.from\_string(emp\_str\_1)

print(new\_emp\_1.email)

print(new\_emp\_1.pay)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/54 understandinf clas methods.py

john.Doe@company.com

70000

The class method in Python is a method, which is bound to the class but not the object of that class. The static methods are also same but there are some basic differences. For class methods, we need to specify @classmethod decorator, and for static method @staticmethod decorator is used.

Syntax for Class Method.

class my\_class:

@classmethod

deffunction\_name(cls, arguments):

#Function Body

return value

Syntax for Static Method.

class my\_class:

@staticmethod

deffunction\_name(arguments):

#Function Body

return value

**What are the differences between Classmethod and StaticMehtod?**

|  |  |
| --- | --- |
| **Class Method** | **Static Method** |
| The class method takes cls (class) as first argument. | The static method does not take any specific parameter. |
| Class method can access and modify the class state. | Static Method cannot access or modify the class state. |
| The class method takes the class as parameter to know about the state of that class. | Static methods do not know about class state. These methods are used to do some utility tasks by taking some parameters. |
| @classmethod decorator is used here. | @staticmethod decorator is used here. |

The Static methods are used to do some utility tasks, and class methods are used for factory methods. The factory methods can return class objects for different use cases.

**Example code**

from datetime import date as dt

class Employee:

def \_\_init\_\_(self, name, age):

self.name = name

self.age = age

@staticmethod

defisAdult(age):

if age > 18:

return True

else:

return False

@classmethod

defemp\_from\_year(emp\_class, name, year):

return emp\_class(name, dt.today().year - year)

def \_\_str\_\_(self):

return 'Employee Name: {} and Age: {}'.format(self.name, self.age)

e1 = Employee('Dhiman', 25)

print(e1)

e2 = Employee.emp\_from\_year('Subhas', 1987)

print(e2)

print(Employee.isAdult(25))

print(Employee.isAdult(16))

**Output**

Employee Name: Dhiman and Age: 25

Employee Name: Subhas and Age: 31

True

False

**Inheritance using subclasses:**

Inheritance allows us to inherit attributes and methods from a parent class. This is useful because we can create subclasses and can get all the functionality of a parent class and we can override and add completely new functionality without effecting the parent class.

As we have done in previous examples with employee class, now suppose we want t see what role does the employee have for example is the person developer or sales and marketing, in both cases employees will have the same data as Employee class would have, so in this case we will inherit the attributes of the class Employee.

class Employee:

raise\_amt=1.04

def \_\_init\_\_(self,first,last,pay):

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

def fullname(self):

return '{}{}'.format(self.first,self.last)

def apply\_raise(self):

self.pay=int(self.pay\*self.raise\_amt)

class Developer(Employee):

raise\_amt=1.10

dev\_1=Developer('Corey','Schafer',50000)

dev\_2=Developer('Test','Employee',60000)

print(dev\_1.email)

print(dev\_2.email)

print(dev\_1.pay)

dev\_1.apply\_raise()

print(dev\_1.pay)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/56 Understanding Class inheritance.py

Corey.Schafer@company.com

Test.Employee@company.com

50000

55000

>>>

We can see that developer class has inherited the attributes from Employee class and the raise amount for a developer has been changed which didn’t impact the primary class, hence we can make changes in subclasses without worrying about the parent class.

Now lets make the program more complicated, lets take some attributes in developer class which is out of scope of parent class. Lets associate programming language with the developer class which is not present in Employee class.

We need to define the init method for developer class like def \_\_init\_\_(self,first,last,pay,prog\_lang): we shouldn’t copy the code:

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

and paste it agin afer the init method of developer class. We can ask the primary class to take the first three values and there are 2 ways of doing so:

super().\_\_init\_\_(first,last,pay) and Employee.\_\_init\_\_(self,first,last,pay)

super() method is preferable.

class Employee:

raise\_amt=1.04

def \_\_init\_\_(self,first,last,pay):

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

def fullname(self):

return '{}{}'.format(self.first,self.last)

def apply\_raise(self):

self.pay=int(self.pay\*self.raise\_amt)

class Developer(Employee):

raise\_amt=1.10

def \_\_init\_\_(self,first,last,pay,prog\_lang):

super().\_\_init\_\_(first,last,pay)

self.prog\_lang=prog\_lang

dev\_1=Developer('Corey','Schafer',50000,'python')

dev\_2=Developer('Test','Employee',60000,'Java')

print(dev\_1.email)

print(dev\_1.prog\_lang)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/56 Understanding Class inheritance.py

Corey.Schafer@company.com

python

>>>

Now lets make the program more complex by adding manager and showing who reports under manager.

class Employee:

raise\_amt=1.04

def \_\_init\_\_(self,first,last,pay):

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

def fullname(self):

return '{}{}'.format(self.first,self.last)

def apply\_raise(self):

self.pay=int(self.pay\*self.raise\_amt)

class Developer(Employee):

raise\_amt=1.10

def \_\_init\_\_(self,first,last,pay,prog\_lang):

super().\_\_init\_\_(first,last,pay)

self.prog\_lang=prog\_lang

class Manager(Employee):

def \_\_init\_\_(self,first,last,pay,employees=None):

super().\_\_init\_\_(first,last,pay)

if employees is None:

self.employees=[]

else:

self.employees=employees

def add\_emp(self,emp):

if emp not in self.employees:

self.employees.append(emp)

def remove\_emp(self,emp):

if emp in self.employees:

self.employees.remove(emp)

def print\_emps(self):

for emp in self.employees:

print('==>',emp.fullname())

dev\_1=Developer('Corey','Schafer',50000,'python')

dev\_2=Developer('Test','Employee',60000,'Java')

mgr\_1=Manager('sue','smith',90000,[dev\_1])

print(mgr\_1.email)

mgr\_1.add\_emp(dev\_2)

##mgr\_1.remove\_emp(dev\_1)

mgr\_1.print\_emps()

##print(dev\_1.email)

##print(dev\_1.prog\_lang)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/56 Understanding Class inheritance.py

sue.smith@company.com

==> CoreySchafer

==> TestEmployee

>>>

We have few more features that is it can tell whether an instance is an object of a class or not, for that we use isinstance built in function.similarly we have issubclass which tells us whether a class is a subclass of another.

class Employee:

raise\_amt=1.04

def \_\_init\_\_(self,first,last,pay):

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

def fullname(self):

return '{}{}'.format(self.first,self.last)

def apply\_raise(self):

self.pay=int(self.pay\*self.raise\_amt)

class Developer(Employee):

raise\_amt=1.10

def \_\_init\_\_(self,first,last,pay,prog\_lang):

super().\_\_init\_\_(first,last,pay)

self.prog\_lang=prog\_lang

class Manager(Employee):

def \_\_init\_\_(self,first,last,pay,employees=None):

super().\_\_init\_\_(first,last,pay)

if employees is None:

self.employees=[]

else:

self.employees=employees

def add\_emp(self,emp):

if emp not in self.employees:

self.employees.append(emp)

def remove\_emp(self,emp):

if emp in self.employees:

self.employees.remove(emp)

def print\_emps(self):

for emp in self.employees:

print('==>',emp.fullname())

dev\_1=Developer('Corey','Schafer',50000,'python')

dev\_2=Developer('Test','Employee',60000,'Java')

mgr\_1=Manager('sue','smith',90000,[dev\_1])

print (isinstance(mgr\_1,Manager))

print (isinstance(mgr\_1,Employee))

print (isinstance(mgr\_1,Developer))

print(issubclass(Developer,Employee))

print(issubclass(Manager,Developer))

print(issubclass(Manager,Employee))

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/56 Understanding Class inheritance.py

True

True

False

True

False

True

>>>

**Operator overloading in python:**

By defining our own special methods we will be able to change the behavior of built in methods.

These methods are surrounded by double underscore(\_\_). This are called as dunder init.we have come across this method. Let us now understand dunder str and dunder repr.

class Employee:

def \_\_init\_\_(self,first, last, pay):

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

def fullname(self):

return'{} {}'.format(self.first,self.last)

def apply\_raise(self):

self.pay=int(self.pay \* self.raise\_amt)

def \_\_repr\_\_(self):

return "Employee('{}','{}','{}')".format(self.first,self.last,self.pay)

def \_\_str\_\_(self):

return '{}-{}'.format(self.fullname(),self.email)

emp\_1=Employee('corey','schafer',50000)

emp\_2=Employee('Test','User',60000)

print(emp\_1)

print(repr(emp\_1))

print(str(emp\_1))

print(emp\_1.\_\_repr\_\_())

print(emp\_1.\_\_str\_\_())

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/57 Operator overloading.py

corey schafer-corey.schafer@company.com

Employee('corey','schafer','50000')

corey schafer-corey.schafer@company.com

Employee('corey','schafer','50000')

corey schafer-corey.schafer@company.com

>>>

>>> print(int.\_\_add\_\_(3,4)) Dunder Add for integer

7

>>> print(str.\_\_add\_\_('a','b')) Dunder Add for strings

ab

>>>

class Employee:

def \_\_init\_\_(self,first, last, pay):

self.first=first

self.last=last

self.pay=pay

self.email=first+'.'+last+'@company.com'

def fullname(self):

return'{} {}'.format(self.first,self.last)

def apply\_raise(self):

self.pay=int(self.pay \* self.raise\_amt)

def \_\_repr\_\_(self):

return "Employee('{}','{}','{}')".format(self.first,self.last,self.pay)

def \_\_str\_\_(self):

return '{}-{}'.format(self.fullname(),self.email)

def \_\_add\_\_(self,other):

return self.pay+other.pay

def \_\_len\_\_(self):

return len(self.fullname())

emp\_1=Employee('corey','schafer',50000)

emp\_2=Employee('Test','User',60000)

print(emp\_1+emp\_2)

print(len(emp\_1))

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/57 Operator overloading.py

110000

13

>>>

Let us now understand property decorators:

class Employee:

def \_\_init\_\_(self,first,last):

self.first=first

self.last=last

self.email=first+'.'+last+'@company.com'

def fullname(self):

return'{} {}'.format(self.first,self.last)

emp\_1=Employee('corey','schafer')

print(emp\_1.first)

print(emp\_1.email)

print(emp\_1.fullname())

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/58 Property decorators.py

corey

corey.schafer@company.com

corey schafer

>>>

What if we change the first name as emp\_1.first=’jim’

class Employee:

def \_\_init\_\_(self,first,last):

self.first=first

self.last=last

self.email=first+'.'+last+'@company.com'

def fullname(self):

return'{} {}'.format(self.first,self.last)

emp\_1=Employee('corey','schafer')

emp\_1.first='jim'

print(emp\_1.first)

print(emp\_1.email)

print(emp\_1.fullname())

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/58 Property decorators.py

jim

corey.schafer@company.com

jim schafer

>>>

So we would like to change the entire data when the first name is modified. To do so we need to appl decorators.

class Employee:

def \_\_init\_\_(self,first,last):

self.first=first

self.last=last

@property

def email(self):

return'{}.{}@email.com'.format(self.first,self.last)

@property

def fullname(self):

return'{} {}'.format(self.first,self.last)

@fullname.setter

def fullname(self,name):

first,last=name.split(' ')

self.first=first

self.last=last

emp\_1=Employee('corey','schafer')

##emp\_1.fullname='corey schafer'

emp\_1.first='jim'

print(emp\_1.first)

print(emp\_1.email)

print(emp\_1.fullname)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/58 Property decorators.py

jim

jim.schafer@email.com

jim schafer

>>>

Another example:

def check\_name(method):

def inner(name\_ref):

if name\_ref.name=="gourav":

print("Hey my name is also same!!!")

else:

method(name\_ref)

return inner

class printing:

def \_\_init\_\_(self,name):

self.name=name

@check\_name

def print\_name(self):

print("entered user name is:",self.name)

p=printing("gourav")

p.print\_name()

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/58 property decorators part 2.py

Hey my name is also same!!!

>>>

**Property decorator deep dive:**

We can use class methods as attributes which is facilitated by property decorators.

class student:

def \_\_init\_\_(self,name,grade):

self.name=name

self.grade=grade

def msg(self):

return self.msg=self.name +"got grade" + self.grade

stud1=student("Nia","B")

print(stud1.name)

print(stud1.grade)

print(stud1.msg())

In the above program msg is converted int method due to which to print it we need parenthesis while calling, but this might be cumbersome for the client who will be working on huge data, hence we will have to apply inbuilt decorator which is property decorator as shown below:

class student:

def \_\_init\_\_(self,name,grade):

self.name=name

self.grade=grade

@property

def msg(self):

return self.msg=self.name +"got grade" + self.grade

stud1=student("Nia","B")

print(stud1.name)

print(stud1.grade)

print(stud1.msg)

Niow what if the cliet comes and customize the message to something different, in that case we need to use setter, to use setter we need to give the method name dot setter.

class student:

def \_\_init\_\_(self,name,grade):

self.name=name

self.grade=grade

@property

def msg(self):

return self.name +" got grade"+ self.grade

@msg.setter

def msg(self,msg):

sent=msg.split(" ")

self.name=sent[0]

self.grade=sent[-1]

stud1=student("Nia","B")

stud1.msg="amulya got grade A"

print(stud1.name)

print(stud1.grade)

print(stud1.msg)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/59 Property decorator deep dive.py

amulya

A

amulya got gradeA

>>>

One more example for decorator:

def decor\_result(result\_func):

def distinction(marks):

for m in marks:

if m>=75:

print("Congrats!! you have got distinction")

else:

result\_func(marks)

return distinction

@decor\_result

def result(marks):

for m in marks:

if m>=33:

pass

else:

print("FAIL")

break

else:

print("PASS")

result([50,60,15,90,66])

Output:

=================== RESTART: C:/Users/Dell/Desktop/sample.py ===================

Congrats!! you have got distinction

FAIL

>>>

**Exception Handling in python:**

When we write a program we might sometimes miss parenthesis or colon etc, this will be called as syntax error. On contrary to that we sometimes get the error which doesn’t occur because of the syntax rather it occurs during the runtime of the program and are known as exceptions or runtime error.

Let us understand this by a program:

x=int(input("Enter first number:"))

y=int(input("Enter second number"))

print("Sum is:",x+y)

This program will not throw any error and the output console will be like:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/61 understanding Exceptional error.py

Enter first number:3

Enter second number4

Sum is: 7

>>>

What if we don’t put int in any ne input like as:

x=int(input("Enter first number:"))

y=(input("Enter second number"))

print("Sum is:",x+y)

the above program will throw type error as:

Traceback (most recent call last):

File "C:/Users/Dell/Desktop/python/Let-us-learn-python-master/61 understanding Exceptional error.py", line 3, in <module>

print("Sum is:",x+y)

**TypeError**: unsupported operand type(s) for +: 'int' and 'str'

>>>

Similarly if we write the same program for division and put the value of y as 0 then it will give **zero division error.**

Error appearing that is **TypeError & zero division error** are the name of a class. There are several such classes and they are called as exception classes. So error occurring during runtime some classes are inbuilt in python. There is a mechanism which works in the background for this predefined classes error as per the user’s input, this mechanism is known as **Default Except Mechanism.**

When there is error in program, python by default **raise** the error as per the category and terminates the program there itself and doesn’t execute further.

Now suppose we want the program to run even after the default except mechanism takes place. All program has two flows which is **Normal Flow and Exceptional Flow.** Exceptional flow occurs when we give value of y as 0 which we saw in an example. We can systematically handle the exceptions and make the program run even when the exception occurs. For these we need to **stop the Default Except Mechanism.** To do so we need to make our own exception mechanism, so that when a program runs we default exception mechanism doesn’t run.one thing to keep in note is that if our code is not capable to run then the default except mechanism will run. This is called as Exceptional handling.

There are 4 situations in exceptional handling:

|  |  |
| --- | --- |
| Python | DEM |
| Python | EM |
| User | DEM |
| user | EM |

1. When python raise its own exception default exception mechanism (DEM) comes into play.
2. Suppose if we writing the program and wants our exception to be raised then user defined exception mechanism works
3. Suppose, we are writing a ATM software code and the user gives the withdrawal amount more then account balance, in this case its not the problem of python code, here we need to raise exception in python on which DEM will work.
4. When we only raise exception and want our the exceptional handling to be done as per our exceptional mechanism.

First scenario is the default one. Let us understand the second scenario where python will raise the exception but will be handled by our mechanism. For this we will use the keywords try, except and finally.

The code where there are chances that exception may occur, such risky codes are written under **try** block. Next we will use keyword **except** where we will have to give the exception class name which should be raised.

Syntax:

try:

statement 1

statement 2

except ExceptionClassName:

statement.

By coding so even if the exception occurs which is built by us, the program will not stop and will execute further.

Let us see that by a program:

x=int(input("Enter first number:"))

y=int(input("Enter second number"))

try:

z=x/y

print("Division result:",z)

except ZeroDivisionError:

print("Invalid attempt of division")

print("Helllo this is last line")

Output:

= RESTART: C:\Users\Dell\Desktop\python\Let-us-learn-python-master\61 understanding Exceptional error.py

Enter first number:10

Enter second number0

Invalid attempt of division

Helllo this is last line

>>>

So it is clear about try and except block, we will dig about the rules related to it.

1. We will write only those codes in try block which has chances to raise exception
2. We cannot use except block without try block
3. Exception class name should be matching with the type of exception which is going to raise in try block
4. If there is no exception in try block, the exception block will be skipped
5. It is not necessary that for a single try block only one except block will be there, for a single try block there can be multiple except block.
6. The reason behind giving multiple expect block is that we can give several lines in try block for which different exception may occur and we can give as many exceptions as many it has chances to occur. However none of the exception matches as per the try block then default exception mechanism will work.

We will see rule number 6 by a program when we don’t give the exception class name as per the exception which can occur in try block, here we will give Type Error instead of zero division error.

s=int(input("Enter first number:"))

t=int(input("Enter second number"))

try:

z=s/t

print("Division result:",z)

except TypeError:

print("Invalid attempt of division")

print("Helllo this is last line")

Output:

Traceback (most recent call last):

File "C:\Users\Dell\Desktop\python\Let-us-learn-python-master\61 understanding Exceptional error.py", line 15, in <module>

z=s/t

ZeroDivisionError: division by zero

>>>

Now let us understand about the keyword finally:

It is such a keyword which will always run, Even if the exception class doesn’t matches with the exceptions before the default exception mechanism runs, hence finally will always run before the DEM. Let us see that via a program:

s=int(input("Enter first number:"))

t=int(input("Enter second number"))

try:

z=s/t

print("Division result:",z)

except TypeError:

print("Invalid attempt of division")

finally:

print("In Finally")

print("Helllo this is last line")

Output:

Enter first number:10

Enter second number0

In Finally

Traceback (most recent call last):

File "C:\Users\Dell\Desktop\python\Let-us-learn-python-master\61 understanding Exceptional error.py", line 15, in <module>

z=s/t

ZeroDivisionError: division by zero

>>>

One important thing to note that is we can write finally block only after the exception block, we cannot put finally block prior to exception block.

Second thing to note is that, if we are putting a try block in a program then it is necessary that either exception or finally block should be given, if we don’t give any exception block then finally block is must. We can give multiple try blocks in a program.

Third thing to note is that instead of writing multiple expect block we can mention all the exception class name in a single except block like as follows:

s=int(input("Enter first number:"))

t=int(input("Enter second number"))

try:

z=s/t

print("Division result:",z)

except (TypeError, ValueError, ZeroDivisionError):

print("Invalid attempt of division")

finally:

print("In Finally")

print("Helllo this is last line")

we use this when we want some kind of common task to perform.

Fourth thing to note is that, suppose we have written a code where none of the exception class matches with the exception raised but we don’t want the DEM to run in such scenario, python gives us the privillage to overcome that like as:

s=int(input("Enter first number:"))

t=int(input("Enter second number"))

try:

z=s/t

print("Division result:",z)

except (TypeError):

print("Invalid attempt of division")

**except:**

**print("Default Exception")**

finally:

print("In Finally")

print("Helllo this is last line")

Output:

Enter first number:10

Enter second number0

Default Exception

In Finally

Helllo this is last line

>>>

Fifth thing to note is that we can use else with try block. Else will get execute when none of the exception is raised in try block. So the difference in try and else block is that finally will run always in any situation but else block will run only if no exception is found.

We have seen 2 scenarios so far where python was raising the inbuilt exception class and also we have given our exception mechanism, now we will see how we can raise exception.

We will see a program where we have raised an exception and handled by us:

x=int(input("Enter first number"))

y=int(input("Enter second number"))

if y==0:

raise ZeroDivisionError("Denominator cannot be zero")

z=x/y

print("Division is:",z)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/62 Understanding exceptional error part 2.py

Enter first number10

Enter second number0

Traceback (most recent call last):

File "C:/Users/Dell/Desktop/python/Let-us-learn-python-master/62 Understanding exceptional error part 2.py", line 4, in <module>

raise ZeroDivisionError("Denominator cannot be zero")

**ZeroDivisionError: Denominator cannot be zero**

>>>

We can see in above program that the default message which use to be printed for ZeroDivisionError is now changed and is showing our customized error message. So this situation is when we raised an exception but didn’t handled.

Now we will see when we raise the exception and we handle it.

x=int(input("Enter first number"))

y=int(input("Enter second number"))

try:

if y==0:

raise ZeroDivisionError("Denominator cannot be zero")

z=x/y

print("Division is:",z)

except ZeroDivisionError:

print("you cannot divide by zero")

Output:

Enter first number10

Enter second number0

you cannot divide by zero

>>>

Now we will understand user define exception class. We have seen the example for ATM where user is trying to input the amount higher than current balance. We will create our own exception class “insufficient balance”

class InsufficientBalance(ZeroDivisionError):

def \_\_init\_\_(self,arg):

self.msg=arg

balance=5000

w=int(input("enter amount to withdraw"))

if w>balance:

raise InsufficientBalance("Insufficient balance in the account")

balance=balance-w

print("Current Balance is:",balance)

Output:

enter amount to withdraw6000

Traceback (most recent call last):

File "C:/Users/Dell/Desktop/python/Let-us-learn-python-master/63 Userdefined Exception.py", line 7, in <module>

raise InsufficientBalance("Insufficient balance in the account")

InsufficientBalance: Insufficient balance in the account

>>>

In the above program we have raised the exception and DEM has handled the exception, now we want to handle the exception so the code will be as:

class InsufficientBalance(ZeroDivisionError):

def \_\_init\_\_(self,arg):

self.msg=arg

balance=5000

w=int(input("enter amount to withdraw"))

try:

if w>balance:

raise InsufficientBalance("Insufficient balance in the account")

balance=balance-w

except InsufficientBalance as i:

print("Exception",i.msg)

else:

print("Withdraw amount",w,"Successfully")

finally:

print("Current Balance is:",balance)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/63 Userdefined Exception.py

enter amount to withdraw8000

Exception Insufficient balance in the account

Current Balance is: 5000

>>>

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/63 Userdefined Exception.py

enter amount to withdraw4000

Withdraw amount 4000 Successfully

Current Balance is: 1000

>>>

Most important thing is catching classes that do not inherit from BaseException is not allowed which means we will have to give parent class name while creating exception class of our own as from previous example the line is : class InsufficientBalance(**ZeroDivisionError**):

Now we will understand the nesting of tr block:

try:

print("line1")

print("line2")

try:

print("line3")

print("line4")

except ZeroDivisionError:

print("Except1")

finally:

print("finally1")

print("line5")

except TypeError:

print("Except2")

finally:

print("finally2")

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/64 Nestingof try block.py

line1

line2

line3

line4

finally1

line5

finally2

>>>

The above is without exception raised, now lets see I the exception occurs the how the code will behave.

try:

print("line1")

3/0

print("line2")

try:

print("line3")

print("line4")

except ZeroDivisionError:

print("Except1")

finally:

print("finally1")

print("line5")

except TypeError:

print("Except2")

finally:

print("finally2")

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/64 Nestingof try block.py

line1

finally2

Traceback (most recent call last):

File "C:/Users/Dell/Desktop/python/Let-us-learn-python-master/64 Nestingof try block.py", line 3, in <module>

3/0

ZeroDivisionError: division by zero

>>>

As the exception occurd in outer try block so the inner try block’s exception and finally will not run and jump to the type error exception but 3/0 is not a type error so DEM will run.

Now lets see how code behaves when we raise exception inside the inner try block:

try:

print("line1")

print("line2")

try:

print("line3")

3/0

print("line4")

except ZeroDivisionError:

print("Except1")

finally:

print("finally1")

print("line5")

except TypeError:

print("Except2")

finally:

print("finally2")

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/64 Nestingof try block.py

line1

line2

line3

Except1

finally1

line5

finally2

>>>

Now let us go through one important concept in this code by editing the inner ty block as type error

try:

print("line1")

print("line2")

try:

print("line3")

3+"5"

print("line4")

except ZeroDivisionError:

print("Except1")

finally:

print("finally1")

print("line5")

except TypeError:

print("Except2")

finally:

print("finally2")

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/64 Nestingof try block.py

line1

line2

line3

finally1

Except2

finally2

>>>

As we can see that line5 is not printed but Except2 is printed. It is because the inner try blocks whole code is considered as a single statement of outer try block and as in inner try block the exception was unhandled it comes to out of inner try block and consider the next line as statement of outer try block and prints except2 as it matches with the exception raised.

**Object Oriented programming In python**

so far we have done several programs where we use to define functions and call those functions for data manipulation such approach is known as procedural oriented way of programming, procedure here represents function.

We have another way of programming which is Object oriented programming. In object oriented we treat data and function as a single entity, in procedural way we were defining function to work on data but in object approach we are treating both as a single entity which is known as object.

**Main Aspects of OOP:**

**Classes and Objects-**

Class is a keyword which has some name when we make it similar while defining function.

Class encapsulates data and functions. Creating class means creating type (similar to data type)

Class is a description of an object or we can say class is a blueprint of an object. For an example for a house construction a map is made prior and then accordingly house is made so the map is the class and the house is an object.

Int, float, list, set, tuple, dict, str, complex and float etc are all built in clsses in python which we have studied so far.

Any object can access its function by applying a dot (.) after the object name. for example we have assigned x=5 which means x is int type and this int class has its own built-in functions and we can call those function as below:

>>> x=5

>>> x.bit\_length

<built-in method bit\_length of int object at 0x646BF7F0>

>>> x.bit\_length()

3

>>>

Built\_length is a inbuilt function in int class which shows the result of requirement of bits when the value is converted in binary.

Variables are also referred as fields and function as methods. Together they are called as attributes. Suppose if we say there are 10 attributes in x, it means we can access 10 things from x.(xyz)

So object is an instance of a class. We can create n number of objects of some class but we have to define it first.

In the below example we will see that for a class int there are three objects which containing different value.

>>> x=5

>>> y=4

>>> z=3

>>> type(x),type(y),type(z)

(<class 'int'>, <class 'int'>, <class 'int'>)

>>>

So far we understood class theoretically, now we will see how we can make class.

Syntax:

Class class name:

Variable

Function

So basically by making class we have combined all the variable and functions under one namespace.

For example:

class test:

i=5

def f1():

print("Hello")

print(test.i)

test.f1()

'''we have accessed the class member and this is test.i and test.f1()

is known as class object.

we need to understand that we haven't created object for the class yet and there

is a difference between class object and object'''

'''we can use it for instanciation or to create object like t1=test(), here class

object is used as a fucntion, it creates an object of a class test'''

we will see now how \_\_init\_\_ function is made in class.

\_\_init\_\_ function gets automatically called when an object is crated and it gets called only once for one object. So whenever an object is created \_\_init\_\_ function gets called.

If \_\_init\_\_ function is not created in class then if an object is created no onternal function will be called.

\_\_init\_\_ function takes minimum 1 argument compulsory, however it can take multiple argument

class test:

i=10

def \_\_init\_\_(self,x,y):

self.a=x #a and b is an instance variable

self.b=y

t1=test(10,20) #instantiation (object created so \_\_init\_\_ function will be called.

t2=test(3,4)

t3=test(100,50)

print(t1.a,t1.b)

print(t2.a,t2.b)

print(t3.a,t3.b)

Output:

== RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/66 init.py =

10 20

3 4

100 50

>>>

Difference between \_\_init\_\_ function and normal function:

|  |  |
| --- | --- |
| \_\_init\_\_ | Function |
| Implicit call | Explicit call |
| One time call | Any number of time it can be called |
| To declare instance variable | For some business logic or to process something |
| One argument is must | No such compulsion |

**Assignment 14:**

1. **Define a class with instance variable roll no, name, semester and branch. Also define instance member function for user input data to set values of instance variable and display students data.**

class student:

def inputstudent(self):

self.rollno=int(input("Enter roll number"))

self.name=input("Enter student's name:")

self.semester=int(input("Enter semester as integer"))

self.branch=input("Enter branch")

def showstudent(self):

print("Roll No:",self.rollno)

print("Name:",self.name)

print("Semester:",self.semester)

print("Branch:",self.branch)

Output:

class student:

def inputstudent(self):

self.rollno=int(input("Enter roll number"))

self.name=input("Enter student's name:")

self.semester=int(input("Enter semester as integer"))

self.branch=input("Enter branch")

def showstudent(self):

print("Roll No:",self.rollno)

print("Name:",self.name)

print("Semester:",self.semester)

print("Branch:",self.branch)

1. **Define a class employee with instance variable empid , name and salary. Define constructor to initialize member variables. Define functions to show employee data.**

class employee:

def \_\_init\_\_(self,empid,name,salary):

self.empid=empid

self.name=name

self.salary=salary

def showemployeedata(self):

print("Empoyee ID:",self.empid)

print("Name:",self.name)

print("salary:",self.salary))

Output:

>>> e1=employee(5,"Gourav",45000)

>>> e1.showemployeedata()

Empoyee ID: 5

Name: Gourav

salary: 45000

>>>

1. **Using class employee (question 2) , create a list of employee (data taken from user) and display list of employee in sorted order according to their names. Also define a function to sort a list of employees according to their salary in descending order.**

class employee:

def inputemployee(self):

self.empid=int(input("Enter the employee ID:"))

self.name=input("Enter the name:")

self.salary=input("Enter the salary:")

return self

def showemployee(self):

print("Employee ID:",self.empid,end=' ')

print("Name:",self.name,end=' ')

print("salary:",self.salary)

@staticmethod

def sortbyname(emp\_list):

emp\_list.sort(key=lambda e:e.name)

@staticmethod

def sortbysalary(emp\_list):

emp\_list.sort(key=lambda e:e.salary, reverse=True)

l1=[employee().inputemployee()

for i in range(int(input("How many employee data you want to enter")))]

employee.sortbyname(l1)

print("list of employee sorted by their names")

for e in l1:

e.showemployee()

employee.sortbysalary(l1)

print("List of employees sorted by their salary in reverse")

for e in l1:

e.showemployee()

Output:

How many employee data you want to enter3

Enter the employee ID:654268

Enter the name:Gourav

Enter the salary:36000

Enter the employee ID:678029

Enter the name:Suraj

Enter the salary:34000

Enter the employee ID:654273

Enter the name:sreya

Enter the salary:45000

list of employee sorted by their names

Employee ID: 654268 Name: Gourav salary: 36000

Employee ID: 678029 Name: Suraj salary: 34000

Employee ID: 654273 Name: sreya salary: 45000

List of employees sorted by their salary in reverse

Employee ID: 654273 Name: sreya salary: 45000

Employee ID: 654268 Name: Gourav salary: 36000

Employee ID: 678029 Name: Suraj salary: 34000

1. **Define a class Book to store book related information like book ID, title, price, author, publisher. Define functions to input, show, change price.**

**Types of variable:**

1. **Instance Variable**
2. **Static Variable**
3. **Local Variable**
4. **Global Variable**

We will see first how to create instance variable:

There are 3 methods of doing so:

1. \_\_init\_\_ method
2. Methods
3. Outside the class

**\_\_init\_\_**

Variables declared under class are not instance variable..

1st method:

class Account:

def \_\_init\_\_(self,a,b):

self.accno=a

self.balance=b

acc1=Account(100,4000)

print(acc1.\_\_dict\_\_)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/67 Instnce variable.py

{'accno': 100, 'balance': 4000}

>>>

2nd method:

Class Account():

def f1(self,a,b):

self.accno=a

self.balance=b

acc1=Account()

acc1.f1(101,4500)

print(acc1.\_\_dict\_\_)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/67 Instnce variable.py

{'accno': 101, 'balance': 4500}

>>>

3rd method:

Class Account():

def f1(self,a,b):

self.accno=a

self.balance=b

acc1=Account()

acc1.accno=102

acc1.balance=6000

print(acc1.\_\_dict\_\_)

outpu:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/67 Instnce variable.py

{'accno': 102, 'balance': 6000}

>>>

**Static Variable:**

* There is no static keyword in python
* This is class specific not object specific.
* As instance variable is object specific, static variable creates a single copy for the entire class.
* It is a shared variables among all the objects.
* For example rate of interest can be referred as static in account information program.
* While defining function if we don’t pass any argument then such functions are called as static function or static method.
* In some scenarios it is necessary to use the annotation **@staticmethod ,**  however it is a better practice to use this annotation whenever we use static method.
* **When we call via class object dot(.) ,current object doesn’t pass implicitly whereas when we call by object the current object is passed implicitly. So basically when we are calling via class name it is treated as static function.**

Ways of defining static variable:

class test:

a=10 #static variable

def \_\_init\_\_(self):

self.x=1 #instance variable

test.b=20 #static variable

def f1(self): #instance member function

self.x=2 #instance variable

test.c=30 #static variable

@staticmethod

def f2(m,n): #static method

test.d=40 #static variable

@classmethod

def f3(cls):

cls.e=50 #static variable

test.f=60 #static variable

test.g=70 #static variable

t1=test()

t1.f1()

test.f2(3,4)

test.f3()

print(test.\_\_dict\_\_)

outut:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/68 static variable.py

{'\_\_module\_\_': '\_\_main\_\_', 'a': 10, '\_\_init\_\_': <function test.\_\_init\_\_ at 0x03275C40>, 'f1': <function test.f1 at 0x03275C88>, 'f2': <staticmethod object at 0x02FDC580>, 'f3': <classmethod object at 0x0326A370>, '\_\_dict\_\_': <attribute '\_\_dict\_\_' of 'test' objects>, '\_\_weakref\_\_': <attribute '\_\_weakref\_\_' of 'test' objects>, '\_\_doc\_\_': None, 'g': 70, 'b': 20, 'c': 30, 'd': 40, 'e': 50, 'f': 60}

>>>

**The key difference between class method and static method is that class method can take implicit argument which is via class object where as in static method there is no implicit argument.**

**The similarity between them is that they cannot access instance member.**

**Local and Global variable:**

Everything is discussed in LEGB rule.

The only thing which is not discussed in LEGB rule is that we can access both local and global variable inside the function by using the keyword **globals()** as shown in the below program:

y=10 #Global Variable

print("outside functiony=",y)

def f1():

y=5 #local variable

print("Inside function local y=",y)

print("Inside function Global y=",**globals()['y']**)

f1()

print("outside function y=",y)

output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/69 Local and Global Variable.py

outside functiony= 10

Inside function local y= 5

Inside function Global y= 10

outside function y= 10

>>>

**Inheritance in python:**

**Encapsulation**: it is an act of combining properties and methods related to the same entity.

Inheritance is one of the key principle of OOP.

Inheritance is defining a new class with the help of an old class.

**Why Inheritance**:

Suppose we are writing a program where we are providing details of few person with entities name and age and also we need to define another class of students with roll no. , age and name. so basically we will write the same code for attributes age and name both the class person and name, this will led to more LOC which we need to overcome.

In such cases we can inherit the code of Person class and **reuse**  in Student class so that we do not need to write the same code again and again. This feature is called as inheritance.

We can say the person class as **super/old/base/parent** whereas we can say the **subtype** of person which is student class as  **sub/new/derived/child**.

**Syntax: class derivedclass(base): and class derivedclass(modue.base)**

class Person:

def \_\_init\_\_(self,n,a):

self.name=n

self.age=a

def showName(self):

print("Name:",self.name)

def showAge(self):

print("Age:",self.age)

class Student(Person):

def \_\_init\_\_(self,r):

self.rollno=r

Person.\_\_init\_\_(self,"Rahul",16) """This will allow the child class parameters to inherit from parent class,we can write this line before self.rollno=r. **we can write an alternative line also which is: super().\_\_init\_\_(“Rahul”,16)** here we are using the keyword **super with parenthesis,** however we will not use **self** as it is explicitly passed"""

def showRollno(self):

print("Roll Number:",self.rollno)

s1=Student(100)

s1.showRollno()

s1.showName()

s1.showAge() '''this will first search in student class first if not found then it

will go to parent class to see the arguments but if it get in child class then

it will not go to parent class. so be careful with this point. even if there

is function in child class and the arguments differ from the parent class then

also it will not go to parent class and will throw error in such case'''

'''if we think that while calling s1.showRollno() objects of parent class will

also be created then it is wrong. in c++ and java it use to happen in inheritance

that if the child class is called then the parent class's objects also use to

get called, but this doesn't happens in case of python, while calling the

child class it creates only the object of that class which is rollno. in python

it actually **OVERRIDES** the parent class if same functions with same number of

parameter is given, however if there is difference in parameters then it is

called as **HIDING**.'''

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/70 Inhertance.py

Roll Number: 100

Name: Rahul

Age: 16

We can also write the above program in different way as:

class Person1:

def \_\_init\_\_(self,n,a):

self.name=n

self.age=a

def showName(self):

print("Name:",self.name)

def showAge(self):

print("Age",self.age)

class Student1(Person1):

def \_\_init\_\_(self,r,n,a):

super().\_\_init\_\_(n,a)

self.rollno=r

def showRollno(self):

print("Roll number",self.rollno)

s2=Student1(100,"Rahul",15)

s2.showRollno()

s2.showName()

s2.showAge()

**Instance Member Variable Name Conflict:**

class Base:

def \_\_init\_\_(self):

self.a=10

def showBase(self):

print("Base a:",self.a)

class derived(Base):

def \_\_init\_\_(self):

self.a=20

super().\_\_init\_\_()

def showDerived(self):

print("Derived a:",self.a)

obj=derived()

obj.showBase()

obj.showDerived()

#in the output it will show both the value as 10

#because no new object a will be created and it will overwrite the derived value of a with base value of a

#what if we write the super line before self.a=20?

class Base:

def \_\_init\_\_(self):

self.a=10

def showBase(self):

print("Base a:",self.a)

class derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.a=20

def showDerived(self):

print("Derived a:",self.a)

obj=derived()

obj.showBase()

obj.showDerived()

#in these output will be 20 as a is created initially and then self.a is running

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/71 Inheritance Name conflict Instance member.py

Base a: 10

Derived a: 10

Base a: 20

Derived a: 20

>>>

So basically name conflict error doesn’t come when we deal with instance member variable.

**Static Member Variable Name Conflict:**

class Base:

x=11

def \_\_init\_\_(self):

self.a=10

Base.x=13

def showBase(self):

print("Base a:",self.a)

class derived(Base):

x=12

def \_\_init\_\_(self):

self.a=20

super().\_\_init\_\_()

def show():

print(Base.x,derived.x)

def showDerived(self):

print("Derived a:",self.a)

print(derived.x)

#output will be 12

#we need to understand that if two x variables are created!! so we will print it via dict

obj=derived()

print(derived.\_\_dict\_\_)

print(Base.x)

print(derived.x)

derived.show()

**#here unlike instancevariable 2 copies gets create one w.rt base and other w.r.t derived**

**#base class variable is accessed if called for base and child class variable is accessed if called for child.**

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/72 Inheritance Static Member Variable Name conflict.py

12

{'\_\_module\_\_': '\_\_main\_\_', 'x': 12, '\_\_init\_\_': <function derived.\_\_init\_\_ at 0x04005D18>, 'show': <function derived.show at 0x04005D60>, 'showDerived': <function derived.showDerived at 0x04005DA8>, '\_\_doc\_\_': None}

13

12

13 12

>>>

**Instance Member function Name conflict:**

**-Overriding**

**-Hiding**

Overriding : when the parent class function’s name and child class function’s name is same with equal number of arguments is known as overriding.

class Base:

def \_\_init\_\_(self):

self.a=10

def f1(self):

print("Base f1")

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.a=20

def f1(self):

print("Derived f1")

super().f1() #it will call the f1 of parent class

obj=Derived()

obj.f1() #here f1 is getting overridden

#here as child class object is called it will run f1 of child

#outout will be Derived f1 without super().f1()

#however if we put the super().f1 in child class function it will

#call the f1 of parent class, that is the process to do

#so now if we execute the program with super().f1() it will call base f1

#if we write the code without super().f1() it will call f1 of base

#even if we dont define f1 in derived class then also it will run the

#f1 of parent class because of inheritance.

**#so overriding is used when we want both parent and child class function to execute**

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/73 Instance member function name conflict overriding.py

Derived f1

Base f1

>>>

Now we will understand **hiding:**

**Hiding:** when function name of parent class and child class is same but number of arguments is different then it is called as Hiding.

class Base:

def \_\_init\_\_(self):

self.a=10

def f1(self):

print("Base f1")

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.a=20

def f1(self,x):

print("Derived f1")

super().f1() #it will call the f1 of parent class

obj=Derived()

obj.f1(3)

#as the object of derived class is called it will see the f1 in derived class only

#if the argument is passed as per the derived class f1 then it will execute both class

**#suppose if we dont pass argument which is 3 in the above program then**

**#it will throw error, it will not go to parent class and search f1 as the obj is of derived class**

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/74 Instance Member Function Name Conflict Hiding.py

Derived f1

Base f1

>>>

**suppose if we dont pass argument which is 3 in the above program then, it will throw error, it will not go to parent class and search f1 as the obj is of derived class, error is shown below :**

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/74 Instance Member Function Name Conflict Hiding.py

Traceback (most recent call last):

File "C:/Users/Dell/Desktop/python/Let-us-learn-python-master/74 Instance Member Function Name Conflict Hiding.py", line 14, in <module>

obj.f1()

TypeError: f1() missing 1 required positional argument: 'x'

>>>

**Static member function name conflict with same number of arguments:**

class Base:

def \_\_init\_\_(self):

self.a=10

@staticmethod

def f1():

print("Base f1")

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.a=20

@staticmethod

def f1():

print("Derived f1")

#super().f1()

Base.f1()

Derived.f1()

#it will show output as derived f1 without the line Base.f1()

#if we write super().f1() which i have commented then it will throw run time error because

#we know that if we do that it will implicitly pass the current object, but here f1

#is static and no such object is created. so if we want to call the parent class also then

#we will have to write Base.f1

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/75 Static Member function Name Conflict.py

Derived f1

Base f1

>>>

**Static member function name conflict with different number of arguments:**

#Now if we change the number of argument then it will work with the same principle of hiding

class Base:

def \_\_init\_\_(self):

self.a=10

@staticmethod

def f1(x):

print("Base f1")

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

self.a=20

@staticmethod

def f1():

print("Derived f1")

#super().f1()

Base.f1()

Derived.f1(3)

Output:

Traceback (most recent call last):

File "C:/Users/Dell/Desktop/python/Let-us-learn-python-master/75 Static Member function Name Conflict.py", line 40, in <module>

Derived.f1(3)

TypeError: f1() takes 0 positional arguments but 1 was given

>>>

**Name conflict between static and instance members:**

class Base:

x=10

def \_\_init\_\_(self):

self.x=20

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

obj=Derived()

print(obj.x,Derived.x)

#if we want both the values to be displayed then

#we need to access static variable by the class name and instance variable by object name

class Base:

x=10

def \_\_init\_\_(self):

self.a=20

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

obj=Derived()

print(obj.x)

#if the names are different then obj.x will take the static member value

class Base:

x=10

def \_\_init\_\_(self):

self.x=20

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

obj=Derived()

print(obj.x)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/76 Name conflict in intance member and static member.py

20 10

10

20

10

>>>

**When the name of instance variable and static variable is same and if we access the variable with object.x then it will be interpreted that it is calling the instance variable. If the instance variable name was not x then it will take the static variable and will treat it as instance variable**

**Accessing base class members in derived class (Inheritance)**

1. Instance member variable of Base class in derived class.
2. Static member variables of base class in derived class.
3. Instance member functions of base class in derived class.
4. Static member functions of Base class in derived class.

**Instance member variable of Base class in derived class.**

class Base:

def \_\_init\_\_(self):

self.a=10

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

def f1(self):

print(self.a)

obj=Derived()

print(obj.a)

#we can access the instance member variables of base class only by applying self.a in child class

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/78 Accessing Instance member variable of base class in derived class.py

10

>>>

**Static member variables of base class in derived class**

class Base:

x=11 #static member variable

def \_\_init\_\_(self):

self.a=10

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

@staticmethod

def f1():

print(x)

Derived.f1()

print(Derived.x)

#the above code will give error as if we cal f1 then it will search for the local variable x

#but we have not created x as a local variable, so to access the static variable,

#we will have to give the class name either Derived.x or Base.x as shown in the below program

class Base:

x=11 #static member variable

def \_\_init\_\_(self):

self.a=10

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

@staticmethod

def f1():

print(Base.x)

Derived.f1()

print(Derived.x)

Output will be 11.

**Instance member function of a Base class in Derived class:**

class Base:

def \_\_init\_\_(self):

self.a=10

def basefunction(self):

print("Base Instance function a=",self.a)

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

def f1(self):

self.basefunction()

obj=Derived()

obj.f1()

#in python if we want to access instace member in class body the we will have to

#access it by using self.functionname as shown in self.basefunction() in above code

#we cannot define static function in Derived class and access instance member function

#as instance itself is not there so we cannot the instance member fucntion.

#insted of self.basefunction() we can write super().basefunction() as in both the cases

#we are calling the base function

Output:

= RESTART: C:\Users\Dell\Desktop\python\Let-us-learn-python-master\80 Accessing Instance member function of a Base class in Derived class.py

Base Instance function a= 10

>>>

**Accessing static member function of base class in derived class:**

class Base:

x=11

def \_\_init\_\_(self):

self.a=10

@staticmethod

def basefunction():

print("Base Static function x=",Base.x)

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

def f1(self):

self.basefunction()

#derived.basefunction()

obj=Derived()

obj.f1()

#instance member function can call both static member and instance member.

#what if we define static function in derived class, lets see that in below code:

class Base:

x=11

def \_\_init\_\_(self):

self.a=10

@staticmethod

def basefunction():

print("Base Static function x=",Base.x)

class Derived(Base):

def \_\_init\_\_(self):

super().\_\_init\_\_()

@staticmethod

def f1():

Base.basefunction()

#derived.basefunction()

obj=Derived()

obj.f1()

#we cannot access the static member function of base class by applying super in derived class

#because super represents an object and there is no object as such

#we cannot use self also as that represents instance and there is no instance in it.

#we will use the class name, either base or derived

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/81 Accessing static member function of base class in derived class.py

Base Static function x= 11

Base Static function x= 11

>>>

**Multiple Inheritance:**

If a class is derived from more than one class, it is known as Multiple Inheritance.

Suppose there is an institute where student has the facility to teach if they come under bright student category and there is a class of person where name age will be there and the class student will have roll no, class teacher will have salary and age where as bright student will have points. So basically student and teacher class will inherit from person class and bright student will inherit from teacher and students.

Let us understand that by a program:

class Person:

def \_\_init\_\_(self,n,a):

self.name=n

self.age=a

class Student(Person):

def \_\_init\_\_(self,n,a,r):

self.rollno=r

Person.\_\_init\_\_(self,n,a)

class Teacher(Person):

def \_\_init\_\_(self,n,a,s,sub):

self.salary=s

self.subject=sub

Person.\_\_init\_\_(self,n,a)

class BrightStudent(Student,Teacher):

def \_\_init\_\_(self,n,a,r,s,sub,p):

self.points=p

Student.\_\_init\_\_(self,n,a,r)

Teacher.\_\_init\_\_(self,n,a,s,sub)

obj=BrightStudent("Rahul",18,100,2000,"Physics",7)

print(obj.\_\_dict\_\_)

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/82 Multiple Inheritance.py

{'points': 7, 'rollno': 100, 'name': 'Rahul', 'age': 18, 'salary': 2000, 'subject': 'Physics'}

>>>

**Data Hiding:**

In C. C++, JAVA we have seen access specifiers where private, protected and public keyword were used for data hiding, but in python how can we do data hiding!

Let’s say, we have made a class and assigned two static variable as x=10 and h=20, both are accessible from outside, however if we want to make the h hidden so that it is not accessible from outside then we will have to give two underscore before h as \_\_h

Let us understand that by coding:

class Test:

x=10 #static, visible from outside the class

\_\_h=20 #static, hidden variable

print(Test.x)

#print(Test.\_\_h)

#it will print x as 10 but will give error for \_\_h as it gets hidden because of the prefix.

#here what python actually does is, it change the hidden variable name internally

#it adds underscore classname in prefix as \_Test\_\_h

#however we can access the \_\_h inside the class itself as shown in below code:

#also we will try to access \_\_h outside the class by the new name which gets change internally

class Test:

x=10 #static, visible from outside the class

\_\_h=20 #static, hidden variable

@staticmethod

def f1():

print(Test.\_\_h)

Test.f1()

print("outside class \_\_h", Test.\_Test\_\_h)

#technically we can say that there is nothing as private variable in python as we can access it somehow

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/83 Data Hiding.py

10

20

outside class \_\_h 20

>>>

Let us see data hiding for instance variable:

############ Data Hiding for Insstance Variable #########

class Test:

x=10 #static, Visible from outside the class

\_\_h=20 #static, hidden variable

def \_\_init\_\_(self):

self.\_\_a=100 #private Instance variable

@staticmethod

def f1():

print(Test.\_\_h)

obj=Test()

print(obj.\_\_dict\_\_)

print(obj.\_Test\_\_a)

Output:

{'\_Test\_\_a': 100}

100

>>>

**#Benifit of using double underscore is that we can use it in base class and derived class**

**#when we call the variable the it will not get conflict as it will appl the prefx of respective class name**

**Polymorphism:**

Poly---🡪 Many

Morph--🡪Forms

Ex:

Take **Right** turn -🡪 Direction

Fight for your **Right -🡪** Fundamental Rights

Am I **Right --🡪 Right or wrong**

if we are asked what is right we can answer any one of them but it is clear when use it in sentence.

class A:

def f1(self):

print("A-f1")

class B:

def f1(self):

print("B-f1")

obj=A()

obj.f1()

obj=B()

obj.f1()

#as python is dynamically typed, it will run both the functions, in run time it resolves

#the object that which object is referring to which function.

#let us understand the polymorphic behaviour more specifically in the below pprogram

class Animal:

def \_\_init\_\_(self,name):

self.name=name

def talk(self): #this function's coding should not happen as per the Animal Class

raise NotImplementedError("Derived class must implement this function")

class Cat(Animal):

def talk(self):

return "Meow"

class Dog(Animal):

def talk(self):

return "woof"

animals=[cat("Rekha"),Cat("Soniya"),Dog("Moti"),Cat("Rupa"),Dog("Gabbar")]

for animal in animals:

print(animal.name,"-",animal.talk())

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/84 Polymorphism.py

A-f1

B-f1

Rekha - Meow

Soniya - Meow

Moti - woof

Rupa - Meow

Gabbar - woof

>>>

**Iterators and Generators:**

Iterators are basically pointer which points to some container (list, set, tuple or any sequences)

Whenever we access any sequences using for loop and print the contents then internally the code gets converted with the help of iterators as shown in below shell code:

>>> l=[1,2,3]

>>> type(l)

<class 'list'>

>>> li=iter(l)

>>> type(li)

<class 'list\_iterator'>

>>> for i in l:

print(i)

1

2

3

>>> for i in iter(l): #it converts the for I in l into for I in iter(l)

print(i)

1

2

3

>>> l

[1, 2, 3]

>>> li=iter(l)

>>> while True:

next(li)

1

2

3

Traceback (most recent call last):

File "<pyshell#19>", line 2, in <module>

next(li)

StopIteration

>>> '''next is a function through which we can access each element of container one by one'''

**Generators** are special function which generates iterators.

'''0 1 1 2 3 5 8 13 .....'''

#Fibonacci Series

def fibonacci(n):

a,b,c=0,1,0

while True:

if c>n:

return

yield a

a,b=b,a+b

c+=1

#yield follwed by value ('a'in this program) returns the nnext value and yield pauses the fuction

#and get resumed after next method is called and stops at yield only.

#when return is executed here it raises an exception whic is StopIteration.

#so, in this progrm insted of writing return we can use raise StopIteration

Output:

========================================== RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/85 Generators.py ==========================================

>>> it=fibonacci(10)

>>> next(it)

0

>>> next(it)

1

>>> next(it)

1

>>> next(it),next(it),next(it)

(2, 3, 5)

>>> it=fibonacci(10)

>>> for i in it:

print(i,end=' ')

0 1 1 2 3 5 8 13 21 34 55

>>>

#**Next doesn’t go back while accessing the elements of sequence, it always starts from beginning and ends at last.**

>>> it=fibonacci(10)

>>> while True:

print(next(it), end=' ')

0 1 1 2 3 5 8 13 21 34 55 Traceback (most recent call last):

File "<pyshell#38>", line 2, in <module>

print(next(it), end=' ')

StopIteration

>>>

**File Handling in python**

**Part 1:**

Whatever variables we make in program has its life till the program runs.

what if we want the variable to be saved even after the program stops running, this are the scenarios in real life projects.

Suppose we have deposited Rs 5000 in bank account, so there must be some variable in the back end which stores the value 5000. Now we withdraw some amount say like 2000 so the remaining amount in account is 3000 for which respective variables are made. In the backend a program is running which is storing those data. Now after three day also it should show the same data. Now if life cycle of a variable is till the program stops, however we want this variables to be permanently stored, this is where the file handling concept comes.

In basic file handling program we don’t need to import any module, in very specific case we will have to do that.

To store the variable we nee to store it in file.

There are two kinds of file:

1. Data files: where the data is a sequence of character. Character includes special symbols also.it is generally .txt,.py,.dat
2. Binary files: where data is basically sequence of bytes lie image, audio, video.

To understand the basics we will deal with Data files mostly. The must operations in file handling is **open (opening the file), Processing (Read or write operations in file), Close (Closing the file once processing is done).**

**Part 2:**

Open()

Syntax:

F=open(filename,mode)

We can give the file name in two ways:

1. If we give simply the file name then it will like file1.txt, it will look in the current directory where the python file is there.
2. We can give the absolute path like: d:\folder\folder2\file1.txt

We have 7 modes in file handling:

1. **‘r’**: read only mode, no other operations can be performed. If the file doesn’t exist in current directory then it will throw FileNotfoundError
2. **‘w’**: write only mode, it will not allow to read the file, so basically if the file exist then it will erase the previous content and will open the fresh file with no content. If the file doesn’t exist then it will create a new file and open it.
3. **‘a’**: append mode, it will open the file but not in read mode, however it will not erase the previous content of the file. It will allow us to write the content from where the data needs to be added and will append it with previous content. . If the file doesn’t exist then it will create a new file and open it.
4. **‘r+’**: reading and writing. It opens the file and pointer is pointed at the starting line or character. If we do read first and then write then it will not have any issues but if we write first then it will replace the character till the place we write, it happens so because the pointer is at the starting.it is best used for modifying the files. It will give error if file doesn’t exist because it is also in reading mode.
5. **‘w+’**: reading and writing. The difference with r+ is that when it opens it erase the previous data so in this mode it is writing first and then reading. It is not used for modification purpose. Second difference with r+ is that it creates the file if the given file name doesn’t exist.
6. **‘a+**’: reading and writing. **a+** Open **for** reading and appending (writing at end **of** file). The file **is** created if it does not exist.
7. **‘x’:** exclusive write mode. It is used for fresh writing only. It always creates a new file, even if the file exist, it will create a new file to open in write mode. It will throw error if the file name which we are creating in x should not match with already existing file. It is recommended if the error comes then we shall rename the existing files and then run the program.

**All the above mode is applicable only for Data File or the text mode file.**

If w want the operations to be performed in binary type of file then we will have to add suffix b at each mode like **rb, wb, ab, r+b, w+b, a+b, xb**

**Part 3:**

Before getting into it, one important thing to note is that we should close the file. If we don’t use it it might led to weird situations, it might effect to efficiency of the system if many files are opened and processing is going on.

Now we will see what kind of operations can be performed in file.

We can operate several functions once the file is opened. We will see that by python shell.

>>> f=open('file1.txt','r')

>>> f.name

'file1.txt'

>>> f.mode

'r'

>>> f.closed #this will return a Boolean value referring whether the file is closed or open.

False

>>> f.readable() #this will return a Boolean value referring whether the file is readable or not.

True

>>> f.writable() #this will return a Boolean value referring whether the file is writable or not.

False

>>>

**Part 4:**

f=open('file1.txt','w')

text=input("Enter some text: ")

f.write(text)

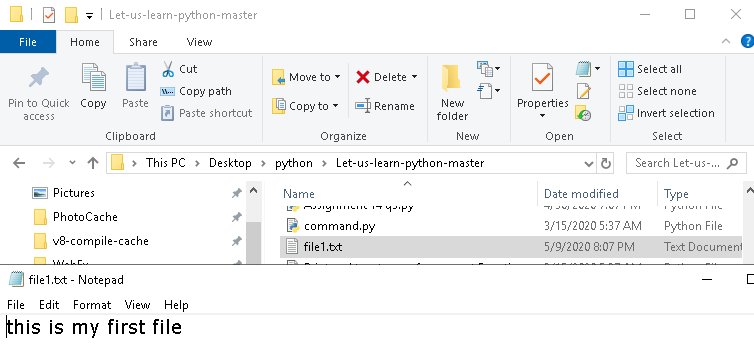
f.close()

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/86 File handling, write operations.py

Enter some text: this is my first file

>>>



**Writelines()**

This function is used to pass the string s shown below:

###writline function####

f=open('file2.txt','w') #it will create a text file named as file2

l=["Bhopal\n","Delhi\n","Indore"]

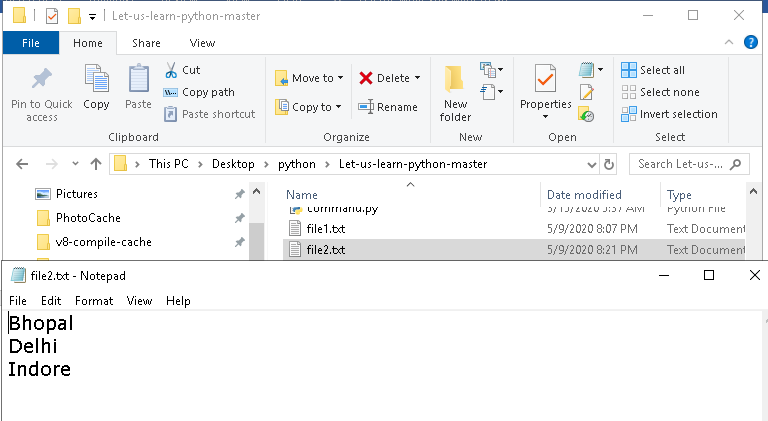
f.writelines(l) #this will print the contents of list

f.close()

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/86 File handling, write operations.py

>>>



Let us see some functions of reading operations:

read(): reads all data and returns in the form of string.

read(n): reads first n character.

readline(): reads one line, if called again then will read second line

readlines(): reads all the lines.

### Read operations ####

f=open('file2.txt','r')

#s0=f.read()

s1=f.read(5)

s2=f.read(5)

#print(s0)

print(s1)

print(s2)

f.close()

Output:  
= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/86 File handling, write operations.py

Bhopa

l

Del

>>>

If we give both read() and read(5) it will give the output as entire string not the character as per the parameter, however if we want the output as per the parameter then just give instructions as read(parameter). In the above program I have commented the statement #s0=f.read(). If we uncomment that then parameterized read statement will not get executed. Parameterized read operations will take space and next line also as a character.

## Read Operations without Read command ###

f=open('file2.txt','r')

for line in f:

print(line)

f.close

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/86 File handling, write operations.py

Bhopal

Delhi

Indore

>>>

## Readlline ##

f=open('file2.txt','r')

s1=f.readline()

print(s1)

s1=f.readline()

print(s1)

s1=f.readline()

print(s1)

s1=f.readline()

print(s1)

f.close()

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/86 File handling, write operations.py

Bhopal

Delhi

Indore

>>>

If we write readline(), cursor will point to the starting of first line and will print the first line, if we call the readline() again, it will print the next line. In the above program there are 3 lines, however if we give 4 readline() it will not throw any error. So there must be some mechanism by which we can understand how many time we need to call the readline() function or how many times the loop should run!! So there must be some way by which we can get the end of file.

W can do so by using while loop.

## loop to understand for end of file ##

f=open('file2.txt','r')

while True:

s1=f.readline()

if s1=='':

break

s1=f.readline()

print(s1)

print("out of while loop")

f.close()

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/86 File handling, write operations.py

Bhopal

Delhi

Indore

out of while loop

>>>

### readlines ###

f=open('file2.txt','r')

s=f.readlines()

print(s)

f.close()

output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/86 File handling, write operations.py

['Bhopal\n', 'Delhi\n', 'Indore']

>>>

**readlines() returns the entire string as a list whereas readline() return individual lines**.

### program to search the city name ###

f=open('file2.txt','r')

s=f.readlines()

x=input("Enter city name:")

x+='\n'

for e in s:

if e==x:

print("Matched city is: ",e)

else:

print("Match not found")

f.close()

Output:

= RESTART: C:/Users/Dell/Desktop/python/Let-us-learn-python-master/86 File handling, write operations.py

Enter city name:Bhopal

Matched city is: Bhopal

>>>

**Part 5:**

**Modify File content:**

Suppose we have a text file in which Indore is written. Now we want to change this content and write INDORE instead of it, so basically we want to override it.

We need to modify the file content with a file handling program in python as shown in next page. We will understand the program in two steps, first with a wrong program but correct logic and next with a correct program but logic applied in different way.

1st code:

Logic is we will have to open the file in reading and writing mode. So we will use the file open statement in r+ mode. As we need to change Indore to INDORE we will have to search for that so for that we will apply readline which will check line by line and then we will use while loop and apply condition to check the line having Indore. Nw when the Indore string matches, the cursor would have moved to next line . so to overwrite the content we will have to get the cursor back to the starting of Indore. To do so we will have to take the help of **seek function, it takes two arguments: 1st argument is offset and 2nd argument is Basis. Lets understand basis first: we can give basis as 0 or 1 or 2. 0 means “from the beginig of the file”, 1 means “from the current position of the cursor ”, 2 means “from the last character of the file”.** In this program we will set the cursor on the basis of current position. So in seek function second argument will be 1. Now the **1st argument denotes how many bytes we want to go back, so here we will put it equal to the string kept in s or x including \n, so the 1st argument will be as -len(s).**  Here we will check whether we can give negative value or not. Then after this we will over write it as f.write(“INDORE\n”).

So with this logic codes comes as :

f=open('file2.txt','r+')

x="Indore\n"

s=f.readline()

while True:

if s==x:

f.seek(-len(s),1)

f.write("INDORE\n")

break

s=f.readline()

f.close()

**The abov code will throw error as:**

=================== RESTART: C:\Users\Dell\Desktop\sample.py ===================

Traceback (most recent call last):

File "C:\Users\Dell\Desktop\sample.py", line 23, in <module>

f.seek(-len(s),1)

io.UnsupportedOperation: can't do nonzero cur-relative seeks

>>>

**So we will have to think of logic in the different way so that we do not need to give -ve value.**

in such case we will check the length from beginning, so we will initialize the length of string as 0 and in condition we will increment the length of string each time. So once we get the total length we will subtract it with the length of Indore\n once matched and the overwrite it. So the code will look like:

f=open('file3.txt','r+')

x="Indore\n"

s=f.readline()

l=0

while True:

if s=='':

break

l+=len(s)

if s==x:

f.seek(l-len(s),0)

f.write("INDORE")

break

s=f.readline()

f.close()

Output: content of the file will be changed.

**Part 6:**

**Read write object in a file:**

**How to write object in a file?**

to perform object by object data we need to import **pickle** module and then access few of its function.

In our program we will use **dump** function from **pickle** which takes 2 arguments, 1st will be what we want to write in a file and the 2nd we need to give file pointer. So this dump function will be repeatedly called and the object will be written in the file. We will also define function to view the data in console. To do so we will have to use the **load** function of pickle which will load the data but while doing so it might throw exception which will come because there will be a situation when there will be no data to be loaded and then it will throw EOFError (End of file error) which we need to handle.

class Student:

def \_\_init\_\_(self,name,age):

self.name=name

self.age=age

students\_list=[Student('Amit',19),Student('Rahul',20),Student('Romesh',18),Student('Ajay',19)]

def saveStudents():

import pickle

f1=open('file3.obj','wb')

for s in students\_list:

pickle.dump(s,f1)

f1.close()

def viewAllStudents():

imort pickle

f2=open('file3.obj','rb')

s\_list=[]

while True:

try:

s\_list+=[pickle.load(f2)]

except EOFError:

break

return s\_list

Output:

=================== RESTART: C:\Users\Dell\Desktop\sample.py ===================

>>> saveStudents()

>>> l=viewAllStudents()

>>> for e in l:

print(e.name,'...',e.age)

Amit ... 19

Rahul ... 20

Romesh ... 18

Ajay ... 19

>>>

**Revision: Important exceptions and other concepts**

**LIST**

V.I

2. What will be the output of the following Python code?

1. names1 = ['Amir', 'Bear', 'Charlton', 'Daman']
2. names2 = names1
3. names3 = names1[:]
5. names2[0] = 'Alice'
6. names3[1] = 'Bob'
8. sum = 0
9. for ls in (names1, names2, names3):
10. if ls[0] == 'Alice':
11. sum += 1
12. if ls[1] == 'Bob':
13. sum += 10
15. print sum

a) 11  
b) 12  
c) 21  
d) 22  
View Answer

Answer: b  
**Explanation:** When assigning names1 to names2, we create a second reference to the same list. Changes to names2 affect names1**. When assigning the slice of all elements in names1 to names3, we are creating a full copy of names1 which can be modified independently.**

What will be the output of the following Python code?

1. values = [[3, 4, 5, 1], [33, 6, 1, 2]]
3. v = values[0][0]
4. for row in range(0, len(values)):
5. for column in range(0, len(values[row])):
6. if v < values[row][column]:
7. v = values[row][column]
9. print(v)

a) 3  
b) 5  
c) 6  
d) 33  
View Answer

Answer: d  
Explanation: Execute in the shell to verify.

>>> list1=[1,2,3,4]

>>> list2=list(list1)

>>> list2.append('a')

>>> list1

[1, 2, 3, 4]

>>> list2

[1, 2, 3, 4, 'a']

>>> # above is creating shallow copy by using buitin function

>>> list3=[1,2,3,4]

>>> list4=list1[:]

>>> list3

[1, 2, 3, 4]

>>> list4.append(5)

>>> list4

[1, 2, 3, 4, 5]

>>> list3

[1, 2, 3, 4]

>>> #above is creating shallow copy by slicing operator.

>>> list6=[1,2,3,4]

>>> list7=[x for x in list1]

>>> list2

[1, 2, 3, 4, 'a']

>>> list7

[1, 2, 3, 4]

>>> list7[3]='a'

>>> list7

[1, 2, 3, 'a']

>>> list6

[1, 2, 3, 4]

>>> #above is creating shallow copy by list comprehension.

")

>>>

>>> import copy

>>> list8=[5,6,7,8]

>>> list9=copy.copy(list1)

>>> list9.append("hello")

>>> list9

[1, 2, 3, 4, 'hello']

>>> list8

[5, 6, 7, 8]

>>> #above method is creating shallow copy by copy module.

>>>

>>> #the above methods doesnot work in nested list, it effects both list so

>>> #we need to understand the deep copy concept.

>>> #first we will see how shallow copy fails in nested list in the below example

>>> import copy

>>> list20=[[1,2],3,4]

>>> list21=copy.copy(list20)

>>> list21

[[1, 2], 3, 4]

>>> list20

[[1, 2], 3, 4]

>>> list21[0][1]="A"

>>> list21

[[1, 'A'], 3, 4]

>>> list20

[[1, 'A'], 3, 4]

>>>

>>> #deepcopy

>>> '''creates a new object and recursively adds the copies of nested objects present n original elements'''

'creates a new object and recursively adds the copies of nested objects present n original elements'

>>> import copy

>>> list1=[1,2,3,4]

>>> lis2=copy.deepcopy(list1)

>>> list2

Traceback (most recent call last):

File "<pyshell#5>", line 1, in <module>

list2

NameError: name 'list2' is not defined

>>> lis2

[1, 2, 3, 4]

>>> lis2.append(10)

>>>

>>> lis2

[1, 2, 3, 4, 10]

>>> list1

[1, 2, 3, 4]

>>> import copy

>>> list1=[[1,2],3,4]

>>> list2=copy.deepcopy(list1)

>>>

>>> list2

[[1, 2], 3, 4]

>>> list1

[[1, 2], 3, 4]

>>> list2[0][0]="gourav"

>>> list2

[['gourav', 2], 3, 4]

>>> list1

[[1, 2], 3, 4]

>>>