DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

PYTHON PROGRAMMING

B.Tech I Year II Sem

Dr Ayesha Banu

Assistant Professor

Department of CSE

Unit – I

Introduction to Python: What is Python?, What is Python Good For?, Python History, How does Python Execute a Program, Review of a Simple Program, Some of the Basic Commands, Variables, Statements, Input/Output Operations, Keywords, Variables, Assigning values, Standard Data Types, Strings, Operands and operators.

Unit – II

Understanding the Decision Control Structures: The if Statement, A Word on Indentation, The if … else Statement, The if … elif … else Statement,

Loop Control Statements: The while Loop, The for Loop, Infinite Loops, Nested Loops.

The break Statement, The continue Statement, The pass Statement, The assert Statement, The return Statement.

Unit – III

Functions- Function Definition and Execution, Scoping, Arguments: Arguments are Objects, Argument Calling by Keywords, Default Arguments, Function Rules, Return Values.

Advanced Function Calling: The apply Statement, The map Statement, Indirect Function Calls.

Unit - IV

Lists: List, Creating List, Updating the Elements of a List, Sorting the List Elements. Storing Different Types of Data in a List, Nested Lists, Nested Lists as Matrices.

Tuples: Creating Tuple, Accessing the Tuple Elements, Basic Operations on Tuples, Functions to Process Tuples, Inserting Elements in a Tuple, Modifying Elements of a Tuple, Deleting Elements from a Tuple.

Sets: Creating Set, Basic Operations on Sets, Methods of Set.

Dictionaries: Operations on Dictionaries, Dictionary Methods, Using for Loop with Dictionaries, Sorting the Elements of a Dictionary using Lambdas, Converting Lists into Dictionary.

Unit – V

Modules: Importing a Module, Tricks for Importing Modules, Packages.

Exceptions and Error Trapping: What is an Exception?, Exception Handling: try..except..else.., try..finally.., Exceptions Nest, Raising Exceptions, Built-In Exceptions.

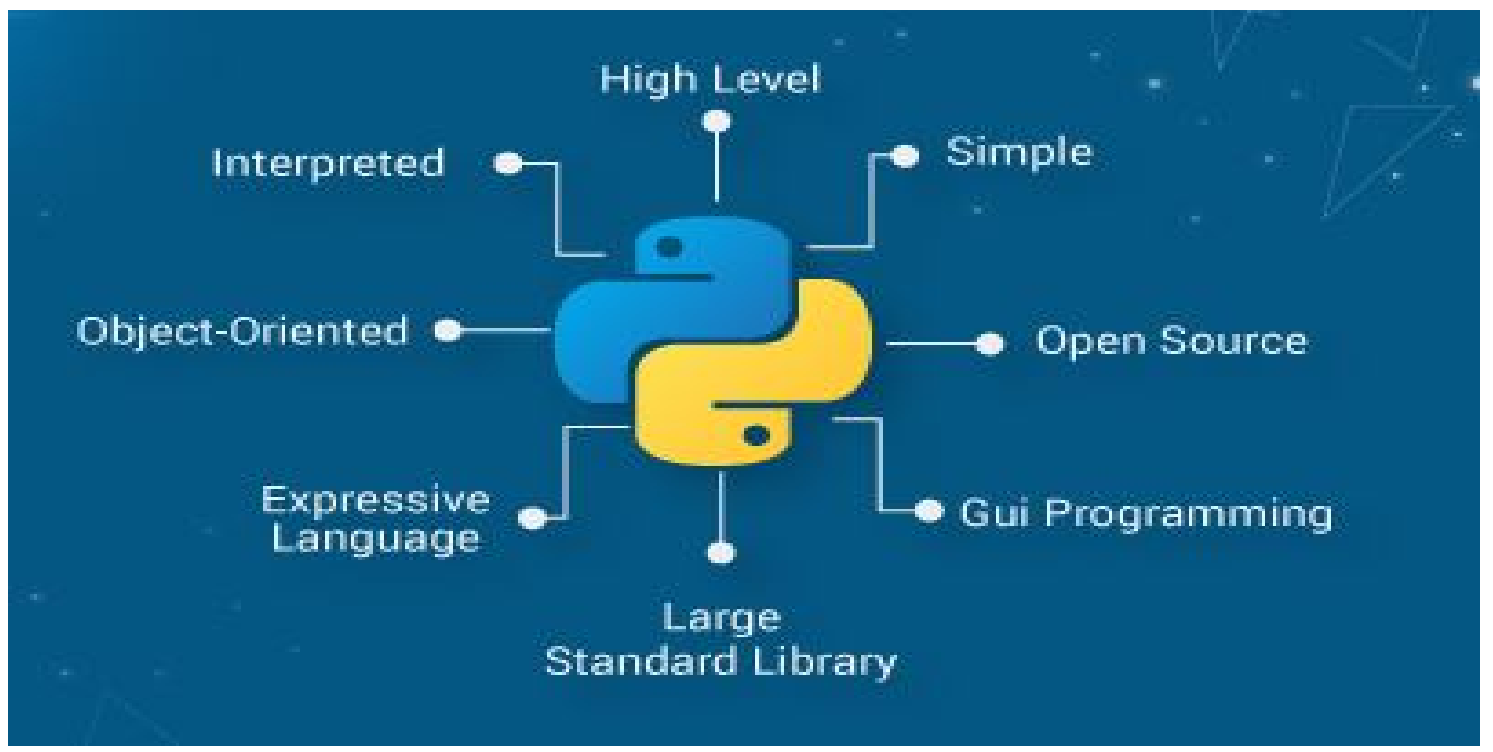
UNIT – I

Topic 1: Introduction to Python: What is Python ???

- Python has become the most popular programming language widely being used today.
- People are shifting to python nowadays from other programming languages like c++, java, c# (c sharp).
- Why people are showing interest towards python ??

Some Features of Python which make it popular

Python Logo



1. Simple: Easy to write compared to other languages. Python is a very developer-friendly language which means that anyone and everyone can learn to code it in a couple of hours or days. As compared to other object-oriented programming languages like Java, C, C++, and C#, Python is one of the easiest to learn.

2. Open Source: it is freely downloadable and all executable versions of python are easily and openly available on internet. Python is an open-source programming language which means that anyone can create and contribute to its development. Python has an online forum where thousands of coders gather daily to improve this language further. Along with this Python is free to download and use in any operating system, be it Windows, Mac or Linux.
3. GUI: Supports Graphical User Interface programming. GUI is one of the key aspects of any programming language because it has the ability to code and make the results more visual.
4. Object-Oriented: One of the key aspects of Python is its object-oriented approach. This basically means that Python recognizes the concept of class and object encapsulation thus allowing programs to be efficient in the long run.

5.High-Level Language: Python has been designed to be a high-level programming language, which means that when you code in Python you don’t need to be aware of the coding structure, architecture as well as memory management.

6. Interpreted: This means that the python interpreter executes codes one line at a time. whereas the compiler executes the code entirely and lists all possible errors at a time. That’s why python shows only one error message at a time. This will help you to clear errors easily.

7. Large Standard Library: Out of the box, Python comes inbuilt with a large number of standard libraries that can be imported at any instance and be used in a specific program. The presence of libraries also makes sure that you don’t need to write all the code yourself and can import the same from those that already exist in the libraries.

Apart from these features python is

- Highly Portable: Suppose you are running Python on Windows and you need to shift the same to either a Mac or a Linux system, then you can easily achieve the same in Python without having to worry about changing the code. This is not possible in other programming languages, thus making Python one of the most portable languages available in the industry.
- Dynamically-Typed-Language: Python is a dynamically-typed language. That means the type (for example- int, double, long, etc.) for a variable is decided at run time not in advance because of this feature we don't need to specify the type of variable.
- Expressive Language: Python can perform complex tasks using a few lines of code.

C	Python
Variables are declared in C.	Python has no declaration.
C doesn’t have native OOP.	Python has OOP which is a part of language.
Pointers are available in C language.	No pointers functionality is available in Python.
C is a compiled language.	Python is an interpreted language.
There is a limited number of built-in functions available in C.	There is a large library of built-in functions in Python.
Declaring of variable type in C is necessary condition.	There is no need to declare a type of variable in Python.
C does not have complex data structures.	Python has some complex data structures.

C is statically typed.	Python is dynamically typed.
Syntax of C is harder than python because of which programmers prefer to use python instead of C	It is easy to learn, write and read Python programs than C.
C programs are saved with .c extension.	Python programs are saved by .py extension.
In C language testing and debugging is harder.	In Python, testing and debugging is not harder than C.
C is complex than Python.	Python is much easier than C.

Therefore Python is a Open Source , Portable, Powerful, Extensible and Easy Programming Language

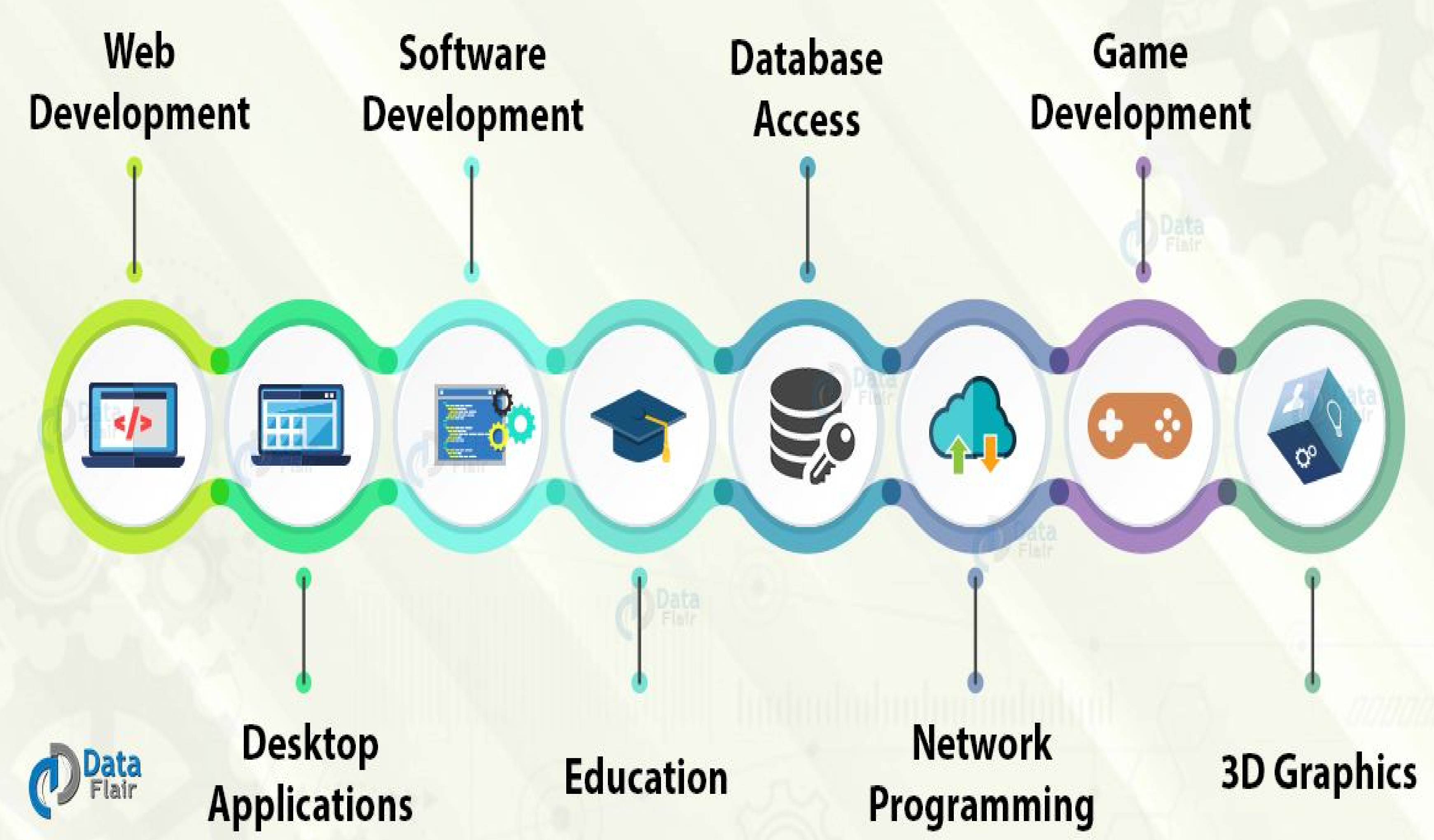
Topic-2: What is Python Good for ?? Applications of Python

With its wide support and extensive features Python is very good and effective for large number of applications and tasks including:

- Mathematics: Python is a versatile language that has various applications in the field of mathematics with its robust mathematical libraries. There are several libraries that can be used to carry out mathematical operations with Python.

1. **Math:** This is the most basic math module that is available in Python. It covers basic mathematical operations like sum, exponential, modulus, etc.
2. **Numpy:** The numpy library in Python is most widely used for carrying out mathematical operations that involve matrices.
3. **SciPy:** This python math library provides all the scientific tools for Python. It contains various models for mathematical optimization, linear algebra, Fourier Transforms, etc.

- Text Processing: Python Programming can be used to process text data for the requirements in various textual data analysis. A very important area of application of such text processing ability of python is for NLP (Natural Language Processing). NLP is used in search engines, newspaper feed analysis and more recently for voice -based applications like Siri and Alexa. Python's Natural Language Toolkit (NLTK) is a group of libraries that can be used for creating such Text Processing systems.
- Rapid Application Development: python can be effectively used for developing any application quickly is less amount of time. Python can also be applied for cross platform dvelopment.


Python Applications
Web Development
Software Development
Database Access
Game Development
Desktop Applications
Education
Network Programming
3D Graphics
Data Flair

Topic -3: History of Python Programming Language

- Python was created by **Guido van Rossum**, a Dutch programmer.
- He was also known as the "Benevolent dictator for life" (BDFL) for python.
- He Worked at the Centrum Wiskunde & Informatica (CWI) in the Netherlands.

- there was a popular BBC comedy TV show called “Monty Python’s Flying Circus” and Van Rossum happened to be a big fan of that show.
- At the time when he began implementing Python, he needed a name that was short, unique, and slightly mysterious, so he decided to call the language “Python.”
- The first ever version of Python (i.e., Python 1.0) was introduced in 1991.
- Python 2.0 was released on 16 October 2000 and had many major new features.
- Python 3.0 (initially called Python 3000 or py3k) was released on 3 December 2008 after a long testing period.
- The language’s core philosophy is summarized in the document “The Zen of Python”.

Thrust Areas of Python

1. Academia: Python is being offered as the introductory programming language in the majority of the computer science departments at various American universities. Python is being adapted by academia for research purposes.
2. Scientific Tools: Scientific tools are essential for simulating and analyzing complex systems. The Python ecosystem consists of these core scientific packages, namely SciPy library, NumPy, Jupyter, Sympy and Matplotlib. Most of these tools are available under Berkeley Software Distribution (BSD) license and can be used without any restrictions.
3. Machine Learning: Many machine-learning algorithms and techniques have been developed that allow computers to learn. Machine Learning has its origin in Computer Science and Statistics. Scikit- Learn is a well-known Machine Learning tool built on top of other Python scientific tools like NumPy, SciPy and Matplotlib.

4. Data Analysis: Pandas library changed the landscape of data analysis in Python. Pandas is built on top of NumPy and has two important data structures, namely Series and DataFrame. Pandas can be used to read Comma-Separated Values (CSV) files, Microsoft Excel, Structured Query Language (SQL) database and Hierarchical Data Format (HDF5) format files.
5. Statistics: Statsmodels is a Python library used for statistical analysis. It supports various models and features like linear aggression models, generalized linear models, discrete choice models and functions for time series analysis.
6. Cloud Computing: OpenStack is entirely written in Python and is used to create a scalable private and public cloud.

4: Execute Python Program: Python Distributions and IDE

- **IDLE** (short for Integrated Development and Learning Environment)is an integrated development environment for Python. Every Python installation comes with an IDLE or even IDE. These are a class of applications that help you write code more efficiently. While there are many IDEs for you to choose from, Python IDLE is very bare-bones, which makes it the perfect tool for a beginning programmer.

  https://www.python.org/

- **Online python compilers:**

- **Google colab:** Colab is a Python development environment that runs in the browser using Google Cloud.

- **Anaconda** is a distribution of the Python and R programming languages for scientific computing (data science, machine learning applications, large-scale data processing, predictive analytics, etc.). The distribution includes data-science packages suitable for Windows, Linux, and macOS. It is developed and maintained by Anaconda, Inc., which was founded by Peter Wang and Travis Oliphant in 2012.

- **PyCharm** is a dedicated Python Integrated Development Environment (IDE) providing a wide range of essential tools for Python developers, tightly integrated to create a convenient environment for productive Python, web, and data science development.

- **Jupyter Notebook** is an environment that we can use to experiment with Python interactively . It allows you to share live Python code with others .

Assignment 1:

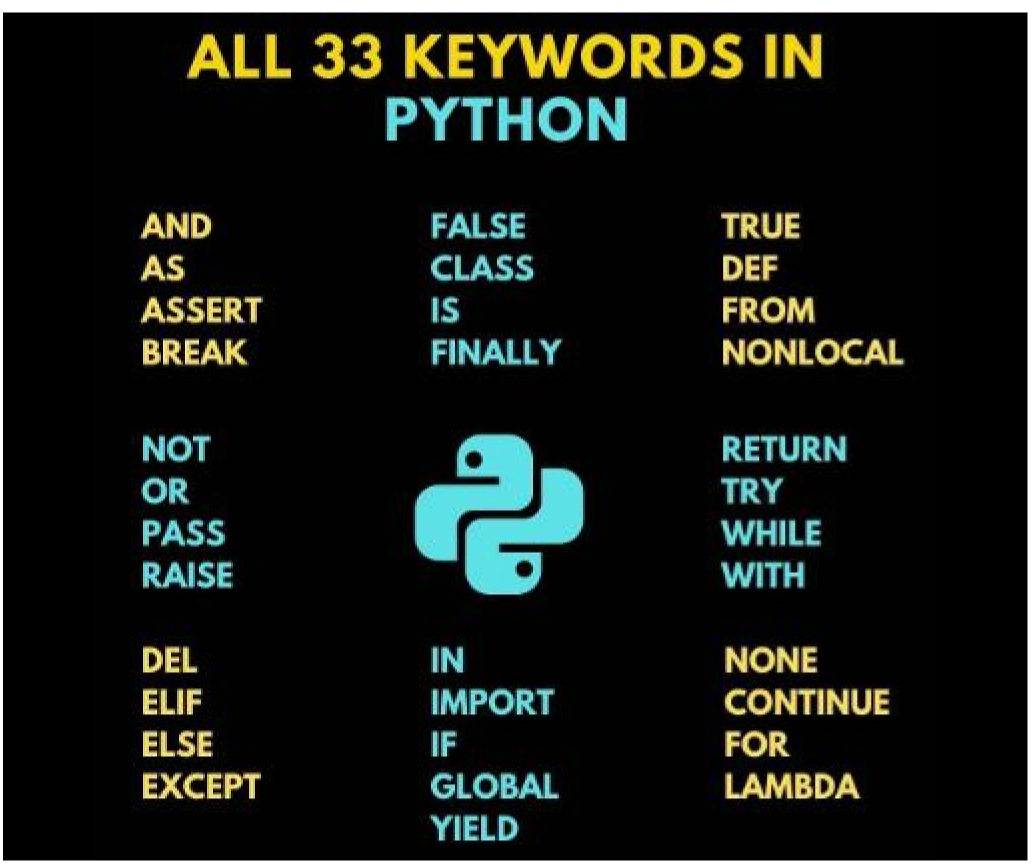
1. Write or explain the different features of python which makes this language easy and popular?
2. Explain with examples where ever required the differences between C and Python programming languages ?
3. Explain briefly the history and major applications of python ?

Identifiers

- An identifier is a name given to a variable, function, class or module. Identifiers may be one or more characters.
- Identifiers can be a combination of letters in lowercase (a to z) or uppercase (A to Z) or digits (0 to 9) or an underscore (_).
- A Python identifier can begin with an alphabet (A – Z and a – z and _) but an identifier cannot start with a digit.
- Keywords cannot be used as identifiers.
- We cannot use spaces and special symbols like !, @, #, $, % etc. as identifiers.
- Identifier can be of any length.

Keywords

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



Variables

- Variable is a named placeholder to hold any type of data which the program can use to assign and modify during the course of execution.
- In Python, there is no need to declare a variable explicitly by specifying whether the variable is an integer or a float or any other type.
- *To define a new variable in Python, we simply assign a value to a name.*
- Variable names can consist of any number of letters, underscores and digits.
- Variable should not start with a number.
- Python Keywords are not allowed as variable names.
- Variable names are case-sensitive.

Assigning Values to Variables

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

  variable_name = expression

- The equal sign (=) also known as simple assignment operator is used to assign values to variables.

Example:

```
x =100
y = 13.4
name ="Python“
```

We do not declare the data type for any variable. Python automatically gives the data type to the variables.

① integer type value is assigned to a variable x

② float type value has been assigned to variable y and

③ string type value is assigned to variable name.

To display or print the assigned variable values we write

```
print(x)
print(y)
print(name)
```

OUTPUT

```
100
13.4
python
```

```
#include <stdio.h>
int main()
 {
int x=100;
printf("%d\n",x);
float y=13.4;
printf("%f\n",y);
char name[10]="python";
 printf("%s\n",name);
 return 0;
}
```

- In Python, not only the value of a variable may change during program execution but also the type of data that is assigned.
- You can assign an integer value to a variable, use it as an integer for a while and then assign a string to the variable.
- A new assignment overrides any previous assignments.

```
x =100
print(x)
x=15.6
print(x)
x="hai"
print(x)
```

OUTPUT

```
100
15.6
hai
```

```
2  #include <stdio.h>
3  int main() {
4      // Write C code here
5      int x=100;
6      float x=15.6;
7      printf("%f",x);
8      return 0;
9  }
```

```
gcc /tmp/zCIQwRCTAY.c -lm
/tmp/zCIQwRCTAY.c: In function 'main':
/tmp/zCIQwRCTAY.c:12:11: error: conflicting types for 'x'
   12 |     float x=15.6;
      |           ^
/tmp/zCIQwRCTAY.c:6:9: note: previous definition of 'x' was here
    6 |     int x=100;
      |         ^
```

Python allows us to assign a single value to several variables simultaneously.

```
a = b = c =1
print(a)
print(b)
print(c)
```

```
1  // Online C compiler to run C program online
2  #include <stdio.h>
3  int main() {
4      // Write C code here
5      int x=y=z=100;
6      printf("%d",x);
7      printf("%d",y);
8      printf("%d",z);
9      return 0;
10 }
```

```
a, b,c=10,20,30
a=10,b=20,c=30
```

```
gcc /tmp/zCIQwRCTAY.c -lm
/tmp/zCIQwRCTAY.c: In function 'main':
/tmp/zCIQwRCTAY.c:5:11: error: 'y' undeclared (first use in this function)
    5 |     int x=y=z=100;
      |           ^
/tmp/zCIQwRCTAY.c:5:11: note: each undeclared identifier is reported only
    once for each function it appears in
/tmp/zCIQwRCTAY.c:5:13: error: 'z' undeclared (first use in this function)
    5 |     int x=y=z=100;
      |             ^
```

Operators

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

1. Arithmetic Operators
2. Assignment Operators
3. Comparison Operators
4. Logical Operators
5. Bitwise Operators

1. Arithmetic Operators:

Arithmetic operators are used to execute arithmetic operations such as addition, subtraction, division, multiplication etc. The following TABLE shows all the arithmetic operators supported by python.

Operator	Operator Name	Description	Example
+	Addition operator	Adds two operands, producing their sum.	p + q = 5
–	Subtraction operator	Subtracts the two operands, producing their difference.	p – q = –1
*	Multiplication operator	Produces the product of the operands.	p * q = 6
/	Division operator	Produces the quotient of its operands where the left operand is the dividend and the right operand is the divisor.	q / p = 1.5
%	Modulus operator	Divides left hand operand by right hand operand and returns a remainder.	q % p = 1
**	Exponent operator	Performs exponential (power) calculation on operators.	p**q = 8
//	Floor division operator	Returns the integral part of the quotient.	9//2 = 4 and 9.0//2.0 = 4.0

Note: The value of p is 2 and q is 3.

2. Assignment Operators

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

```
x = 5
x = x + 1
print(x)
```

6

- Compound assignment operators support shorthand notation for avoiding the repetition of the left-side variable on the right side.
- For example, the statement

  x = x + 1

- can be written x += 1

List of Assignment Operators

Operator	Operator Name	Description	Example
=	Assignment	Assigns values from right side operands to left side operand.	z = p + q assigns value of p + q to z
+=	Addition Assignment	Adds the value of right operand to the left operand and assigns the result to left operand.	z += p is equivalent to z = z + p
−=	Subtraction Assignment	Subtracts the value of right operand from the left operand and assigns the result to left operand.	z −= p is equivalent to z = z − p
*=	Multiplication Assignment	Multiplies the value of right operand with the left operand and assigns the result to left operand.	z *= p is equivalent to z = z * p
/=	Division Assignment	Divides the value of right operand with the left operand and assigns the result to left operand.	z /= p is equivalent to z = z / p
**=	Exponentiation Assignment	Evaluates to the result of raising the first operand to the power of the second operand.	z**= p is equivalent to z = z ** p
//=	Floor Division Assignment	Produces the integral part of the quotient of its operands where the left operand is the dividend and the right operand is the divisor.	z //= p is equivalent to z = z // p
%=	Remainder Assignment	Computes the remainder after division and assigns the value to the left operand.	z %= p is equivalent to z = z % p

```
3   p=10
4   q=12
5   q += p
6   print(q)
7   q*=p
8   print(q)
9   q/=p
10  print(q)
11  q%=p
12  print(q)
13  q**=p
14  print(q)
15  q//=p
16  print(q)
```

```
22
220
22.0
2.0
1024.0
102.0
>
```

3. Comparison Operators

- When the values of two operands are to be compared then comparison operators are used.
- The output of these comparison operators is always a Boolean value, either True or False.
- The operands can be Numbers or Strings or Boolean values.

List of Comparison Operators

Operator	Operator Name	Description	Example
==	Equal to	If the values of two operands are equal, then the condition becomes True.	(p == q) is not True.
!=	Not Equal to	If values of two operands are not equal, then the condition becomes True.	(p != q) is True
>	Greater than	If the value of left operand is greater than the value of right operand, then condition becomes True.	(p > q) is not True.
<	Lesser than	If the value of left operand is less than the value of right operand, then condition becomes True.	(p < q) is True.
>=	Greater than or equal to	If the value of left operand is greater than or equal to the value of right operand, then condition becomes True.	(p >= q) is not True.
<=	Lesser than or equal to	If the value of left operand is less than or equal to the value of right operand, then condition becomes True.	(p <= q) is True.

Note: The value of p is 10 and q is 20.

Activate Windows

```
1  # Online Python compiler (interpreter) to run Python online.
2  # Write Python 3 code in this online editor and run it.
3  print(10==20)
4  print(12>20)
5  print(2<10)
6  print(10!=10)
7  print("p">"a")
8  print("ayesha">"rama")
9
10
```

```
False
False
True
False
True
False
>
```

4. Logical Operators

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

List of Logical Operators

Operator	Operator Name	Description	Example
and	Logical AND	Performs AND operation and the result is True when both operands are True	p and q results in False
or	Logical OR	Performs OR operation and the result is True when any one of both operand is True	p or q results in True
not	Logical NOT	Reverses the operand state	not p results in False

Note: The Boolean value of p is True and q is False.

Boolean Logic Truth Table

P	Q	P and Q	P or Q	Not P
True	True	True	True	False
True	False	False	True	
False	True	False	True	True
False	False	False	False	

```
1 # Online Python compiler (interpreter) to run Python online.
2 # Write Python 3 code in this online editor and run it.
3 print(True and False)
4 print(True or False)
5 print((10 < 0) and (10 > 2))
6 print((10 < 0) or (10 > 2))
7 print(not(1 > 2 and 9 > 6))
```

```
False
True
False
True
True
>
```

5. Bitwise Operators

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

List of Bitwise Operators

<table>
<tr><th>Operator</th><th>Operator Name</th><th>Description</th><th>Example</th></tr>
<tr><td>&</td><td>Binary AND</td><td>Result is one in each bit position for which the corresponding bits of both operands are 1s.</td><td>p & q = 12 (means 0000 1100)</td></tr>
<tr><td>|</td><td>Binary OR</td><td>Result is one in each bit position for which the corresponding bits of either or both operands are 1s.</td><td>p | q = 61 (means 0011 1101)</td></tr>
<tr><td>^</td><td>Binary XOR</td><td>Result is one in each bit position for which the corresponding bits of either but not both operands are 1s.</td><td>(p ^ q) = 49 (means 0011 0001)</td></tr>
<tr><td>~</td><td>Binary Ones Complement</td><td>Inverts the bits of its operand.</td><td>(~p) = –61 (means 1100 0011 in 2's complement form due to a signed binary number.</td></tr>
<tr><td><<</td><td>Binary Left Shift</td><td>The left operands value is moved left by the number of bits specified by the right operand.</td><td>p << 2 = 240 (means 1111 0000)</td></tr>
<tr><td>>></td><td>Binary Right Shift</td><td>The left operands value is moved right by the number of bits specified by the right operand.</td><td>p >> 2 = 15 (means 0000 1111)</td></tr>
</table>

Note: The value of p is 60 and q is 13.

Activate Windows
Go to PC settings to activate

Bitwise Truth Table

P	Q	P & Q	P | Q	P ^ Q	~ P
0	0	0	0	0	1
0	1	0	1	1	
1	0	0	1	1	0
1	1	1	1	0	

```
1  # Online Python compiler (interpreter) to run Python online.
2  # Write Python 3 code in this online editor and run it.
3  p=60
4  q=13
5  print(p&q)
6  print(p|q)
7  print(p^q)
8  print(~p)
9  print(p<<2)
10 print(p>>2)
```

```
12
61
49
-61
240
15
>
```

Bitwise and (&)				
a	=	0011 1100	→	(60)
b	=	0000 1101	→	(13)
a & b	=	0000 1100	→	(12)

Bitwise or (|)				
a	=	0011 1100	→	(60)
b	=	0000 1101	→	(13)
a | b	=	0011 1101	→	(61)

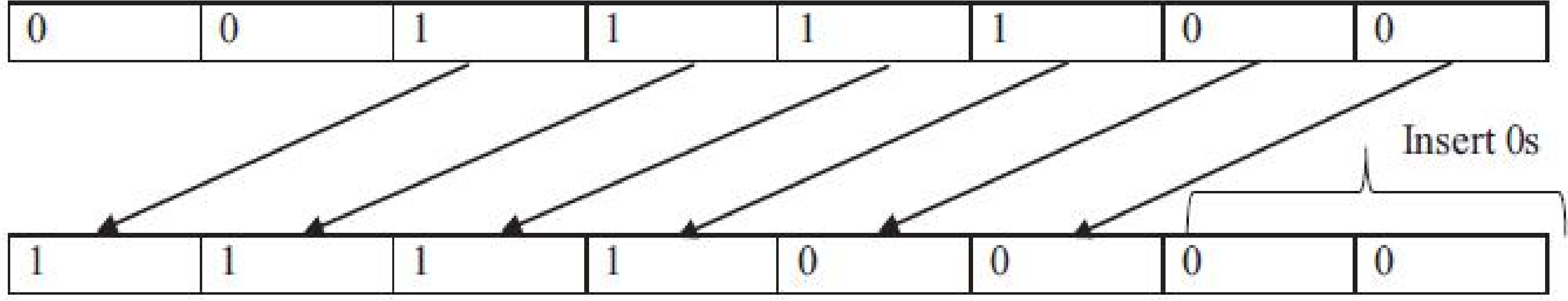
Bitwise exclusive or (^)				
a	=	0011 1100	→	(60)
b	=	0000 1101	→	(13)
a ^ b	=	0011 0001	→	(49)

One's Complement (~)				
a	=	0011 1100	→	(60)
~ a	=	1100 0011	→	(-61)

Binary left shift (<<)				
a	=	0011 1100	→	(60)
a << 2	=	1111 0000	→	(240)
left shift of 2 bits				

Binary right shift (>>)				
a	=	0011 1100	→	(60)
a >> 2	=	0000 1111	→	(15)
right shift of 2 bits				

E 2.2 shows how the expression 60 << 2 would be evaluated in a byte.



Precedence and Associativity

- Operator precedence determines the way in which operators are parsed with respect to each other.
- Associativity determines the way in which operators of the same precedence are parsed.

Operator Precedence in Python

Operators	Meaning
()	Parentheses
**	Exponent
+x, −x, ~x	Unary plus, Unary minus, Bitwise NOT
*, /, //, %	Multiplication, Division, Floor division, Modulus
+, −	Addition, Subtraction
<<, >>	Bitwise shift operators
&	Bitwise AND
^	Bitwise XOR
|	Bitwise OR
==, !=, >, >=, <, <=, is, is not, in, not in	Comparisons, Identity, Membership operators
not	Logical NOT
and	Logical AND
or	Logical OR

Data Types

Data types specify the type of data like numbers and characters to be stored and manipulated within a program.

- Numeric Types: int, float, complex
- Boolean Type: bool
- Text Type: str
- Special data type: None
- Sequence Types: list, tuple, range
- Mapping Type: dict
- Set Types: set, frozenset

Numeric data type

- Integers, floating point numbers and complex numbers fall under Python numbers category. Complex numbers are written in the form, x + yj, where x is the real part and y is the imaginary part.
- To know the data type of any variable we use the keyword type

```
1  # Online Python compiler (interpreter) to run Python online.
2  # Write Python 3 code in this online editor and run it.
3  x=10
4  print(type(x))
5  x=13.4
6  print(type(x))
7  x="python"
8  print(type(x))
9  x=2+4j
10 print(type(x))
```

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

Boolean

- Booleans are very useful and essential when we start using conditional statements. The Boolean values, True and False are treated as reserved words.

Strings

- A string consists of a sequence of one or more characters, which can include letters, numbers, and other types of characters. A string can also contain spaces.
- We can use single quotes or double quotes to represent strings and it is also called a string literal.
- Multiline strings can be denoted using triple quotes.

```
s = '''This is
        Multiline
                string'''
```

None

None is another special data type in Python. None is frequently used to represent the absence of a value or no value.

Sequence Type:

A Sequence is an ordered collection of items, indexed by positive integers.

It is a combination of mutable (a mutable variable is one, whose value can be changed) and

immutable (an immutable variable is one, whose value can not be changed) data types.

There are two types of sequence data type available in Python, they are:

- list
- tuple

- **List:** Lists are used to store multiple items in a single variable.
- List items are ordered, changeable, and allow duplicate values.
- x = ["apple", "banana", "cherry"]   list
- y = ['apple', 'banana', 'cherry']   list

```
[30] x=["apple", "banana","cherry"]
     print(type(x))

     <class 'list'>

     y=['apple', 'banana','cherry']
     print(type(y))

     <class 'list'>
```

- **Tuple:** Tuples are used to store multiple items in a single variable.
- A tuple is a collection which is ordered and unchangeable.
- Tuples are written with round brackets.
- Tuple items are ordered, unchangeable, and allow duplicate values.
- x = ("apple", "banana", "cherry") tuple
- Y= ('apple', 'banana', 'cherry') tuple

```
[34] x=("apple", "banana","cherry")
     print(type(x))

     <class 'tuple'>

[36] y=('apple', 'banana','cherry')
     print(type(y))

     <class 'tuple'>
```

Mapping Type: dictionary(dict)

- Dictionaries are used to store data values in key: value pairs.
- A dictionary is a collection which is ordered*, changeable and does not allow duplicates.

z = {"name" : "John", "age" : 36} dict

```
z={"name":"john", "age":36}
print(type(z))
```

```
<class 'dict'>
```

Set Types: set, frozenset

Set is an unordered collection of unique items.

Set is defined by values separated by comma inside braces { }. Items in a set are not ordered.

We can perform set operations like union, intersection on two sets.

Sets have unique values.

They eliminate duplicates.

```
a = {5,2,3,1,4}

# printing set variable
print("a = ", a)

# data type of variable a
print(type(a))
```

Output

```
a =  {1, 2, 3, 4, 5}
<class 'set'>
```

```
a = {1,2,2,3,3,3}
print(a)
```

Output

```
{1, 2, 3}
```

- Frozen set is just an immutable version of a Python set .
- While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.

Type Conversion

The process of converting the value of one data type (integer, string, float, etc.) to another data type is called type conversion. Python has two types of type conversion.

1. Implicit Type Conversion
2. Explicit Type Conversion

Implicit Type Conversion

- In Implicit type conversion, Python automatically converts one data type to another data type. This process doesn't need any user involvement.
- Let's see an example where Python promotes the conversion of the lower data type (integer) to the higher data type (float) to avoid data loss.

```
x=10
print(type(x))
y=12.6
print(type(y))
x=x+y
print(x)
print(type(x))
```

```
<class 'int'>
<class 'float'>
22.6
<class 'float'>
>
```

we can see the type of 'x' got automatically changed to the "float" type from the "integer" type. this is a simple case of Implicit type conversion in python.

Now, let's try adding a string and an integer

```
x=10
print(type(x))
y="126"
print(type(y))
X=x+y
```

```
<class 'int'>
<class 'str'>
Traceback (most recent call last):
  File "<string>", line 7, in <module>
TypeError: unsupported operand type(s) for +: 'int' and
    'str'
>
```

As we can see from the output, we got TypeError. Python is not able to use Implicit Conversion in such conditions.

However, Python has a solution for these types of situations which is known as Explicit Conversion.

Explicit Type Conversion

- In Explicit Type Conversion, users convert the data type of an object to required data type.
- We use the predefined functions like int(), float(), str(), etc to perform explicit type conversion.
- This type of conversion is also called typecasting because the user casts (changes) the data type of the objects.

Syntax :

<required_datatype>(expression)

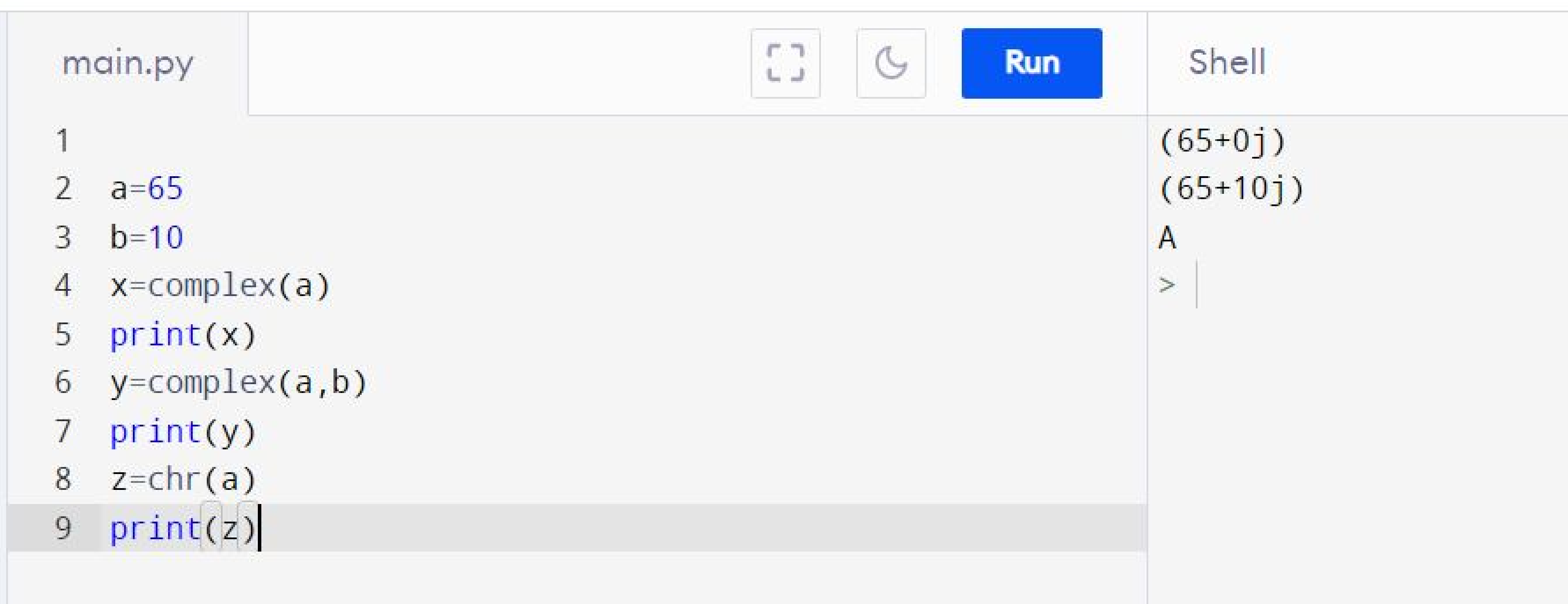
Typecasting can be done by assigning the required data type function to the expression.

```
3   x=10
4   print(type(x))
5   y="126"
6   print(type(y))
7   y=int(y)
8   print(type(y))
9   x=x+y
10  print(x)
11  x=float(x)
12  print(type(x))
13  print(x)
14  x=str(x)
15  print(type(x))
16  print(x)
```

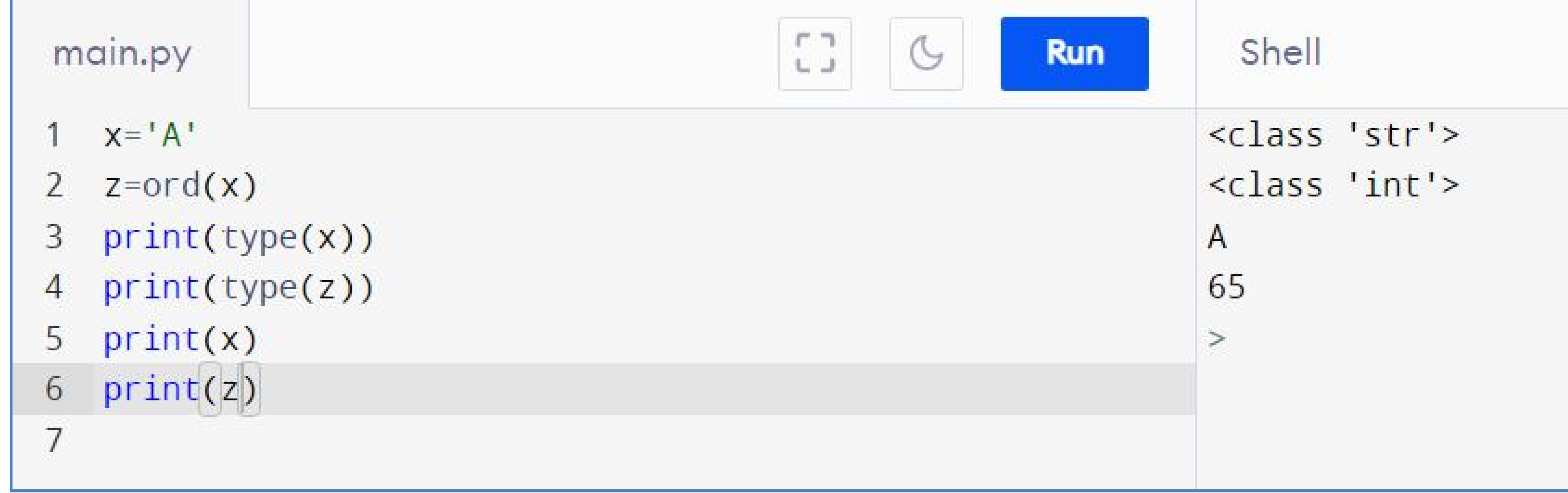
```
<class 'int'>
<class 'str'>
<class 'int'>
136
<class 'float'>
136.0
<class 'str'>
136.0
>
```

complex(): this is used to convert integer value to complex number.

chr(): this is used to convert the integer value in to corresponding ASCII value



- **ord() : ** This function is used to convert a **character to integer.** The *ord()* function returns an integer representing Unicode code point for the given Unicode character.



- **hex() : ** This function is to convert integer to hexadecimal string.

  Convert an integer number (of any size) to a lowercase hexadecimal string prefixed with "0x" using hex() function.

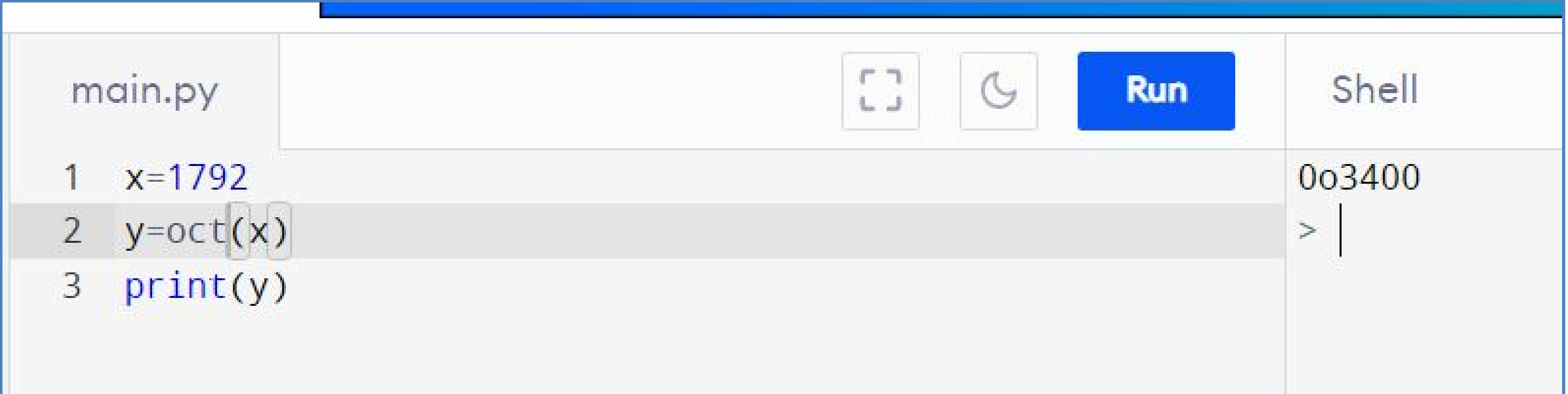
Example:	divide by 16	quotient	remainder
X=1128	1128/16	70	8
	70/16	4	6
	4/16	0	4

Hex(x) = 468

- **oct() :** This function is to convert **integer to octal string**. Convert an integer number (of any size) to an octal string prefixed with “0o” using *oct()*

Example:

Decimal Number	Operation	Quotient	Remainder	Octal Number
1792	÷ 8	224	0	0
224	÷ 8	28	0	00
28	÷ 8	3	4	400
3	÷ 8	0	3	3400



Additional Operators of Python

1. Membership Operators:

Python's membership operators test for membership in a sequence, such as strings, lists, or tuples. There are two membership operators .

- **in:** Evaluates to true if it finds a variable in the specified sequence and false otherwise.

Example: x in y, here in results in a 1 if x is a member of sequence y.

- **not in:** Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.

Example: x not in y, here not in results in a 1 if x is not a member of sequence y.

Untitled11.ipynb

File Edit View Insert Runtime Tools Help All changes saved

+ Code + Text

```
x=['a','b','c']
print('b' in x)
```

True

```
x=['a','b','c']
print('b' not in x)
```

False

2. Identity Operators

Identity operators compare the memory locations of two objects. There are two Identity operators

- **Is :** Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.

Example: x is y, here **is** results in 1 if id(x) equals id(y).

- **is not** :Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.

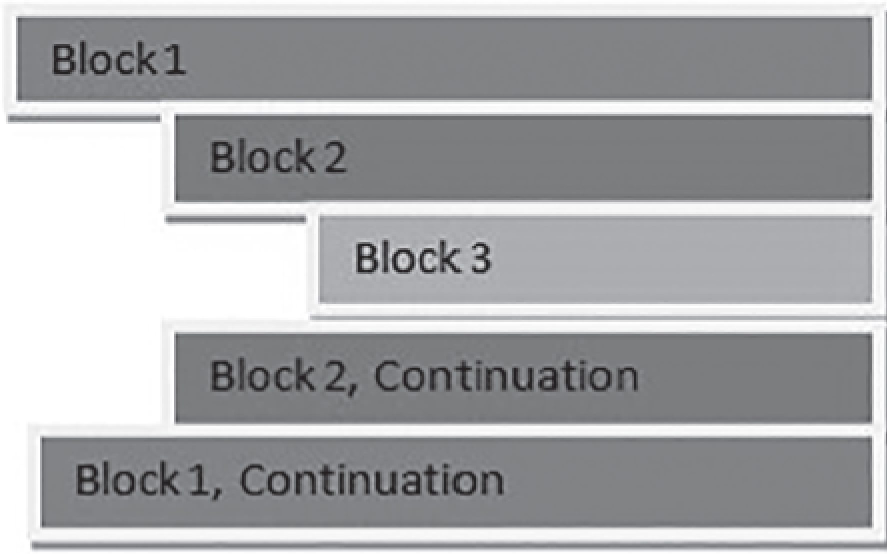
Example: x is not y, here **is not** results in 1 if id(x) is not equal to id(y).

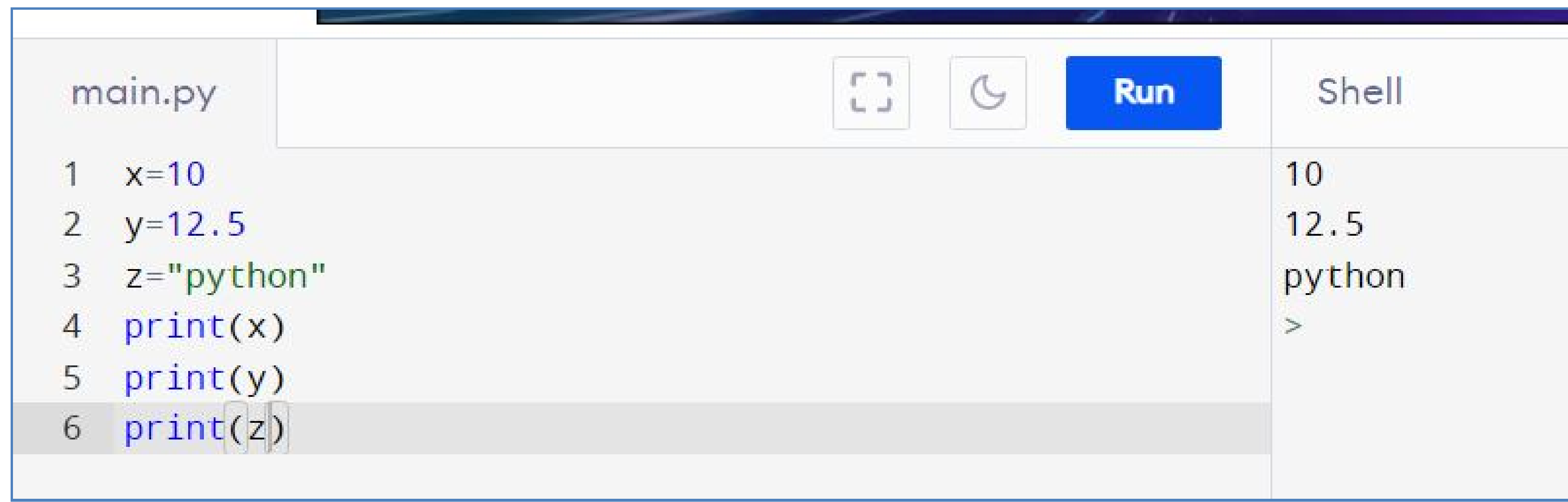
```
a=100
b=20
c=a
print(c is a)
print(b is a)
print(id(a))
print(id(c))
```

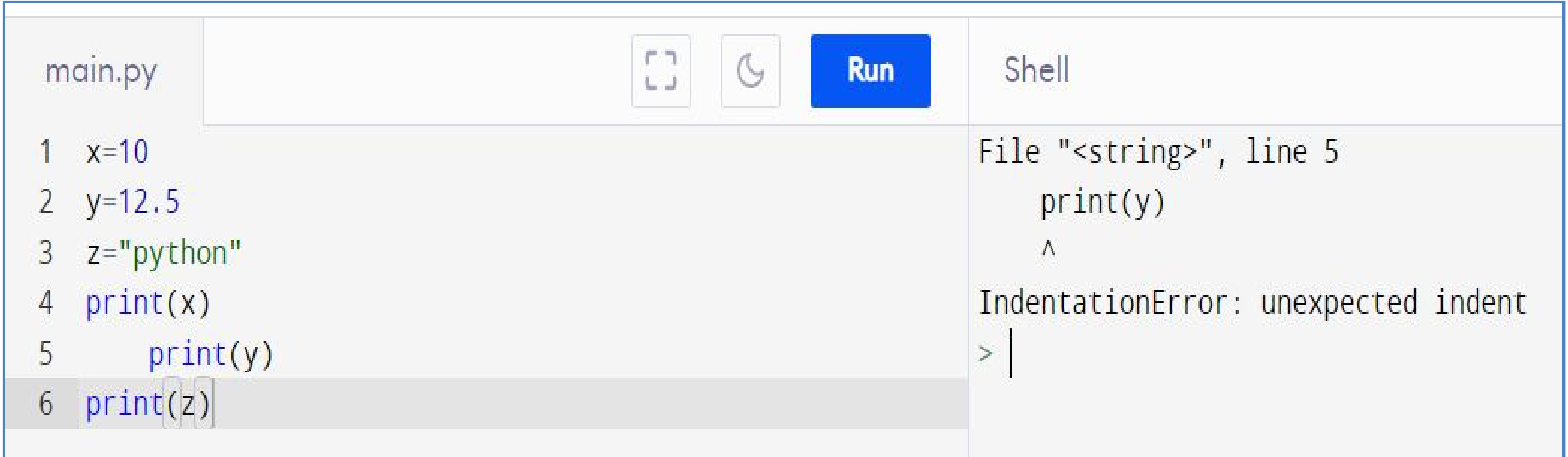
```
True
False
94543155144288
94543155144288
```

Indentation

- Indentation refers to the spaces at the beginning of a code line.
- In many programming languages the indentation in code is only for readability , but this indentation in Python is very important and also mandatory to write programs.
- Python uses indentation to indicate a block of code. In Python, Programs get structured through indentation




main.py
Run
Shell
1 x=10
2 y=12.5
3 z="python"
4 print(x)
5 print(y)
6 print(z)
10
12.5
python
>


main.py
Run
Shell
1 x=10
2 y=12.5
3 z="python"
4 print(x)
5 print(y)
6 print(z)
File "<string>", line 5
print(y)
^
IndentationError: unexpected indent
>

Comments

- Comments are an important part of any program.
- A comment is a text that describes what the program or a particular part of the program is trying to do and is ignored by the Python interpreter.
- Comments are used to help you and other programmers understand, maintain, and debug the program.
- Python uses two types of comments: single-line comment and multiline comments.

Single Line Comment

- In Python, use the hash (#) symbol to start writing a comment.
- Hash (#) symbol makes all text following it on the same line into a comment.
- For example, #This is single line Python comment

Multiline Comments

- use triple quotes. These triple quotes are generally used for multiline strings.

For example,

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

Reading Input

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

  variable_name = input([prompt])

prompt is a string written inside the parenthesis that is printed on the screen. The prompt statement gives an indication to the user that needs to enter the value through the keyboard.

Example:

```
name=input("what is your name")
course=input("which course you study")
mobile=input("give your mobile number")
```

raw_input() function

- Python raw_input function is used to get the values from the user. We call this function to tell the program to stop and wait for the user to input the values.
- It is a built-in function. The input function is **used only in Python 2.x** version.
- The Python 2.x has two functions to take the value from the user.
- The first one is input function and another one is raw_input() function.
- The raw_input() function is similar to input() function in Python 3.x.
- Developers are recommended to use raw_input function in Python 2.x. Because there is a vulnerability in input function in Python 2.x version.

Printing Output

- The *print() function allows a program to display text onto the console.*

Example:

```
name=input("what is your name")
course=input("which course you study")
mobile=input("give your mobile number")
print(name)
print(course)
print(mobile)
```

```
name=input("what is your name")
course=input("which course you study")
mobile=input("give your mobile number")
print(name)
print(course)
print(mobile)
```

```
what is your name  Ayesha
which course you studyPhD
give your mobile number123456
  Ayesha
PhD
123456
```

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

  1. str.format()
  2. f-strings

1. *str.format() Method*

To insert the value of a variable or expression or an object into another string and display it to the user as a single String then we use this str.format() method. The format() method returns a new string with inserted values. The format() method works for all releases of Python 3.x.

The syntax for format() method is,

Print("str".format(variable))

<u>Example: 1</u>

```
country = input("Which country do you live in?")
print("I live in {0}".format(country))
```

<u>Output</u>

```
Which country do you live in? India
I live in India
```

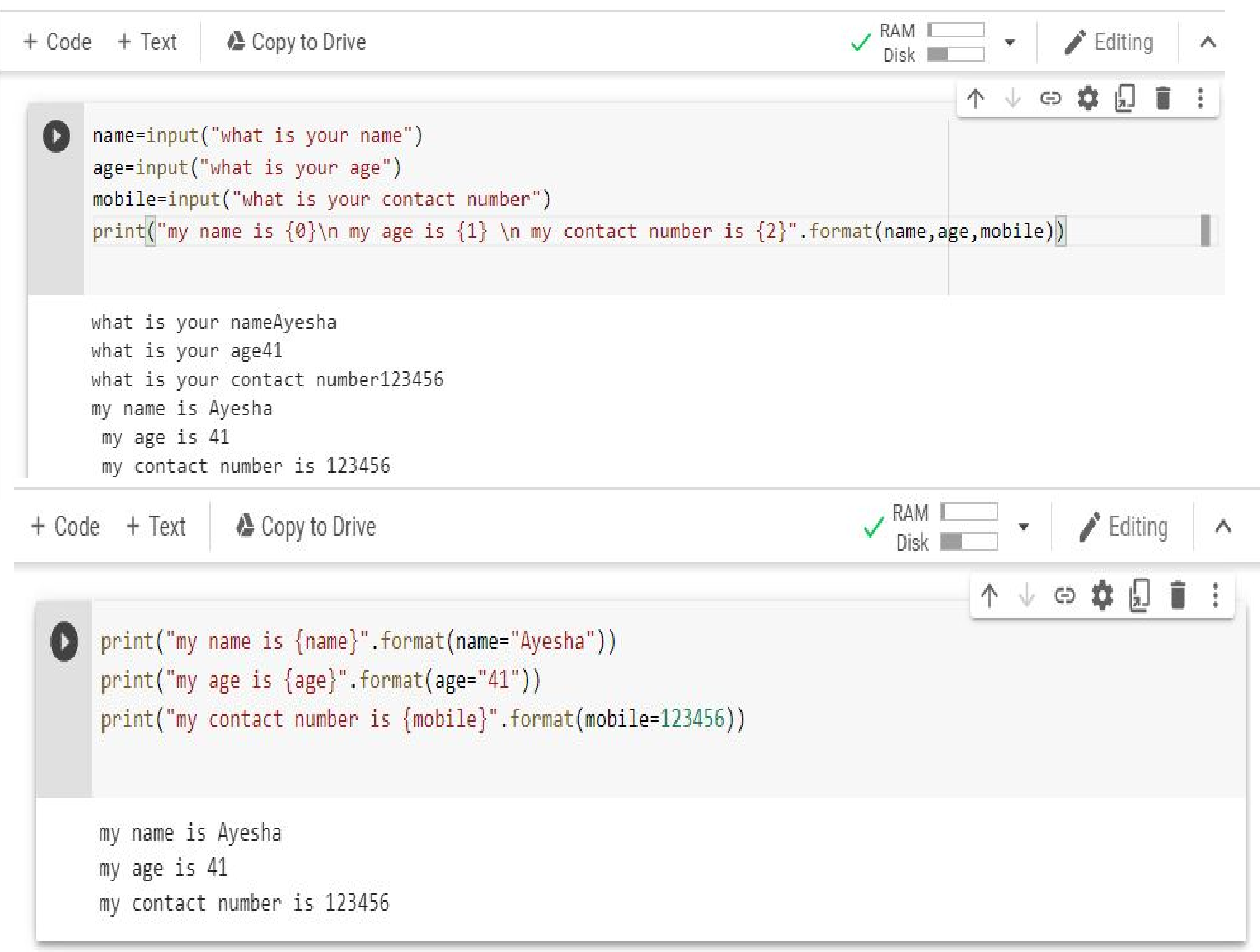
The 0 inside the curly braces {0} is the index of the first (0th) argument (here in this example, it is variable country)

<u>Example 2:</u>

```
a = 10
b = 20
print("The values of a is {0} and b is {1}".format(a, b))
print("The values of b is {1} and a is {0}".format(a, b))
```

<u>Output</u>

```
The values of a is 10 and b is 20
The values of b is 20 and a is 10
```


+ Code
+ Text
Copy to Drive
RAM
Disk
Editing
name=input("what is your name")
age=input("what is your age")
mobile=input("what is your contact number")
print("my name is {0}\n my age is {1} \n my contact number is {2}".format(name,age,mobile))
what is your nameAyesha
what is your age41
what is your contact number123456
my name is Ayesha
my age is 41
my contact number is 123456
+ Code
+ Text
Copy to Drive
RAM
Disk
Editing
print("my name is {name}".format(name="Ayesha"))
print("my age is {age}".format(age="41"))
print("my contact number is {mobile}".format(mobile=123456))
my name is Ayesha
my age is 41
my contact number is 123456

Formatting Types

:< Left aligns the result (within the available space)

```
txt = "We have {:<6} students in CSM / CSD."
print(txt.format(210))
```

:> Right aligns the result (within the available space)

```
txt = "We have {:>6} students in CSM / CSD."
print(txt.format(210))
```

:^ Center aligns the result (within the available space)

```
txt = "We have {:^6} students in CSM / CSD."
print(txt.format(210))
```

```
txt="there are {:<6} students in CSM/CSD"
print(txt.format(210))
txt="there are {:>6} students in CSM/CSD"
print(txt.format(210))
txt="there are {:^6} students in CSM/CSD"
print(txt.format(210))
```

```
there are 210    students in CSM/CSD
there are    210 students in CSM/CSD
there are  210   students in CSM/CSD
```

2. **f-strings:** Formatted strings or f-strings were introduced in Python 3.6. A f-string is a string literal that is prefixed with "f".

Example:

```
country = input("Which country do you live in?")
print(f"I live in {country}")
```

Output

```
Which country do you live in? India
I live in India
```

Input string is assigned to variable country . Observe the character **f** prefixed before the quotes and the variable name is specified within the curly braces.

Python Mathematical Functions

1. factorial(x) : Returns factorial of x. where $x \geq 0$
2. gcd(x, y): Returns the Greatest Common Divisor of x and y
3. remainder(x, y): Find remainder after dividing x by y.
4. pow(x, y): Return the x to the power y value.
5. sqrt(x): Finds the square root of x
6. ceil() :- returns the smallest integral value greater than the number. If number is already integer, same number is returned.
7. floor() :- returns the greatest integral value smaller than the number. If number is already integer, same number is returned.
8. exp(a) :- This function returns the value of e raised to the power a (e**a)

To use these mathematical functions in python we must import math library in the program

For complex numbers import cmath library in the program

main.py

Run

Shell

```
1  import math
2  x=5
3  y=10
4  print("x =",x)
5  print("y =",y)
6  print("factorial of x is",math.factorial(x))
7  print("GCD of x and y is",math.gcd(x,y))
8  print("Remainder of x div y is",math.remainder(y, x))
9  print("x power y value is",math.pow(x,y))
10 print("square root of x value is",math.sqrt(x))
11 z=2.3
12 print("z =",z)
13 print ("The ceil of z is :",math.ceil(z))
14 print ("The floor of z is :",math.floor(z))
15 print ("e power x values is :",math.exp(x))
```

```
x = 5
y = 10
factorial of x is 120
GCD of x and y is 5
Remainder of x div y is 0.0
x power y value is 9765625.0
square root of x value is 2.23606797749979
z = 2.3
The ceil of z is : 3
The floor of z is : 2
e power x values is : 148.4131591025766
>
```

Activate Window
Go to PC settings to a

9. min() : function returns the item with the lowest value
10. max: function returns the item with the highest value

main.py | Run | Shell

```
1
2  print(min(2,4,7,9,12))
3  print(max(12,56,23,87,98))
4  print(min("Mike", "John", "Vicky"))
5  print(max("Mike", "John", "Vicky"))
```

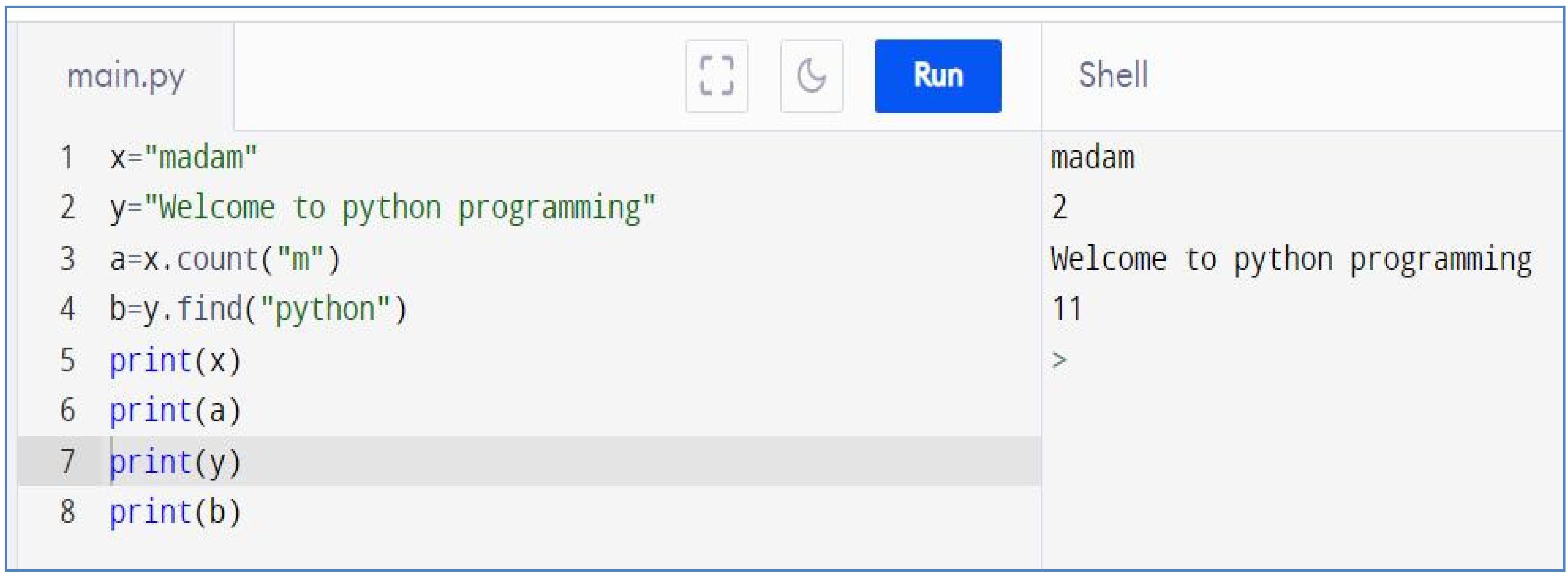
```
2
98
John
Vicky
>
```

Some functions do not need to import math library function

Python String Functions

1. capitalize() Converts the first character to upper case
2. casefold() Converts string into lower case

3. count() Returns the number of times a specified value occurs in a string
4. find() Searches the string for a specified value and returns the position of where it was found



5. upper(): converts the string in to upper case
6. swapcase()Swaps cases, lower case becomes upper case and vice versa
7. title()Converts the first character of each word to upper case

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

PYTHON PROGRAMMING

B.Tech I Year II Sem

Dr Ayesha Banu

Assistant Professor

Department of CSE

Practice Programs

1. Write a Python program to find those numbers which are divisible by 7 and multiple of 5, between 1500 and 2700 (both included).
2. Write a Python program that accepts a string and calculate the number of digits and letters.
3. Find the sum of the series 2 +22 + 222 + 2222 + .. n terms
4. Program to Find the Sum of the Series: 1 + x^2/2 + x^3/3 + … x^n/n
5. Program to Find the Sum of the Series: 1 + 1/2 + 1/3 + ….. + 1/N
6. Write a program to check whether the last digit of a number(entered by user) is divisible by 3 or not.
7. Accept three sides of triangle and check whether the triangle is possible or not.

(triangle is possible only when sum of any two sides is greater than 3rd side)

8. Write the output of the following if a = 9

```
if (a > 5 and a <=10):
    print("Hello")
else:
    print("Bye")
```

9. Accept the age of 4 people and display the youngest one?

Unit – I

Introduction to Python: What is Python?, What is Python Good For?, Python History, How does Python Execute a Program, Review of a Simple Program, Some of the Basic Commands, Variables, Statements, Input/Output Operations, Keywords, Variables, Assigning values, Standard Data Types, Strings, Operands and operators.

Unit – II

Understanding the Decision Control Structures: The if Statement, A Word on Indentation, The if … else Statement, The if … elif … else Statement,

Loop Control Statements: The while Loop, The for Loop, Infinite Loops, Nested Loops.

The break Statement, The continue Statement, The pass Statement, The assert Statement, The return Statement.

Unit – III

Functions- Function Definition and Execution, Scoping, Arguments: Arguments are Objects, Argument Calling by Keywords, Default Arguments, Function Rules, Return Values.

Advanced Function Calling: The apply Statement, The map Statement, Indirect Function Calls.

Unit - IV

Lists: List, Creating List, Updating the Elements of a List, Sorting the List Elements. Storing Different Types of Data in a List, Nested Lists, Nested Lists as Matrices.

Tuples: Creating Tuple, Accessing the Tuple Elements, Basic Operations on Tuples, Functions to Process Tuples, Inserting Elements in a Tuple, Modifying Elements of a Tuple, Deleting Elements from a Tuple.

Sets: Creating Set, Basic Operations on Sets, Methods of Set.

Dictionaries: Operations on Dictionaries, Dictionary Methods, Using for Loop with Dictionaries, Sorting the Elements of a Dictionary using Lambdas, Converting Lists into Dictionary.

Unit – V

Modules: Importing a Module, Tricks for Importing Modules, Packages.

Exceptions and Error Trapping: What is an Exception?, Exception Handling: try..except..else.., try..finally.., Exceptions Nest, Raising Exceptions, Built-In Exceptions.

Unit – II

Understanding the Decision Control Structures: The if Statement, A Word on Indentation, The if … else Statement, The if … elif … else Statement.

Loop Control Statements: The while Loop, The for Loop, Infinite Loops, Nested Loops. The break Statement, The continue Statement, The pass Statement, The assert Statement, The return Statement.

Control Flow Statements

- Python programs are generally executed *sequentially* from top to bottom, in the order that they appear. Apart from sequential control flow statements you can employ decision making and looping control flow statements to break up the flow of execution thus enabling your program to conditionally execute particular blocks of code.

The control flow statements in Python Programming Language are

1. **Sequential Control Flow Statements:** This refers to the line by line execution, in which the statements are executed sequentially, in the same order in which they appear in the program.
2. **Decision Control Flow Statements:** Depending on whether a condition is True or False, the decision structure may skip the execution of an entire block of statements or even execute one block of statements instead of other (if, if…else and if…elif…else).
3. **Loop Control Flow Statements:** This is a control structure that allows the execution of a block of statements multiple times until a loop termination condition is met (*for* loop and *while* loop). Loop Control Flow Statements are also called Repetition statements or Iteration statements.

Indentation

- Indentation refers to the spaces at the beginning of a code line.
- In many programming languages the indentation in code is only for readability , but this indentation in Python is very important and also mandatory to write programs.
- Python uses indentation to indicate a block of code. In Python, Programs get structured through indentation

main.py

```
1 x=10
2 y=12.5
3 z="python"
4 print(x)
5 print(y)
6 print(z)
```

Shell

```
10
12.5
python
>
```

main.py

```
1 x=10
2 y=12.5
3 z="python"
4 print(x)
5     print(y)
6 print(z)
```

Shell

```
File "<string>", line 5
    print(y)
    ^
IndentationError: unexpected indent
>
```

The *if* Decision Control Flow Statement (Simple if)

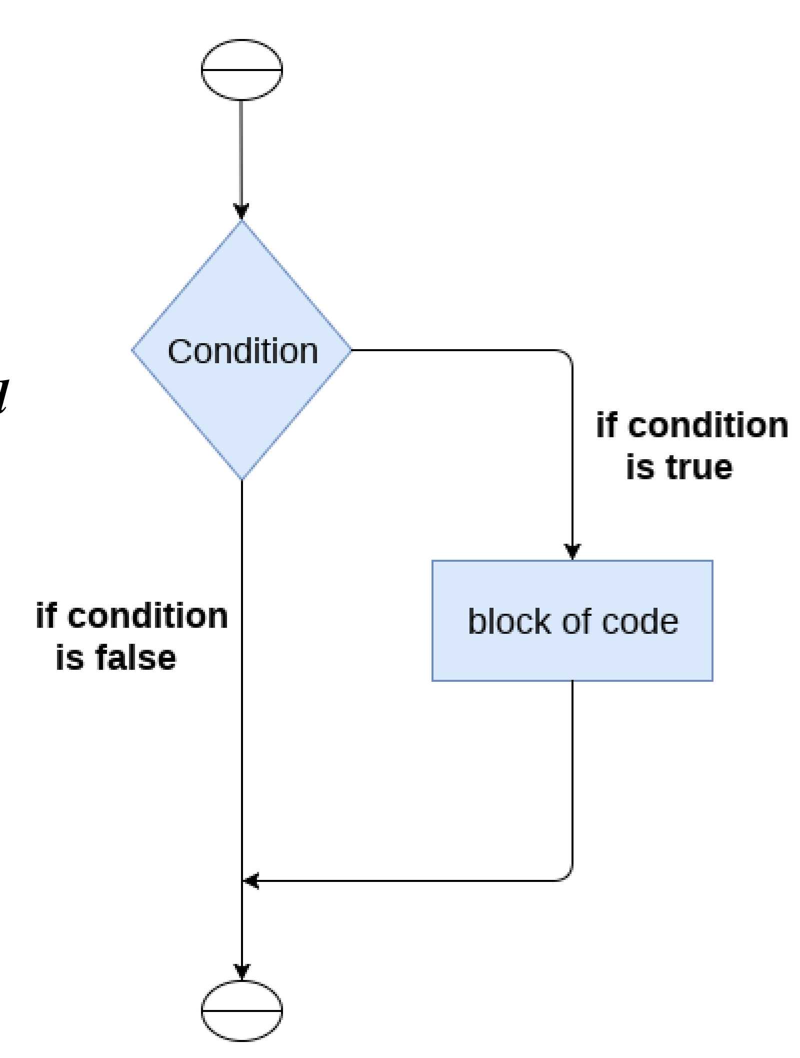
The syntax for ***if*** statement is,

Keyword

if Boolean_ Expression: → ***Colon should be present at the end***

statement (s)

Indentation



The *if* decision control flow statement starts with *if* keyword and ends with a colon.

- The expression in an *if* statement should be a Boolean expression.
- The *if* statement decides whether to run some particular statement or not depending upon the value of the Boolean expression.
- If the Boolean expression evaluates to *True* then statements in the *if* block will be executed;
- otherwise the result is *False* then none of the statements are executed.

Example:

```
 if 20 > 10:
 …      print("20 is greater than 10")
```

Output

20 is greater than 10

Example:

```
 if 20 < 10:
 …      print("20 is less than 10")
```

Output

Example 2: write a if condition to check given number is even

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even")
```

Output:

enter the number? 10

Number is even

Output:

enter the number? 13

Example 3: write a if condition to check given number is positive

```
num = int(input("enter the number?"))
if num> 0:
    print("Number is positive")
```

Output:

enter the number? 10

Number is positive

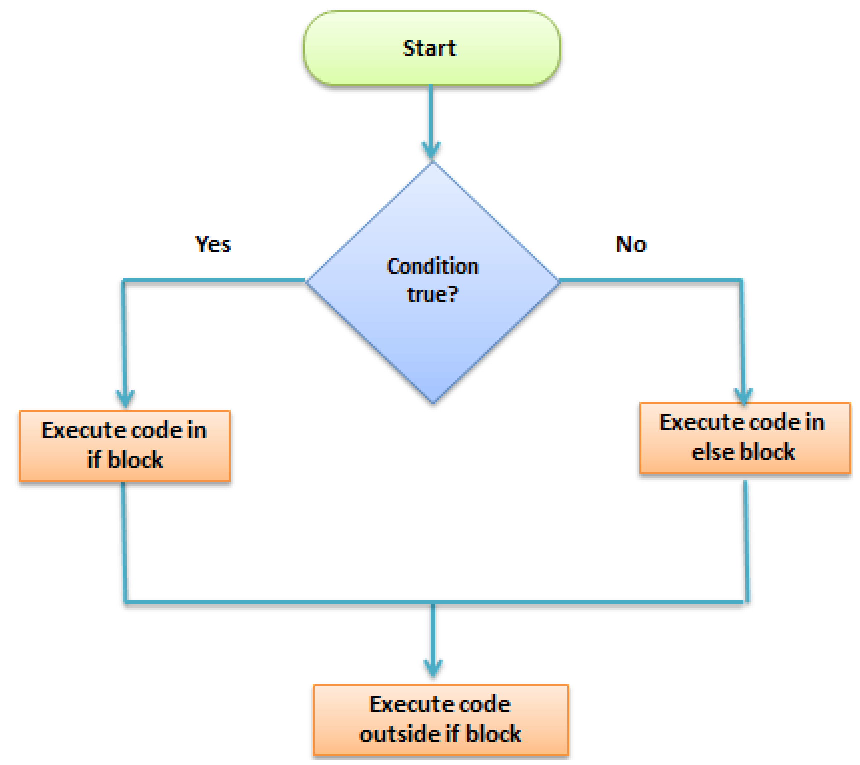
Output:

enter the number? -12

The *if…else* Decision Control Flow Statement

An *if* statement can also be followed by an *else* statement which is optional. An *else* statement does not have any condition. Statements in the *if* block are executed if the Boolean_Expression is *True*. Use the optional *else* block to execute statements if the Boolean_Expression is *False*.

The ***if…else*** statement allows for a two-way decision.



The syntax for ***if…else*** statement is,

```
if Boolean_Expression:
    statement_1
else:
    statement_2
```

Example 1: Program to check whether a number is even or odd.

```
num = int(input("enter the number?"))
if num%2 == 0:
    print("Number is even...")
else:
    print("Number is odd...")
```

Output:

enter the number? 10

Number is even..

Output:

enter the number? 13

Number is odd..

Example 2: Program to check whether a person is eligible to vote or not.

```
age = int (input("Enter your age? "))
if age>=18:
    print("You are eligible to vote !!");
else:
    print("Sorry! you have to wait !!");
```

Output:

Enter your age? 90You are eligible to vote !!

Example 3: Program to check whether a number is positive or negative.

```
num = int(input("enter the number?"))
if num< 0:
    print("Number is positive...")
else:
    print("Number is negative...")
```

Output:

enter the number?10

Number is positive

Output:

enter the number? -12

Number is negative

Example 4: Program to Find the Greater of Two Numbers

```
x = int(input("enter the first value?"))
Y = int(input("enter the second value?"))
if x>y:
    print(“the greatest number is :“,x)
else:
    print(“the greatest number is :“,y)
```

Output:

Enter the first number 8

Enter the second number 10

the greater number is 10

Output:

Enter the first number 12

Enter the second number 10

the greater number is 12

The *if…elif…else* Decision Control Statement

The ***if…elif…else*** is also called as multi-way decision control statement.

When we need to choose from several possible alternatives, then an *elif* statement is used along with an *if* statement.

The keyword ‘*elif*’ is short for ‘else if’ and is useful to avoid excessive indentation.

The *else* statement must always come last, and will again act as the default action.

The syntax for ***if…elif…else*** statement is,

if Boolean_Expression_1:

statement_1

elif Boolean_Expression_2:

statement_2

elif Boolean_Expression_3:

statement_3

:

:

:

else:

statement_last

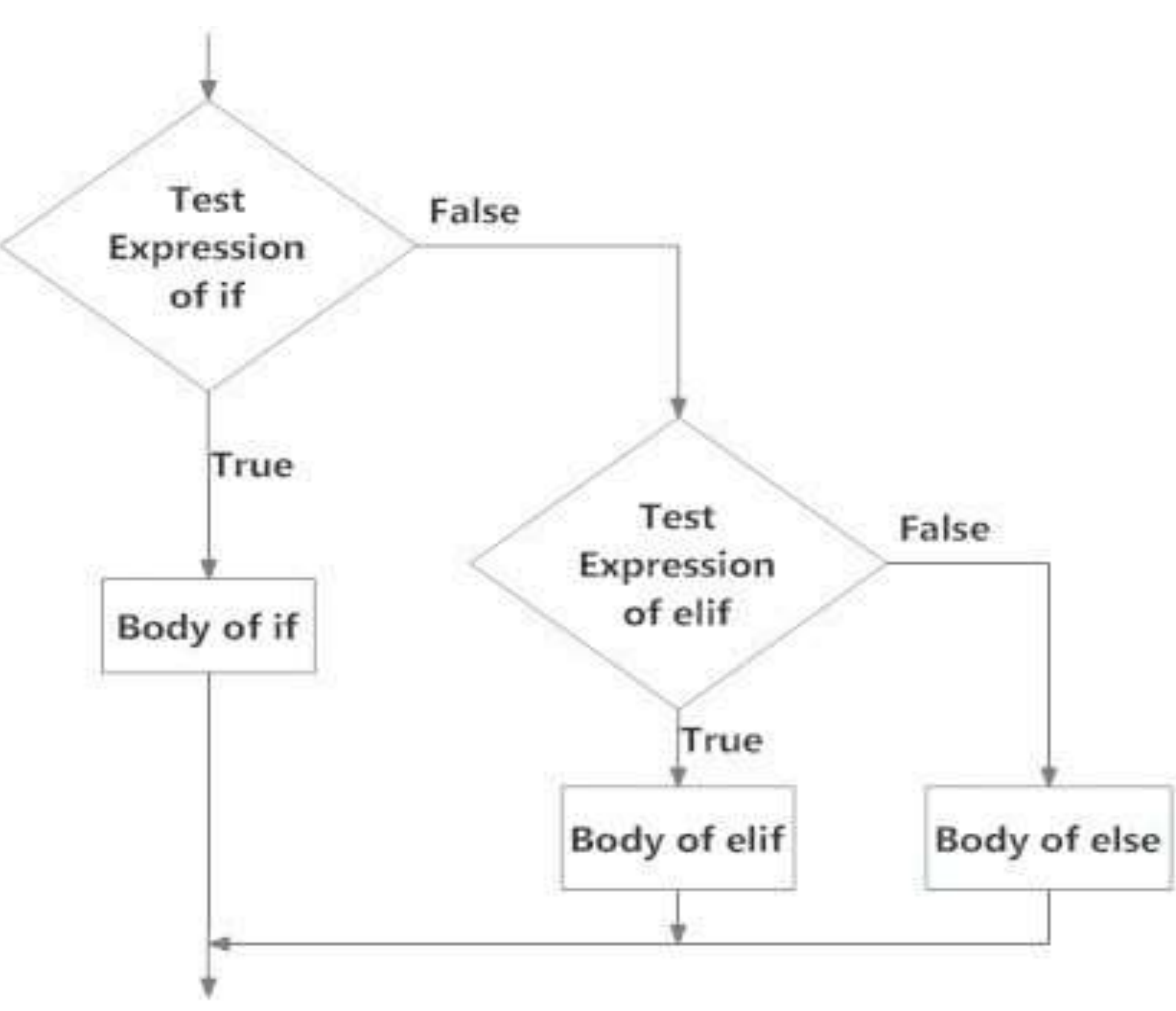


Fig: Operation of if...elif...else statement

Example 1:Python program to find the largest among the three numbers

```
num1 = float(input("Enter first number: "))
num2 = float(input("Enter second number: "))
num3 = float(input("Enter third number: "))
if (num1 >= num2) and (num1 >= num3):
  largest = num1
elif (num2 >= num1) and (num2 >= num3):
  largest = num2
else:
  largest = num3
print("The largest number is", largest)
```

OUTPUT

```
Enter first number: 47
Enter second number: 58
Enter third number: 21
The largest number is 58.0
```

Example:2 Write a Program to Prompt for a Score between 0.0 and 1.0. If the Score Is Out of Range, Print an Error. If the Score Is between 0.0 and 1.0, Print a Grade Using the Following Table

Score	Grade
>= 0.9	A
>= 0.8	B
>= 0.7	C
>= 0.6	D
< 0.6	F

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

OUTPUT

```
Enter your score  0.92
Your Grade is "A"
```

Example 3: Write a python program to calculate the grades of a student ? Accept name , rollno, 5 subject marks. Display the result

The grade must be calculated as per following rules:

Average Mark	Grade
91-100	A1
81-90	A2
71-80	B1
61-70	B2
51-60	C1
41-50	C2
33-40	D
21-32	E1
0-20	E2

```
print("enter students name and htno")
name=input()
htno=int(input())
print("Enter Marks Obtained in 5 Subjects: ")
m1 = int(input())
m2 = int(input())
m3 = int(input())
m4 = int(input())
m5 = int(input())
tot = m1+m2+m3+m4+m5
avg = tot/5
print("**************************")
print("Name:=\t", name)
print("Hall tkt no:\t",htno)
```

```
print("total marks:\t",tot)
print("average marks:\t",avg)
if avg>=91 and avg<=100:
    print("Your Grade is A1")
elif avg>=81 and avg<91:
    print("Your Grade is A2")
elif avg>=71 and avg<81:
    print("Your Grade is B1")
elif avg>=61 and avg<71:
    print("Your Grade is B2")
elif avg>=51 and avg<61:
    print("Your Grade is C1")
elif avg>=41 and avg<51:
    print("Your Grade is C2")
elif avg>=33 and avg<41:
    print("Your Grade is D")
```

```
elif avg>=21 and avg<33:
    print("Your Grade is E1")
elif avg>=0 and avg<21:
    print("Your Grade is E2")
else:
    print("Invalid Input!")
print("***************")
```

- Write a program to calculate electricity bill using following details and print the bill

  units = (prev_month_read) – (curr_month_read)

  units 1 – 100 bill= units*1.5

  units 101-200 bill=units* 2.5

  units 201-300 bill= units* 4

  units 300 – 350 bill= units *5

  units > 350 fixed bill 1500

- Write a program to calculate Employees Salary and print the pay slip

  Input: number of days worked
  wages per day

  calculate: basic pay=wages*days

  HRA=basic*0.1 (10 % of basic pay)

  DA=basic*0.05 (5% of basic pay)

  PF=basic*0.12 (12 % of basic pay) netsalary=basic pay +HRA+DA-PF;

Nested *if* Statement

In some situations, you have to place an *if* statement inside another statement. An *if* statement that contains another *if* statement either in its *if* block or *else* block is called a Nested *if* statement.

The syntax of the nested *if* statement is,

```
if Boolean_Expression_1:
    if Boolean_Expression_2:
        statement_1
    else:
        statement_2
else:
    statement_3
```

❖ **Program to Check If a Given Year Is a Leap Year**

```
year = int(input("enter any year"))
if (year % 4) == 0:
   if (year % 100) == 0:
       if (year % 400) == 0:
           print(year,"is a leap year")
       else:
           print(year,"is not a leap year")
   else:
       print(year,"is a leap year")
else:
   print(year,"is not a leap year")
```

OUTPUT

Enter any year 2014
2014 is not a Leap Year

Enter any year 2000
2000 is a Leap Year

❖Program to find greatest of 3 numbers using nested if

```
a=int(input("Enter A: "))
b=int(input("Enter B: "))
c=int(input("Enter C: "))
if a>b:
    if a>c:
        g=a
    else:
        g=c
else:
    if b>c:
        g=b
    else:
        g=c
print("Greater  = ",g)
```

Output

```
Enter A: 10
Enter B: 20
Enter C: 30
Greater  =  30
```

Python program to find roots of quadratic equation

```
import math
print("enter a, b, c values")
a=int(input())
b=int(input())
c=int(input())
dis = b * b - 4 * a * c
sqrt_val = math.sqrt(abs(dis))
# checking condition for discriminant
if dis > 0:
    print(" real and different roots ")
    print((-b + sqrt_val)/(2 * a))
    print((-b - sqrt_val)/(2 * a))
elif dis == 0:
    print(" real and same roots")
    print(-b / (2 * a))
 # when discriminant is less than 0
else:
    print("Complex Roots")
    print(- b / (2 * a), " + i", sqrt_val)
    print(- b / (2 * a), " - i", sqrt_val)
```

Input :a = 1, b = 2, c = 1
Output : Roots are real and same -1.0

Input :a = 2, b = 2, c = 1
Output : Roots are complex -0.5 + i 2.0
 -0.5 - i 2.0

Input :a = 1, b = 10, c = -24
Output : Roots are real and different 2.0 -12.0

1. Program to Check if a character is Vowel or Consonant
2. program to enter week number and print day of week
3. Accept three sides of a triangle and check whether it is an equilateral, isosceles or scalene triangle.

- An equilateral triangle is a triangle in which all three sides are equal.
- A scalene triangle is a triangle that has three unequal sides.
- An isosceles triangle is a triangle with (at least) two equal sides.

4. Write a program to accept two numbers and mathematical operators and perform operation accordingly.
5. Accept three numbers from the user and display the second largest number.
6. Accept three sides of triangle and check whether the triangle is possible or not.

(triangle is possible only when sum of any two sides is greater than 3rd side)

Short Hand If

If you have only one statement to execute, you can put it on the same line as the if statement.

Example

One line if statement:

```
if a > b: print("a is greater than b")
```

Short Hand If ... Else

If you have only one statement to execute, one for if, and one for else, you can put it all on the same line:

Example

One line if else statement:

```
a = 2
b = 330
print("A") if a > b else print("B")
```

This technique is known as **Ternary Operators**, or **Conditional Expressions**.

We can also have multiple else statements on the same line:

Example

One line if else statement, with 3 conditions:

```
a = 330
b = 330
print("A") if a > b else print("=") if a == b else print("B")
```

```
a = 330
b = 330
if a > b:
 print("A")
elif a == b:
 print("=")
else:
 print("B")
```

And

The and keyword is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, AND if c is greater than a:

```
a = 200
b = 33
c = 500
if a > b and c > a:
  print("Both conditions are True")
```

Or

The or keyword is a logical operator, and is used to combine conditional statements:

Example

Test if a is greater than b, OR if a is greater than c:

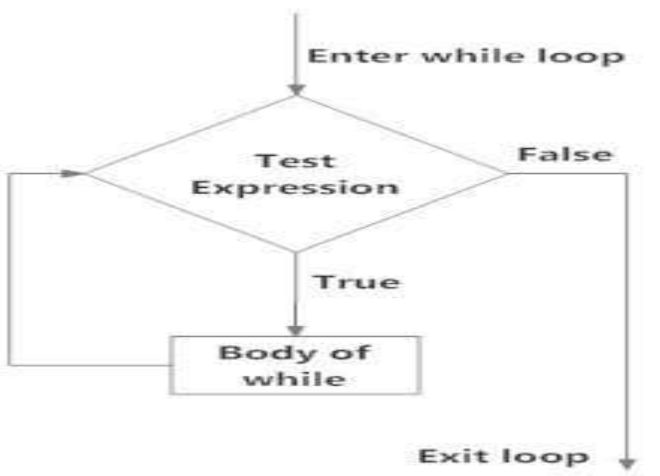
```
a = 200
b = 33
c = 500
if a > b or a > c:
  print("At least one of the conditions is True")
```

Python Loops

Python has two primitive loop commands:

1. while loops
2. for loops

The *while* Loop



The syntax for *while* loop is,

while Boolean_Expression:

statement(s)

The *while* loop starts with the *while* keyword and ends with a colon. With a *while* statement, the first thing that happens is that the Boolean expression is evaluated before the statements in the *while* loop block is executed.

If the Boolean expression evaluates to *False*, then the statements in the *while* loop block are never executed.

If the Boolean expression evaluates to *True*, then the *while* loop block is executed.

After each iteration of the loop block, the Boolean expression is again checked, and if it is *True*, the loop is iterated again. Each repetition of the loop block is called an iteration of the loop. This process continues until the Boolean expression evaluates to *False* and at this point the *while* statement exits.

Program 1 : Display First 10 Numbers Using *while* Loop Starting from 0

```
i = 0
while i < 10:
   print(i)
   i = i + 1
```

Program 2 : Display First 10 Numbers Using *while* Loop in reverse order

```
i = 10
while i >= 0:
   print(i)
   i = i - 1
```

Program 3: Find sum and product of n natural numbers

```
n=int(input("enter n value"))
i = 1
sum=0
prod=1
while i<= n:
   sum=sum+i
   prod=prod*i
   i = i + 1
print("the sum of",n,"natural numbers is",sum)
print("the prod of",n,"natural numbers is",prod)
```

OUTPUT

```
enter n value10
the sum of 10 natural numbers is 55
the prod of 10 natural numbers is 3628800
```

??? Write a program to find factorial of a given number

??? Write a program to find sum of squares of first 10 natural numbers

4. Program to accept any number and find sum of digits of the number

```
n=int(input("enter any number"))
sum=0
temp=n
while n>0:
   rem=n%10
   sum=sum+rem
   n=n//10
print("the sum of digits of",temp,"is",sum)
```

OUTPUT

```
enter any number548
the sum of digits of 548 is 17
```

5. Write a to determine whether a given number is armstrong.

```
n=int(input("enter any number"))
sum=0
temp=n
while n>0:
   rem=n%10
   sum=sum+(rem*rem*rem)
   n=n//10
if sum==temp:
    print("the number is armstrong")
else:
    print("the number is not armstrong")
```

OUTPUT

```
enter any number125
the number is not armstrong

enter any number153
the number is armstrong
```

6. Program to Check whether a given number is palindrome

```
n=int(input("enter any number"))
rev=0
temp=n
while n!=0:
   rem=n%10
   rev=rev*10+rem
   n=n//10
print("the number entered is",temp)
print("the reverse of the number is",rev)
if temp==rev:
    print("the number is palindrome")
else:
    print("the number is not palindrome")
```

OUTPUT

enter any number151

the number entered is 151

the reverse of the number is 151

the number is palindrome

enter any number122

the number entered is 122

the reverse of the number is 221

the number is not palindrome

The *for* Loop

The syntax for the *for* loop is,

for iteration_variable in sequence:

statement(s)

The *for* loop starts with *for* keyword and ends with a colon. The first item in the sequence gets assigned to the iteration variable *iteration_variable*. Here, *iteration_variable* can be any valid variable name.

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

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

Example

Print each fruit in a fruit list:

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

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

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

```
apple
banana
mango
```

Looping Through a String

Even strings are iterable objects, they contain a sequence of characters:

Example

Loop through the letters in the word “apple":

```
for x in "apple":
    print(x)
```

```
] for x in "apple":
    print(x)

a
p
p
l
e
```

```
fruits="apple"
for x in fruits:
  print(x)

a
p
p
l
e
```

Program to find the sum and product of all numbers stored in a list

```
numbers = [6, 5, 3, 8, 4, 2, 5, 4, 11]
sum = 0
prod = 1

for val in numbers:
    sum = sum+val
    prod = prod*val

print("The sum is", sum)
print("The product is", prod)
```

OUTPUT

```
The sum is 48
The product is 1267200
```

The range() function

We can generate a sequence of numbers using range() function

Example: range(10) will generate numbers from 0 to 9 (10 numbers).

Program to print WELCOME 10 times

```
for i in range(10):
   print("Welcome")
```

prints Welcome 10 times

```
for i in range(10):
   print(i)
```

prints numbers 0 to 9

The range() function defaults to 0 as a starting value,

however it is possible to specify the starting value by adding a parameter:

range(2, 6), which means values from 2 to 6 (but not including 6):

Example:

```
for i in range(2,10):
    print(i)
```

prints numbers 2 to 9 , 2 is the starting value

The range() function defaults to increment the sequence by 1,

however it is possible to specify the increment value by adding a third parameter:

range(2, 30, **3**): prints numbers starting from 2 to 29 by incrementing 3 times.

Example:

```
for i in range(2,30,3):
    print(i)
```

prints numbers 2 5 8 11 14 17 20 23 26 29

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

range([start ,] stop [, step])

Both start and step arguments are optional and the range argument value should always be an integer.

start → value indicates the beginning of the sequence. If the start argument is not specified, then the sequence of numbers start from zero by default.

stop → Generates numbers up to this value but not including the number itself.

step → indicates the difference between every two consecutive numbers in the sequence. The step value can be both negative and positive but not zero.

NOT E: The square brackets in the syntax indicate that these arguments are optional. You can leave them out.

Program 1: program to find the factorial of a number

```
n=int(input("enter any number"))
fact=1
if n<0:
   print("factorial does not exist for -ve numbers")
elif n==0:
   print("factorial of 0 is equal to 1")
else:
   for i in range(1,n+1):
           fact=fact*i
   print("factorial of",n,"is equal to",fact)
```

OUTPUT

enter any number-8

factorial does not exist for -ve numbers

enter any number0

factorial of 0 is equal to 1

enter any number5

factorial of 5 is equal to 120

Program 2: program to Fibonacci series for a given number

```
n=int(input("enter any number"))
first=0
second=1
print(first)
print(second)
for i in range(2,n):
   total=first+second
   print(total)
   first=second
   second=total
```

OUTPUT

```
enter any number10
0
1
1
2
3
5
8
13
21
34
```

Program 3: program to check if a number is prime or not

```
n=int(input("enter any number"))
nof=0
for i in range(1,n+1):
   if n%i==0:
            nof=nof+1
print("number of factors=",nof)
if nof==2:
   print(n,"is a prime number")
else:
   print(n,"is not a prime number")
```

OUTPUT

```
enter any number11
number of factors= 2
11 is a prime number

enter any number24
number of factors= 8
24 is not a prime number
```

Else in For Loop

The else keyword in a for loop specifies a block of code to be executed when the loop is finished:

Example:

Print all numbers from 0 to 5, and print a message when the loop has ended:

```
for x in range(6):
    print(x)
  else:
    print("Finally finished!")
```

Run »


```
for x in range(6):
  print(x)
else:
  print("Finally finished!")
```


```
0
1
2
3
4
5
Finally finished!
```

Note: The else block will NOT be executed if the loop is stopped by a break statement.

Python Nested Loops

- A nested loop is a loop inside a loop.
- The "inner loop" will be executed one time for each iteration of the "outer loop":
- Python programming language allows to use one loop inside another loop.

```
for iterating_var in sequence: -------------→Outer Loop
   for iterating_var in sequence: -------------→Inner Loop
         statements(s)
   statements(s)
```

```
while expression:
   while expression:
         statement(s)
   statement(s)
```

Program to print patterns of stars

```
n=int(input("enter n value"))
for i in range(0,n):
    for j in range(0,i+1):
        print("*",end=" ")
    print()
```

```
enter n value3
*
* *
* * *
```

```
n=int(input("enter n value"))
for i in range(n+1,0,-1):
    for j in range(0,i-1):
        print("*",end=" ")
    print()
```

```
enter n value3
* * *
* *
*
```

```
n=int(input("enter n value"))
for i in range(1,n+1):
    for j in range(1,i+1):
        print(i,end=" ")
    print()
```

```
enter n value5
1
2 2
3 3 3
4 4 4 4
5 5 5 5 5
```

```
n=int(input("enter n value"))
for i in range(1,n+1):
    for j in range(1,i+1):
        print(j,end=" ")
    print()
```

```
enter n value5
1
1 2
1 2 3
1 2 3 4
1 2 3 4 5
```

https://www.javatpoint.com/how-to-print-pattern-in-python

Program to print prime numbers between 2 to 20 using nested while loop

```
i = 2
while(i < 20):
   j = 2
   while(j <= (i/j)):
      if not(i%j):
        break
      j = j + 1
   if (j > i/j) :
     print(i, " is prime")
   i = i + 1
```

OUTPUT

2

3

5

7

11

13

17

19

Program to print multiplication table using nested while loop

```
i=1
while i<=5:
  j=1
  while j<=5:
       print(i*j,end=" ")
       j+=1
  i+=1
  print("\n")
```

```
1 2 3 4 5

2 4 6 8 10

3 6 9 12 15

4 8 12 16 20

5 10 15 20 25
```

Python Infinite Loop

- An Infinite Loop in Python is a continuous repetitive conditional loop that gets executed until an external factor interferes in the execution flow, or we terminate the program.
- A loop becomes infinite loop if a condition never becomes FALSE.

```
i=0
while(i<=10):
   print("hai")
```

we do not give the increment statement and the loop enters in to an infinite loop

```
i=0
while True :
    print("Hello")
i=i+1
```

we do not give the termination condition and the loop enters in to an infinite loop

Transfer Statements of Python:

Transfer statements are also called as Loop Control Statements that change execution of loop from its normal sequence. There are 3 transfer statements supported by python

1. break
2. continue
3. Pass

Python break statement

- The break statement terminates the loop containing it. Control of the program flows to the statement immediately after the body of the loop.
- If the break statement is inside a nested loop (loop inside another loop), the break statement will terminate the innermost loop.

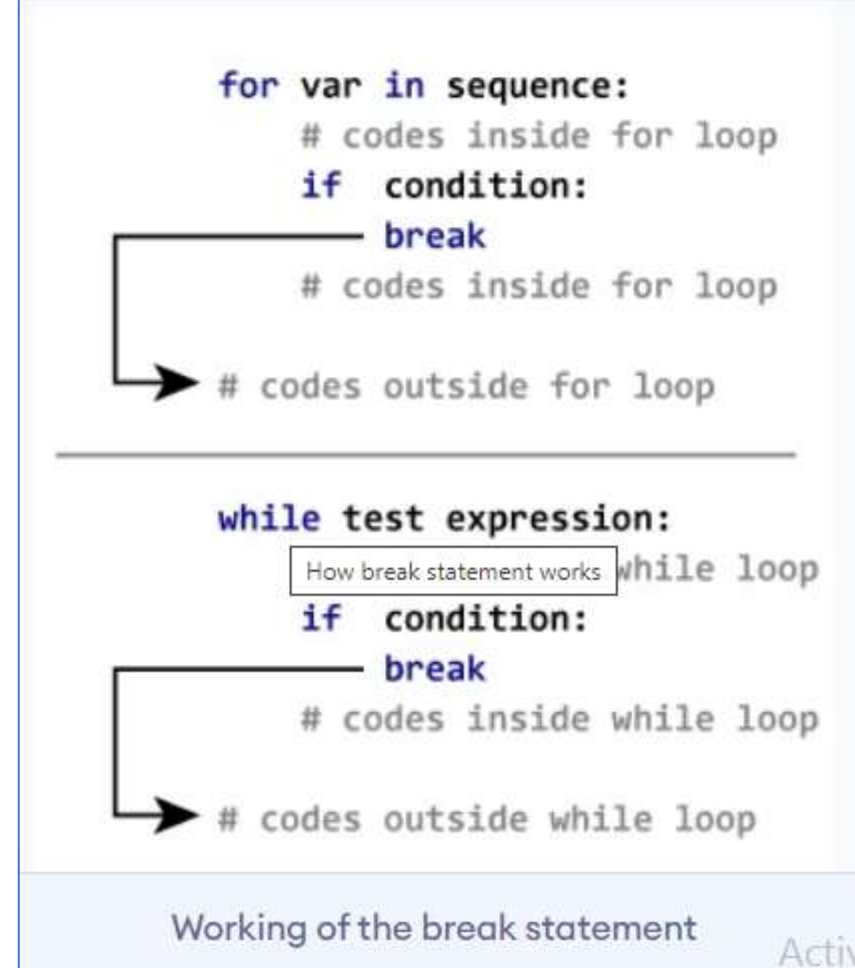
Example:

```
for val in "vaagdevi":
    if val == "d":
        break
    print(val)
print("The  end")
```

main.py Run Shell

```
1 for val in "vaagdevi":
2     if val == "d":
3         break
4     print(val)
5
6 print("The end")
```

```
v
a
a
g
The end
>
```



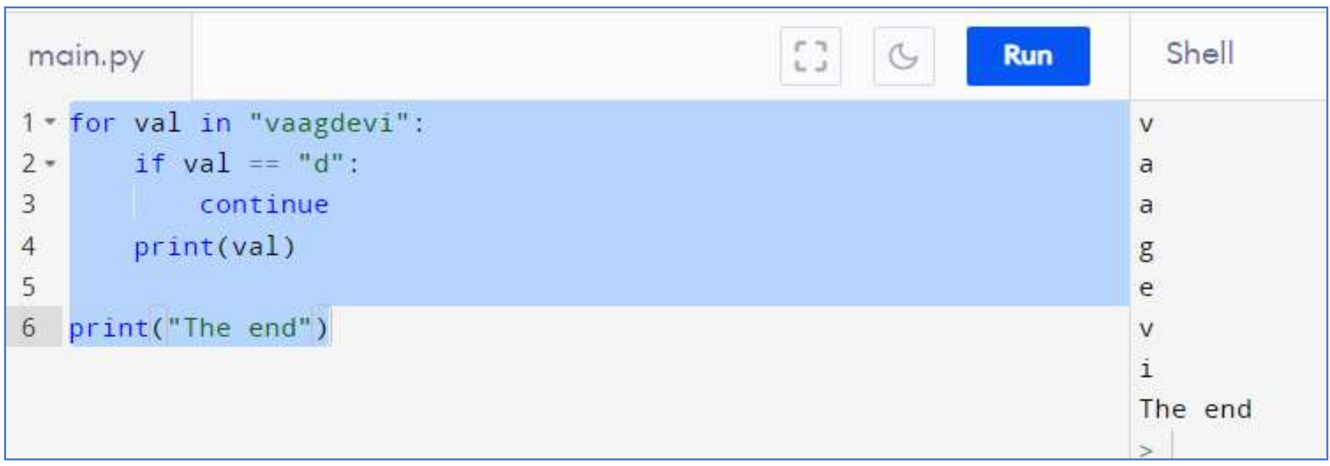
Working of the break statement

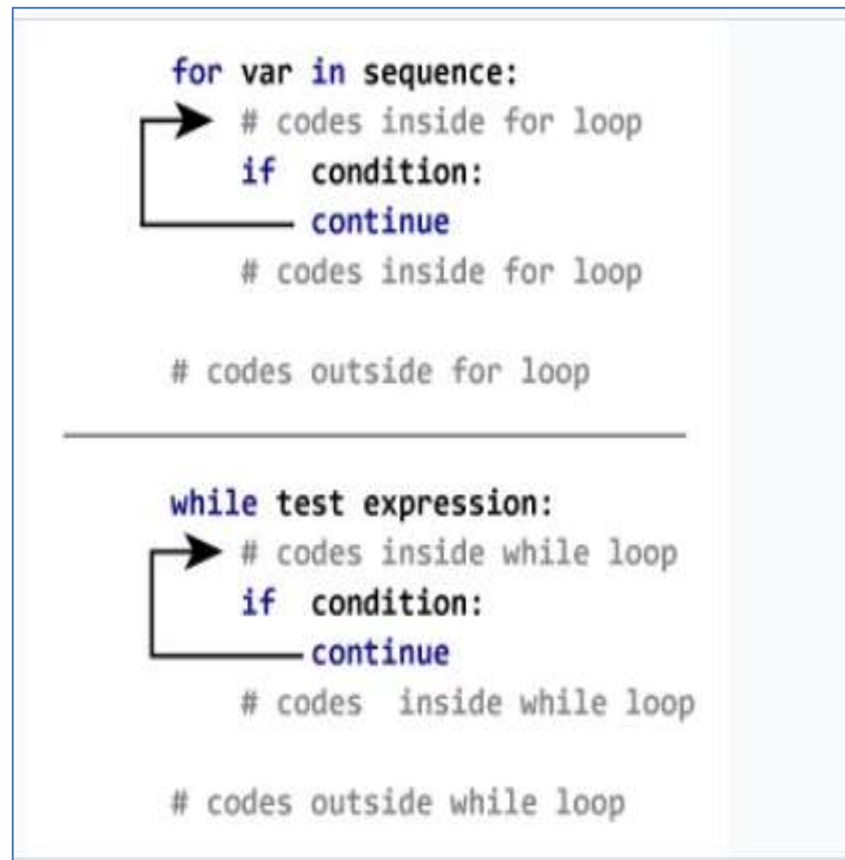
In this program, we iterate through the "vaagdevi" sequence. We check if the letter is d, upon which we break from the loop. Hence, we see in our output that all the letters up till g gets printed. After that, the loop terminates.

Python continue statement

The continue statement is used to skip the rest of the code inside a loop for the current iteration only. Loop does not terminate but continues on with the next iteration.

```
for val in "vaagdevi":
    if val == "d":
        continue
    print(val)
print("The end")
```





Python pass statement

In Python programming, the pass statement is like a null statement. The difference between a comment and a pass statement in Python is that while the interpreter ignores a comment entirely, pass is not ignored.

Why to use pass statement:

For example we want to declare a loop or function in our code but we want to implement that function in future, which means we are not yet ready to write the body of the function. In this case we cannot leave the body of function empty as this would raise error because it is syntactically incorrect, in such cases we can use pass statement which **does nothing** but **makes the code syntactically correct**.

Example:

```
alphabets = {'p', 'a', 's', 's'}
For i in alphabets:
    pass
```

Example:

```
Num = [20, 11, 9, 66, 4, 89, 44]
for  i in Num:
        if num%2 == 0:
                pass
        else:
                print(Num)
```

```
11
9
89
```

Unit 2: Assignment (Submission Date May 19)

1. Explain the different decision control or conditional control statements with syntax, flowchart and example ?
2. Explain what are while loop and for loop in python with syntax and example ?
3. What is nested loop and infinite loop in python ? Explain ?
4. Explain the different transfer statements of python with suitable example ?

Unit 2: Programs Assignment(Sub Date May 25)

1. write a program to find the largest among the three numbers
2. write a program to find roots of quadratic equation
3. Write a program to accept two numbers and mathematical operator and perform operation accordingly.
4. Program to accept any number and find sum of digits of the number
5. Write a program to determine whether a given number is armstrong or not.
6. Write a Program to Check whether a given number is palindrome
7. Write a program to print Fibonacci series for a given number
8. Write a program to check if a number is prime or not
9. Write a program to print pyramid of numbers
10. Write a program using break , continue and pass statements

https://forms.office.com/r/XytA0UfNju

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

PYTHON PROGRAMMING

B.Tech I Year II Sem

Dr Ayesha Banu

Assistant Professor

Department of CSE

Unit – I

Introduction to Python: What is Python?, What is Python Good For?, Python History, How does Python Execute a Program, Review of a Simple Program, Some of the Basic Commands, Variables, Statements, Input/Output Operations, Keywords, Variables, Assigning values, Standard Data Types, Strings, Operands and operators.

Unit – II

Understanding the Decision Control Structures: The if Statement, A Word on Indentation, The if … else Statement, The if … elif … else Statement,

Loop Control Statements: The while Loop, The for Loop, Infinite Loops, Nested Loops.

The break Statement, The continue Statement, The pass Statement, The assert Statement, The return Statement.

Unit – III

Functions- Function Definition and Execution, Scoping, Arguments: Arguments are Objects, Argument Calling by Keywords, Default Arguments, Function Rules, Return Values.

Advanced Function Calling: The apply Statement, The map Statement, Indirect Function Calls.

Unit - IV

Lists: List, Creating List, Updating the Elements of a List, Sorting the List Elements. Storing Different Types of Data in a List, Nested Lists, Nested Lists as Matrices.

Tuples: Creating Tuple, Accessing the Tuple Elements, Basic Operations on Tuples, Functions to Process Tuples, Inserting Elements in a Tuple, Modifying Elements of a Tuple, Deleting Elements from a Tuple.

Sets: Creating Set, Basic Operations on Sets, Methods of Set.

Dictionaries: Operations on Dictionaries, Dictionary Methods, Using for Loop with Dictionaries, Sorting the Elements of a Dictionary using Lambdas, Converting Lists into Dictionary.

Unit – V

Modules: Importing a Module, Tricks for Importing Modules, Packages.

Exceptions and Error Trapping: What is an Exception?, Exception Handling: try..except..else.., try..finally.., Exceptions Nest, Raising Exceptions, Built-In Exceptions.

UNIT - III

Functions- Function Definition and Execution, Scoping, Arguments: Arguments are Objects, Argument Calling by Keywords, Default Arguments, Function Rules, Return Values.
Advanced Function Calling: The apply Statement, The map Statement, Indirect Function Calls.

- Functions are one of the fundamental building blocks in Python programming language.
- Functions are used when we have a block of statements that needs to be executed multiple times within the program.
- Rather than writing the block of statements repeatedly to perform the action, we can use a function to perform that action.
- This block of statements are grouped together and is given a name which can be used to invoke it from other parts of the program.
- Functions also reduce the size of the program by eliminating elementary code.
- Functions help to break a larger program into smaller parts and make it more easy.
- it avoids repetition and makes the code reusable.
- Functions can be either Built-in Functions or User-defined functions.

Built-In Functions

The Python interpreter has a number of functions that are built into it and are always available. The list of the built-in functions are

		Built-in Functions		
abs()	delattr()	hash()	memoryview()	set()
all()	dict()	help()	min()	setattr()
any()	dir()	hex()	next()	slice()
ascii()	divmod()	id()	object()	sorted()
bin()	enumerate()	input()	oct()	staticmethod()
bool()	eval()	int()	open()	str()
breakpoint()	exec()	isinstance()	ord()	sum()
bytearray()	filter()	issubclass()	pow()	super()
bytes()	float()	iter()	print()	tuple()
callable()	format()	len()	property()	type()
chr()	frozenset()	list()	range()	vars()
classmethod()	getattr()	locals()	repr()	zip()
compile()	globals()	map()	reversed()	__import__()
complex()	hasattr()	max()	round()	

Function Name	Syntax	Explanation
abs()	*abs(x)* where x is an integer or floating-point number.	The abs() function returns the absolute value of a number.
min()	*min(arg_1, arg_2, arg_3,…, arg_n)* where arg_1, arg_2, arg_3 are the arguments.	The min() function returns the smallest of two or more arguments.
max()	*max(arg_1, arg_2, arg_3,…,arg_n)* where arg_1, arg_2, arg_3 are the arguments.	The max() function returns the largest of two or more arguments.
divmod()	*divmod(a, b)* where a and b are numbers representing numerator and denominator.	The divmod() function takes two numbers as arguments and return a pair of numbers consisting of their quotient and remainder. For example, if a and b are integer values, then the result is the same as (a // b, a % b). If either a or b is a floating-point number, then the result is (q, a % b), where q is the whole number of the quotient.

pow()	*pow(x, y)* where x and y are numbers.	The pow(x, y) function returns x to the power y which is equivalent to using the power operator: x**y.
len()	*len(s)* where s may be a string, byte, list, tuple, range, dictionary or a set.	The len() function returns the length or the number of items in an object.

Example:

main.py

```
1 print(abs(-3))
2 print(min(5, 2, 1, 4, 3))
3 print(max(4, 8, 6, 7, 2))
4 print(divmod(5, 2))
5 print(divmod(8.5, 3))
6 print(pow(3, 2))
7 print(len("python"))
```

Run

Shell

```
3
1
8
(2, 1)
(2.0, 2.5)
9
6
>
```

User Defined Functions

Functions that readily come with Python are called built-in functions. Functions that we define ourselves to do certain specific task are referred as user-defined functions.

Advantages of user-defined functions

- User-defined functions help to decompose a large program into small segments which makes program easy to understand, maintain and debug.
- If repeated code occurs in a program. Function can be used to include those codes and execute when needed by calling that function.
- In this concept the user only has to define the function and call the function for executing the code inside it.

Function Definition

The syntax for function definition is,

def function_name (parameter_1, parameter_2, …, parameter_n):

statement(s)

In Python, a function definition consists of the *def* keyword, followed by

- The name of the function. The function's name has to adhere to the same naming rules as variables: use letters, numbers, or an underscore, but the name cannot start with a number. Also, you cannot use a keyword as a function name.
- A list of parameters to the function are enclosed in parentheses and separated by commas. Some functions do not have any parameters at all while others may have one or more parameters.

- A colon is required at the end of the function header. The first line of the function definition which includes the name of the function is called the function header.
- Block of statements that define the body of the function start at the next line of the function header and they must have the same indentation level.
- The *def* keyword introduces a function definition. The term parameter or formal parameter is often used to refer to the variables as found in the function definition.

Function Call

- Defining a function does not execute it. Defining a function simply names the function and specifies what to do when the function is called. Calling the function actually performs the specified actions with the indicated parameters.

- The syntax for function call or calling function is,

  function_name(argument_1, argument_2,…,argument_n)

- Arguments are the actual value that is passed into the calling function.
- When a function is called, the formal parameters are temporarily “bound” to the arguments and their initial values are assigned through the calling function.
- When we call a function, the control flows from the calling function to the function definition.
- Once the block of statements in the function definition is executed, then the control flows back to the calling function and proceeds with the next statement.
- Python interpreter keeps track of the flow of control between different statements in the program.

Note: When the control returns to the calling function from the function definition then the formal parameters and other variables in the function definition no longer contain any values.

Creating a Function

In Python a function is defined using the def keyword:

Example

```
def  welcome():
     print(“Welcome to Python Class")
```

Calling a Function

To call a function, use the function name followed by parenthesis:

Example

```
def  welcome():
     print(“Welcome to Python Class")
welcome()
```

main.py Run Shell

```
1 def  welcome():
2     print(“Welcome to Python Class")
3 welcome()
4
```

```
Welcome to Python Class
>
```

Function Parameters or Arguments

The terms parameter and argument can be used for the same thing: information that are passed into a function.

From a function's perspective:

- A parameter is the variable listed inside the parentheses in the function definition.
- An argument is the value that are sent to the function when it is called.

Types of formal arguments

We can call a function by using the following types of formal arguments –

- ➢Default arguments
- ➢Required arguments
- ➢Keyword arguments
- ➢Variable-length arguments

Default Arguments

- In some situations, it might be useful to set a default value to the arguments or parameters of the function definition.
- Each default parameter has a default value as part of its function definition.
- We assign a default value to an argument using the assignment operator in python(=).
- When we call a function without a value for an argument, its default value is used.
- Usually, the default parameters are defined at the end of the parameter list, after any required parameters.
- non-default parameters cannot follow default parameters. The default value is evaluated only once.

main.py

Run

```python
def student(name,age=25):
    print(name)
    print(age)
student(name="rakesh",age=22)
student(name="rahul")
```

Shell

```
rakesh
22
rahul
25
>
```

- In above example age is the default argument with default value 25. when we first call the student function we pass the values for both name and age. Hence they are printed as it is.
- When we call the student function for the second time, we pass only the value for name and in this case the default value for age is considered and printed.

main.py Run

```
def student(age=25,name):
    print(name)
    print(age)
student(name="rakesh",age=22)
student(name="rahul")
```

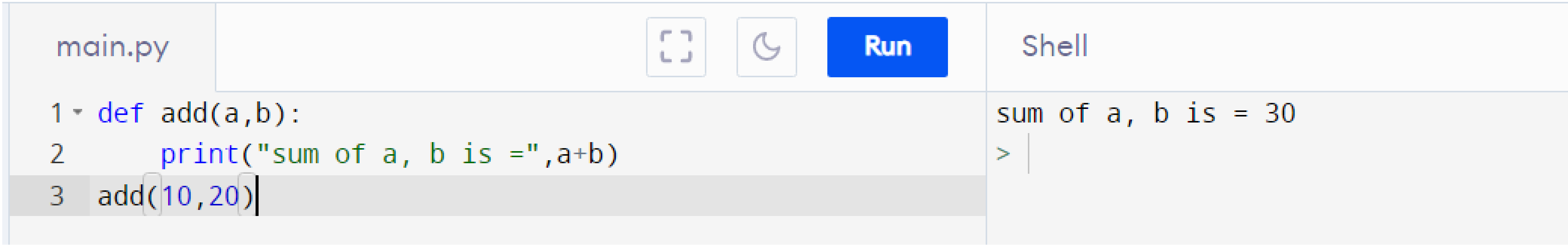
Shell Clear

```
File "<string>", line 1
SyntaxError: non-default argument follows default argument
>
```

- When the non-default parameters follow the default parameters in function definition this will give an error.
- In above example name is the non default argument but it follows the default argument age in the student function definition.
- This gives an error when the program is executed

Required arguments

- These are the arguments passed to a function in correct positional order.
- These arguments are also called as **positional arguments.**
- Here, the number of arguments in the function call should match exactly with the function definition.



- **When no: of arguments do not match it shows error as follows**

main.py

```python
def add(a,b):
    print("sum of a,b =",a+b)
add(10)

```

Shell

```
Traceback (most recent call last):
  File "<string>", line 3, in <module>
TypeError: add() missing 1 required positional argument: 'b'
>
```

Keyword Arguments

- When we call a function with some values, these values get assigned to the arguments according to their position.

Example:

main.py

```python
def student(name,course,college):
    print("Stu Name:=",name)
    print("Stu Course:=",course)
    print("Stu College:=",college)
student("Rakesh","CSE","Vaagdevi")
```

Shell

```
Stu Name:= Rakesh
Stu Course:= CSE
Stu College:= Vaagdevi
>
```

- In above example first value get assigned to argument name.
- Similarly second value is assigned to argument course and third value to argument college.
- This assigning of values is done as per the position or order of the arguments.
- Python allows functions to be called using keyword arguments.
- When we call functions in this way, the order (position) of the arguments can be changed.

main.py

Run

```python
def student(name,course,college):
    print("Stu Name:=",name)
    print("Stu Course:=",course)
    print("Stu College:=",college)
student("Vaagdevi","Rakesh","CSE")
```

Shell

```
Stu Name:= Vaagdevi
Stu Course:= Rakesh
Stu College:= CSE
>
```

- In above example the program show no error syntactically but the meaning of values have changed because the order of values in function call have mismatched.
- Here we can use key word arguments and give values in any order but still get the correct output.

Variable-length arguments

- These arguments are also called as Arbitrary Arguments.
- Python allows us to have the arbitrary or variable number of arguments.
- This is especially useful when we are **not sure in the advance** that how many arguments, the function would require.
- We define the arbitrary arguments while defining a function using the **asterisk (*)** sign.
- An asterisk (*) is placed before the variable name that holds the values of all non keyword variable arguments.
- This tuple remains empty if no additional arguments are specified during the function call.

E & ENGINEERING

main.py

```python
def fruits(*fnames):
    for f in fnames:
        print(f)
fruits("Orange","Banana","Apple","Grapes")
```

Shell

```
Orange
Banana
Apple
Grapes
>
```

- In above example we don't know how many fruit names will be printed or passed while calling the function. In such a case we use the variable length arguments.

main.py

```python
def minval(num1, num2):
    if num1 < num2:
        print("min value=",num1)
    else:
        print("min value=",num2)

minval(23, 50)
```

Shell

```
min value= 23
>
```

- In above program the function takes 2 arguments and prints the minimum value among the two.
- But when we want to pass multiple values and print the minimum among them we can use the variable length arguments.

main.py

```
1 def minval(*num):
2     min = num[0]
3     for i in num:
4         if i < min:
5             min = i
6     print("min value:=",min)
7
8 minval(4, 1, 6, 7, 2)
```

Shell

```
min value:= 1
>
```

- A function definition can have both formal arguments and also variable length arguments. This tuple remains empty if no additional arguments are specified during the function call.

- In above example the first function call does not pass any variable length arguments, hence the tuple remains empty and only the formal argument value 10 is printed.
- The second function call passes more number of arguments and here both formal and variable length argument values 70 60 50 all are printed.

Variable-length keyword arguments

- Python can accept variable length *keyword arguments also* better known as **kwargs.
- It behaves similarly to *args, but stores the arguments in a dictionary instead of tuples:

main.py　　Run　　Shell

```
1 def kwarg_type_test(**kwargs):
2     print(kwargs)
3
4 kwarg_type_test(a="hi")
5 kwarg_type_test(roses="red", violets="blue")
```

```
{'a': 'hi'}
{'roses': 'red', 'violets': 'blue'}
>
```

Python return statement

- A return statement is used to end the execution of the function call and return the result of the computation done to the calling function.
- The return statement is mostly the last statement in the function body.
- The statements after the return statement are not executed.
- If the return statement is without any expression, then the special value None is returned.
- In simple words the return keyword is used to exit a function and return a value.

The syntax for *return* statement is,

```
def fun():
    statements . .
    return [expression]
```

- The *return* statement terminates the execution of the function definition in which it appears and returns control to the calling function. It can also return an optional value to its calling function.

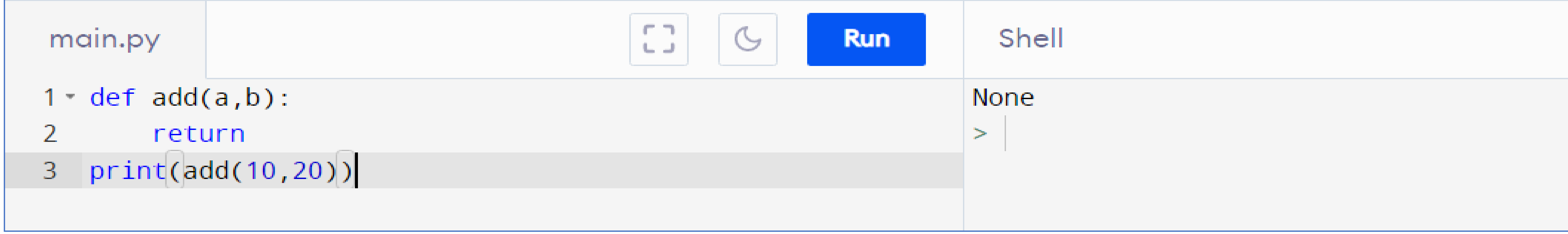
main.py

```python
def add(a,b):
    return a+b
print(add(10,20))
```

Shell

```
30
>
```

When the return statement has no expression then it returns None value



The statements after the return statement are not executed.

main.py

```python
def add(a,b):
    return a+b
    print("hello python")
print(add(10,20))
```

Shell

```
30
>
```

- A function can return only a single value, but that value can be a list or tuple.
- In some cases, we may return multiple values if they are related to each other.
- In such case, return the multiple values separated by a comma which by default is constructed as a tuple by Python.

main.py

```python
def fun(x):
    y0 = x + 1
    y1 = x * 3
    y2 = y0 ** y1
    return (y0, y1, y2)
print(fun(2))
```

Shell

```
(3, 6, 729)
>
```

- In Python, it is possible to define functions without a *return* statement. Functions like this are called **void functions**, and they return *None*.

Scope and Lifetime of Variables

- variables are the containers for storing data values.
- The location where we can find a variable and also access it if required is called the **scope of a variable**.
- **Local Scope:** A variable created inside a function is called local variable and belongs to the *local scope* of that function, and can only be used inside that function.

main.py

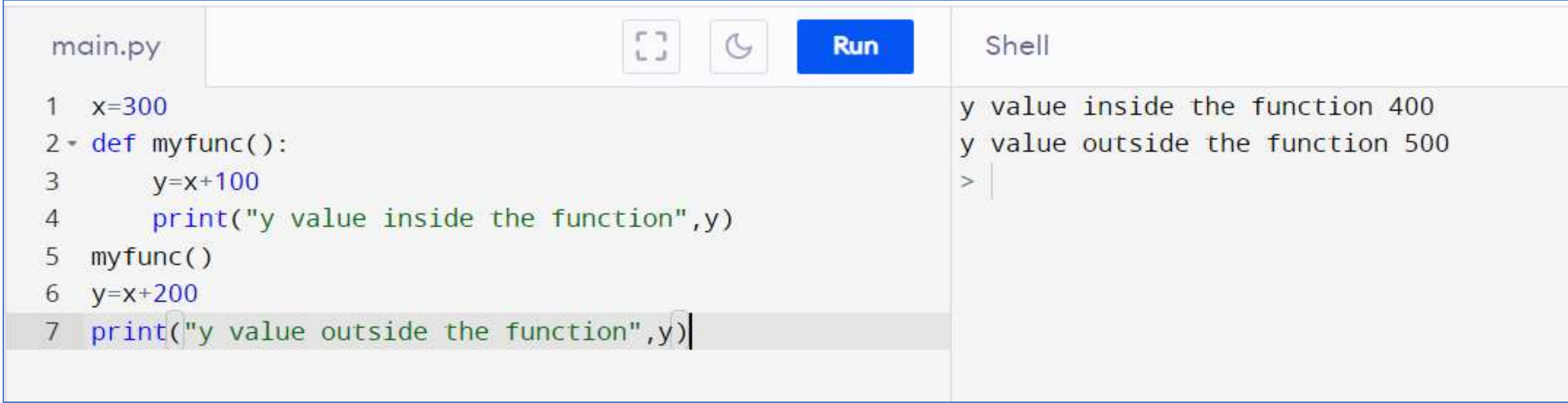
```
1 def myfunc():
2   x = 300
3   print(x)
4 myfunc()
5 print(x)
```

Shell

```
300
Traceback (most recent call last):
  File "<string>", line 6, in <module>
NameError: name 'x' is not defined
>
```

- In above example x is a local variable and its scope is confined to the function definition only.
- When we try to access this variable outside the function(in the last line), we get an error because variable x is out of its local scope.

- **Global Scope:** Variables that are defined outside the function body are called global variables and they have global scope.
- global variables can be accessed throughout the program body by all functions.



- Here x is a global variable which is declared outside the function body.
- X can be accessed inside the function (y=x+100) and also outside the function (y=x+200).
- The scope of variable x is global.

- The lifetime of a variable refers to the amount of time or duration of the variable’s existence i.e how long does the variable exists with its assigned value to be accessed.
- The local variable is created and destroyed every time the function is executed and its life time exist as long as the function is executing.
- It is possible to access global variables from inside a function, as long as you have not defined a local variable with the same name.
- A local variable can have the same name as a global variable, but they are totally different so changing the value of the local variable has no effect on the global variable.
- The local variable has meaning only inside the function in which it is defined.

Python program for swapping two numbers using functions

main.py

Run

Shell

```
def swapping(a,b):
    print("a,b values before swapping=",a,b)
    temp=a
    a=b
    b=temp
    print("a,b values after swapping=",a,b)
swapping(10,20)
```

```
a,b values before swapping= 10 20
a,b values after swapping= 20 10
>
```

python program using functions to find LCM of two numbers

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

main.py

Run

Shell

```
def lcm(x, y):
    if x > y:
        max = x
    else:
        max = y
    while(True):
        if((max % x == 0) and (max % y == 0)):
            lcm = max
            break
        max += 1
    return lcm
num1 = int(input("Enter first number: "))
num2 = int(input("Enter second number: "))
print("The L.C.M. of", num1,"and", num2,"is", lcm(num1, num2))
```

```
Enter first number: 12
Enter second number: 18
The L.C.M. of 12 and 18 is 36
>
```

Activate Windo
Go to PC settings to

program to perform the arithmetic operations using the functions to each operation (Record Program)

main.py | Run | Shell

```
1  def add(x, y):
2      return x + y
3  def subtract(x, y):
4      return x - y
5  def multiply(x, y):
6      return x * y
7  def divide(x, y):
8      return x / y
9  print("Select operation.")
10 print("1.Add")
11 print("2.Subtract")
12 print("3.Multiply")
13 print("4.Divide")
14 choice = input("Enter choice(1/2/3/4):")
15
```

```
Select operation.
1.Add
2.Subtract
3.Multiply
4.Divide
Enter choice(1/2/3/4):3
Enter first number: 10
Enter second number: 10
10 * 10 = 100
>
```

```
16  num1 = int(input("Enter first number: "))
17  num2 = int(input("Enter second number: "))
18
19  if choice == '1':
20      print(num1,"+",num2,"=", add(num1,num2))
21
22  elif choice == '2':
23      print(num1,"-",num2,"=", subtract(num1,num2))
24
25  elif choice == '3':
26      print(num1,"*",num2,"=", multiply(num1,num2))
27  elif choice == '4':
28      print(num1,"/",num2,"=", divide(num1,num2))
29  else:
30      print("Invalid input")
```

Write a program to define functions to calculate the area of a circle, reactangle and square using default arguments **(Record Program)**

main.py

```
def cirarea(radius=5):
    area=3.14*radius*radius
    print("Area of circle=",area)
cirarea()
def rectarea(len=10,brd=5):
    area=len*brd
    print("Area of rectangle=",area)
rectarea()
def sqarea(side=6):
    area=side*side
    print("Area of square=",area)
sqarea()
```

Shell

```
Area of circle= 78.5
Area of rectangle= 50
Area of square= 36
>
```

Write a program to define a function to display the grade of a student by using positional arguments (rno, sub1, sub2, sub3) Rec Program

```
1  def stugrade(rollno,sub1,sub2,sub3):
2          total=sub1+sub2+sub3
3          avg=total/3
4          print("Rollno:=",rollno)
5          print("Sub1 marks:=",sub1)
6          print("Sub2 marks:=",sub2)
7          print("Sub3 marks:=",sub3)
8          print("Total marks:=",total)
9          print("Average:=",avg)
10         if avg>=90:
11             print("Grade:= O")
12         elif avg>=80:
13             print("Grade:= A+")
14         elif avg>=70:
15             print("Grade:= A")
16         elif avg>=60:
17             print("Grade:= B+")
18         elif avg>=50:
19             print("Grade:= B")
20         elif avg>=40:
21             print("Grade:= C")
22         else:
23             print("Grade:= F")
24 stugrade(1007,75,80,85)
```

```
Rollno:= 1007
Sub1 marks:= 75
Sub2 marks:= 80
Sub3 marks:= 85
Total marks:= 240
Average:= 80.0
Grade:= A+
>
```

main.py

```python
def shopping(cname,milk,sugar,soaps,pulses):
        print("enter price of milk packet")
        p1=int(input())
        print("enter price of 1 kg sugar")
        p2=int(input())
        print("enter price of 1 soap")
        p3=int(input())
        print("enter price of 1 kg pulses")
        p4=int(input())
        bill1=milk*p1
        bill2=sugar*p2
        bill3=soaps*p3
        bill4=pulses*p4
        totalbill=bill1+bill2+bill3+bill4
        print("**************************************************")
        print("customer name:=",cname)
        print()
        print("**************************************************")
        print("item name\tno: of items\tprice\tbill")
        print("**************************************************")
        print("milk \t\t\t",milk,"\t\t\t",p1,"\t\t\t",bill1)
        print("sugar\t\t\t",sugar,"\t\t\t",p2,"\t\t\t",bill2)
        print("soaps\t\t\t",soaps,"\t\t\t",p3,"\t\t\t",bill3)
        print("pulses\t\t\t",pulses,"\t\t\t",p4,"\t\t\t",bill4)
        print("**************************************************")
        return totalbill
bill=shopping(cname="raju",soaps=4,pulses=3,milk=4,sugar=5)
print("Total Bill:=", bill)
print("**************************************************")
```

```
enter price of milk packet
21
enter price of 1 kg sugar
35
enter price of 1 soap
28
enter price of 1 kg pulses
68
**************************************************
customer name:= raju

**************************************************
item name   no: of items    price   bill
**************************************************
milk            4           21          84
sugar           5           35          175
soaps           4           28          112
pulses          3           68          204
**************************************************
Total Bill:= 575
**************************************************
```

Activate Windows
Go to PC settings to activ

Python Nested Functions

- A function defined inside the body of another function is called a nested function.
- There are both outer function and inner function

main.py

```
def outfun(x):
    print("outer function x value=",x)
    def infun(y):
        print("inner function y value=",y)
        print("x * y value =",x*y)
    infun(10)
outfun(5)
```

Shell

```
outer function x value= 5
inner function y value= 10
x * y value = 50
>
```

main.py

Run

```
def grade():
    avg=0
    def total(s1,s2,s3):
        return s1+s2+s3
    avg=(total(60,75,80))/3
    print(avg)
    def result():
        if avg>=90:
            print("O+")
        elif avg>=80:
            print("A+")
        elif avg>=70:
            print("B+")
        else:
            print("F")
    result()
grade()
```

Shell

```
71.66666666666667
B+
>
```

Activate Windo
Go to PC settings to

Recursive Functions in Python

- In Python, a function can call other functions.
- It is even possible for the function to call itself.
- These types of construct are termed as recursive functions.

```
def recurse():
    ...
    recurse()
    ...

recurse()
```

recursive call

Recursive Function in Python

recursive function to find the factorial of an integer.

main.py

```
1 def factorial(x):
2     if x == 1:
3         return 1
4     else:
5         return (x * factorial(x-1))
6 num = 5
7 print("The factorial of", num, "is", factorial(num))
```

Shell

```
The factorial of 5 is 120
>
```

Program to find reverse a number using Recursion (Rec Prog)

main.py

```
Rev = 0
def Reverse(N):
    global Rev
    if(N > 0):
        Rem = N %10
        Rev = (Rev *10) + Rem
        Reverse(N //10)
    return Rev

N = int(input("Please Enter any Number: "))
Rev = Reverse(N)
print("\n Reverse of entered number is = ",Rev)
```

Shell

```
Please Enter any Number: 123

 Reverse of entered number is =  321
>
```

program to convert a decimal number to binary number using recursive function.(Rec Prog)

main.py

Run

```
1 def Binary(n):
2     if n > 1:
3         Binary(n//2)
4     print(n % 2,end = '')
5
6 dec = int(input("Enter the Decimal Number"))
7 print("the binary value is")
8 Binary(dec)
9 print()
```

Shell

```
Enter the Decimal Number12
the binary value is
1100
>
```

Program to find sum of digits of a number using Recursion

main.py

```
sum=0
def sumofdig(n):
    global sum
    if n > 0:
        rem=n%10
        sum=sum+rem
        sumofdig(n//10)
    return sum
n=int(input("enter the number"))
sum=sumofdig(n)
print("the number is",n)
print("sum of digits of the number is",sum)
```

Shell

```
enter the number123
the number is 123
sum of digits of the number is 6
>
```

Indirect Function Call

When we define any functions in python we can call the function for execution in two ways

- Direct function call
- Indirect function call

Example:1

```
def add(a,b):
        c=a+b
        print("a+b value=",c)
add(10,20)
```

Output:

```
a+b value= 30
```

This way of calling the function is direct function call

Example:1

```
def add(a,b):
        c=a+b
        print("a+b value=",c)
add(10,20)
z=add
z(15,25)
```

Output:

```
        a+b value= 30
        a+b value= 40
```

Here declaring a variable z and assigning the name of the function will support in calling the same function indirectly as z(15,25)

we can also use the following notation for indirect function calling

```
def add(a,b):
        c=a+b
        print("a+b value=",c)
z=add
def indirect(func,arg1,arg2):
        print(func(arg1,arg2))
indirect(z,20,30)
```

Output:

a+b value= 50

1. Write a Python function to find the Max of three numbers.
2. Write a Python function to check whether a number is in a given range.
3. Write a Python function that takes a number as a parameter and check the number is prime or not

Unit - IV

- Lists: List, Creating List, Updating the Elements of a List, Sorting the List Elements. Storing Different Types of Data in a List, Nested Lists, Nested Lists as Matrices.
- Tuples: Creating Tuple, Accessing the Tuple Elements, Basic Operations on Tuples, Functions to Process Tuples, Inserting Elements in a Tuple, Modifying Elements of a Tuple, Deleting Elements from a Tuple.
- Sets: Creating Set, Basic Operations on Sets, Methods of Set.
- Dictionaries: Operations on Dictionaries, Dictionary Methods, Using for Loop with Dictionaries, Sorting the Elements of a Dictionary using Lambdas, Converting Lists into Dictionary.

SETS

- A set is a collection which is both *unordered* and *unindexed*.
- A set is an unordered collection with no duplicate items.
- Primary uses of sets include membership testing and eliminating duplicate entries.
- Sets also support mathematical operations, such as union, intersection, difference, and symmetric difference.
- Curly braces **{ }** or the set() function can be used to create sets with a comma-separated list of items inside curly brackets **{ }**.
- Note: to create an empty set you have to use set() and not **{ }** as the latter creates an empty dictionary.

Creating a set:

- A set is created by using the set() function or placing all the elements within a pair of curly braces.

Ex-1:

```
thisset = set(["apple","banana","cherry"])
print(thisset)
```

Ex-2:

```
thisset = {"apple", "banana", "cherry"}
print(thisset)
```

Output: {'banana', 'apple', 'cherry'}

Note: the set list is unordered, meaning: the items will appear in a random order.

Sets are unordered, so you cannot be sure in which order the items will appear.

Set Items:

Set items are unordered, unchangeable, and do not allow duplicate values.

Unordered:

Unordered means that the items in a set do not have a defined order.

Set items can appear in a different order every time you use them, and cannot be referred to by index or key.

Unchangeable:

Sets are unchangeable, meaning that we cannot change the items after the set has been created.

Once a set is created, you cannot change its items, but you can add new items.

Duplicates Not Allowed:

Sets cannot have two items with the same value.

Example

Duplicate values will be ignored:

```
thisset = {"apple", "banana", "cherry", "apple"}
print(thisset)
```

Output:

{'apple', 'cherry', 'banana'}

len():To determine how many items a set has, use the len() method.

Example:

Get the number of items in a set:

```
thisset = {"apple", "banana", "cherry"}
print(len(thisset))
```

Set Items - Data Types:

Set items can be of any data type:

Example

```
set1 = {"apple", "banana", "cherry"}
set2 = {1, 5, 7, 9, 3}
set3 = {True, False, False}
```

A set can contain different data types:

Example

```
set1 = {"abc", 34, True, 40, "male"}
```

Access Items:

You cannot access items in a set by referring to an index or a key.

But you can loop through the set items using a for loop, or ask if a specified value is present in a set, by using the in keyword.

Example

Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}
for x in thisset:
  print(x)
```

Output:

```
apple
banana
cherry
```

- Check if "banana" is present in the set:
- thisset = {"apple", "banana", "cherry"}
  print("banana" in thisset)

output: True

- Add Items
- Once a set is created, you cannot change its items, but you can add new items.
- To add one item to a set use the add() method.
- Example

```
thisset = {"apple", "banana", "cherry"}
thisset.add("orange")
print(thisset)
```

Output:

{'banana', 'cherry', 'apple', 'orange'}

Add Sets:

- To add items from another set into the current set, use the update() method.
- Example
- Add elements from set2 into set1:
- set1 = {"apple", "banana", "cherry"}
  set2 = {"pineapple", "mango", "papaya"}
  set1.update(set2)
  print(set1)
  output:

{'apple', 'mango', 'cherry', 'pineapple', 'banana', 'papaya'}

Add Any Iterable:

- The object in the update() method does not have to be a set, it can be any iterable object (tuples, lists, dictionaries etc.).
- Example
- Add elements of a list to at set:
- thisset = {"apple", "banana", "cherry"}
  mylist = ["kiwi", "orange"]
  thisset.update(mylist)
  print(thisset)

Output:

{'banana', 'cherry', 'apple', 'orange', 'kiwi'}

Remove Item:

- To remove an item in a set, use the remove(), or the discard() method.
- Example
- Remove "banana" by using the remove() method:
- ```
  thisset = {"apple", "banana", "cherry"}
  thisset.remove("banana")
  print(thisset)
  ```
- **Note:** If the item to remove does not exist, remove() will raise an error.

- Remove "banana" by using the discard() method:
- ```
  thisset = {"apple", "banana", "cherry"}
  thisset.discard("banana")
  print(thisset)
  ```

Note: If the item to remove does not exist, discard() will **NOT** raise an error.

- Remove the last item by using the pop() method:

```
thisset = {"apple", "banana", "cherry"}
x = thisset.pop()
print(x)
print(thisset)
```

Output: apple
{'banana', 'cherry'}

Note: Sets are *unordered*, so when using the pop() method, you do not know which item that gets removed.

- The clear() method empties the set:

```
thisset = {"apple", "banana", "cherry"}
thisset.clear()
print(thisset)
```

Output: thisset()

The del keyword will delete the set completely:

```
thisset = {"apple", "banana", "cherry"}
del thisset
print(thisset)
```

output:

```
Traceback (most recent call last):
  File "demo_set_del.py", line 5, in <module>
    print(thisset) #this will raise an error because the set no longer
exists
NameError: name 'thisset' is not defined
```

Loop Items:

You can loop through the set items by using a for loop:

Example:

Loop through the set, and print the values:

```
thisset = {"apple", "banana", "cherry"}

for x in thisset:
    print(x)
```

Output:

```
banana
apple
cherry
```

Join Two Sets:

- There are several ways to join two or more sets in Python.
- You can use the **union()** method that returns a new set containing all items from both sets
- Example
- The union() method returns a new set with all items from both sets:
- set1 = {"a", "b" , "c"}
  set2 = {1, 2, 3}
  set3 = set1.union(set2)
  print(set3)

Output: {'c', 'b', 3, 1, 2, 'a'}

```
x = {"a", "b", "c"}
y = {"f", "d", "a"}
z = {"c", "d", "e"}

result = x.union(y, z)

print(result)
```

Python Set intersection() Method

Return a set that contains the items that exist in both set x, and set y:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
z = x.intersection(y)
print(z)
```

The intersection() method returns a set that contains the similarity between two or more sets.

```
x = {"a", "b", "c"}
y = {"c", "d", "e"}
z = {"f", "g", "c"}
result = x.intersection(y, z)
print(result)
```

Keep All, But NOT the Duplicates:

- The symmetric_difference_update() method will keep only the elements that are NOT present in both sets.

Example: Keep the items that are not present in both sets:

```
x = {"apple", "banana", "cherry"}
y = {"google", "microsoft", "apple"}
x.symmetric_difference_update(y)
print(x)
```

Output:

{'google', 'banana', 'microsoft', 'cherry'}

Set Methods

Python has a set of built-in methods that you can use on sets.

Method	Description
add()	Adds an element to the set
clear()	Removes all the elements from the set
copy()	Returns a copy of the set
difference()	Returns a set containing the difference between two or more sets
difference_update()	Removes the items in this set that are also included in another, specified set
discard()	Remove the specified item
intersection()	Returns a set, that is the intersection of two other sets
intersection_update()	Removes the items in this set that are not present in other, specified set(s)
isdisjoint()	Returns whether two sets have a intersection or not

issubset()	Returns whether another set contains this set or not
issuperset()	Returns whether this set contains another set or not
pop()	Removes an element from the set
remove()	Removes the specified element
symmetric_difference()	Returns a set with the symmetric differences of two sets
symmetric_difference_update()	inserts the symmetric differences from this set and another
union()	Return a set containing the union of sets
update()	Update the set with the union of this set and others

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

PYTHON PROGRAMMING

B.Tech I Year II Sem

Dr Ayesha Banu

Assistant Professor

Department of CSE

Unit – I

Introduction to Python: What is Python?, What is Python Good For?, Python History, How does Python Execute a Program, Review of a Simple Program, Some of the Basic Commands, Variables, Statements, Input/Output Operations, Keywords, Variables, Assigning values, Standard Data Types, Strings, Operands and operators.

Unit – II

Understanding the Decision Control Structures: The if Statement, A Word on Indentation, The if … else Statement, The if … elif … else Statement,

Loop Control Statements: The while Loop, The for Loop, Infinite Loops, Nested Loops.

The break Statement, The continue Statement, The pass Statement, The assert Statement, The return Statement.

Unit – III

Functions- Function Definition and Execution, Scoping, Arguments: Arguments are Objects, Argument Calling by Keywords, Default Arguments, Function Rules, Return Values.

Advanced Function Calling: The apply Statement, The map Statement, Indirect Function Calls.

Unit - IV

Lists: List, Creating List, Updating the Elements of a List, Sorting the List Elements. Storing Different Types of Data in a List, Nested Lists, Nested Lists as Matrices.

Tuples: Creating Tuple, Accessing the Tuple Elements, Basic Operations on Tuples, Functions to Process Tuples, Inserting Elements in a Tuple, Modifying Elements of a Tuple, Deleting Elements from a Tuple.

Sets: Creating Set, Basic Operations on Sets, Methods of Set.

Dictionaries: Operations on Dictionaries, Dictionary Methods, Using for Loop with Dictionaries, Sorting the Elements of a Dictionary using Lambdas, Converting Lists into Dictionary.

Unit – V

Modules: Importing a Module, Tricks for Importing Modules, Packages.

Exceptions and Error Trapping: What is an Exception?, Exception Handling: try..except..else.., try..finally.., Exceptions Nest, Raising Exceptions, Built-In Exceptions.

Data Types

Data types specify the type of data like numbers and characters to be stored and manipulated within a program.

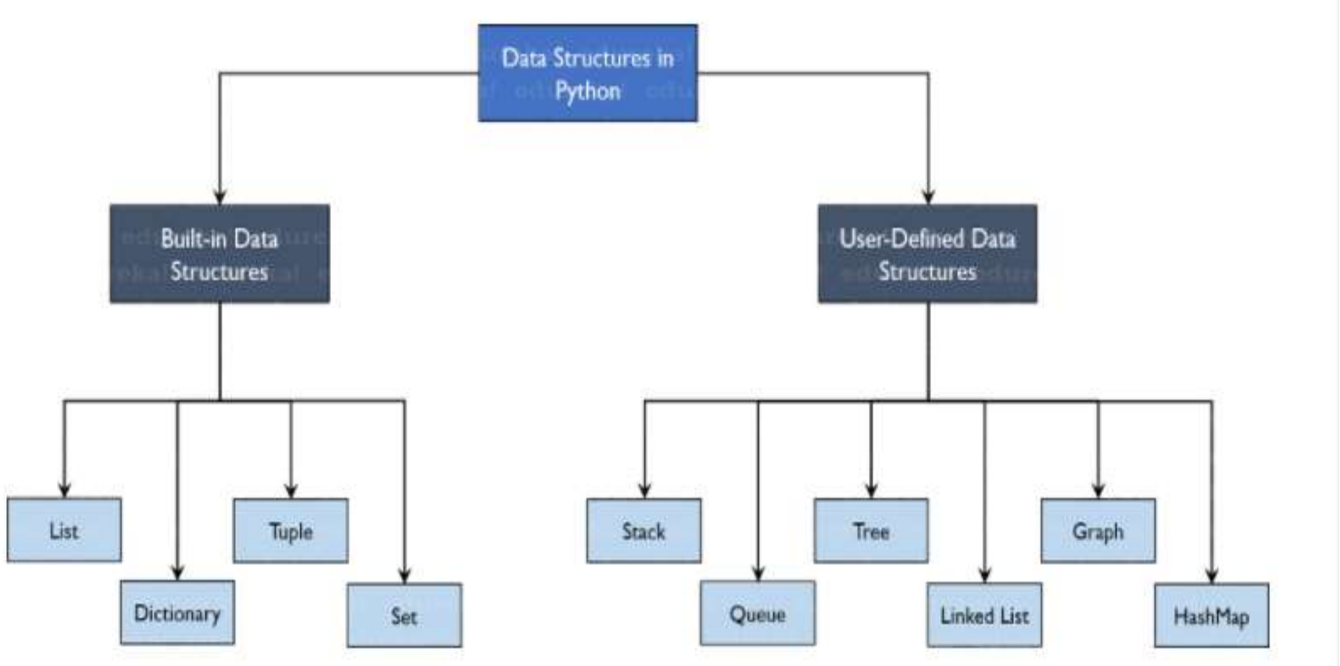
- Numeric Types: int, float, complex
- Boolean Type: bool
- Text Type: str
- Special data type: None
- Sequence Types: list, tuple, range
- Mapping Type: dict
- Set Types: set, frozenset

- Python offers a range of compound /collection data types often referred to as sequences (list, tuple, range, Set, Dictionary)
- List isone of the most frequently used and very versatile data types used in Python.
- List is a collection which is ordered and changeable (mutable) and also allows duplicate values.
- Tuple is a collection which is ordered and unchangeable (Un mutable) and also allows duplicate values.
- Set is a collection which is unordered and un indexed and do not allow duplicate values.
- Dictionary is a collection which is ordered and changeable with no duplicate values.
- When choosing a compound data type, it is important to understand the properties of that data type.
- These compound data types are also called as built in data structures of python

What is a Data Structure?

- Organizing, managing and storing data is important as it enables easier access and efficient modifications.
- Data Structures allows to organize the data in such a way that it enables us to store collections of data, relate them and perform operations on them accordingly.

Types of Data Structures in Python



1. LIST

- Most of the times a variable can hold only single value.
- However, in many cases, we may need to assign more than one value.
- List is the most frequently used data structure to store multiple items in a single variable.
- Lists are used to store data of different data types in a sequential manner.
- There are addresses assigned to every element of the list, which is called as Index.
- The index value starts from 0 and goes on until the last element called the positive index.
- There is also negative indexing which starts from -1 enabling you to access elements from the last to first.
- List items are ordered, changeable, and allow duplicate values.
- List items are indexed, the first item has index [0], the second item has index [1] etc.
- When we say that lists are **ordered,** it means that the items have a defined order, and that order will not change.
- If you add new items to a list, the new items will be placed at the end of the list.

- The list is **changeable**, meaning that we can change, add, and remove items in a list after it has been created.
- To change the value of a specific item, refer to the index number:

Example

```
a = ["apple", "banana", "cherry"]
print(a)
a[1] = "mango"
print(a)
```

Output

```
['apple', 'banana', 'cherry']
['apple', 'mango', 'cherry']
```

- List **Allow Duplicates** that means lists can have items with the same value:

Example:

```
a = ["apple", "banana", "cherry", "apple", "cherry"]
print(a)
```

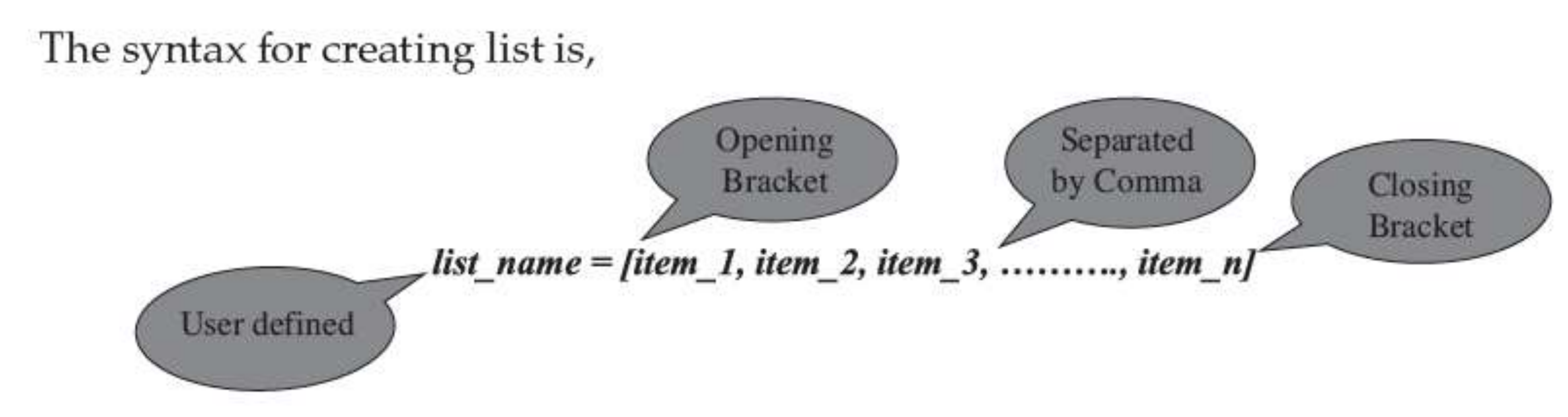
Output:

```
['apple', 'banana', 'cherry', 'apple', 'cherry']
```

- **Therefore a list is ordered, changeable and also allow duplicate values.**

2. Creating a list

- To create a list, we use the square brackets and add elements into the list separated by commas.
- If we do not pass any elements inside the square brackets, we get an empty list as the output.




```
list1 = []        #create empty list
print(list1)
list2 = [1, 2, 3, 'example', 3.132]       #creating list with data
print(list2)
```

Output:

[ ]

[1, 2, 3, ‘example’, 3.132]

List Items - Data Types:

List items can be of any data type:

Example:

```
list1 = ["apple", "banana", "cherry"]   # all  strings
list2 = [1, 5, 7, 9, 3]  #  all numeric
list3 = [True, False, False]  #  all boolean
print(list1)
print(list2)
print(list3)
```

Output:

```
['apple', 'banana', 'cherry']
[1, 5, 7, 9, 3]
[True, False, False]
```

A list can contain different data types:

```
list1 = ["abc", 34, True, 40, "male"]
print(list1)
```

Output

```
['abc', 34, True, 40, 'male']
```

List Length

To determine how many items a list has, use the len() function:

Example:

```
list1 = ["abc", 34, True, 40, "male"]
print(len(list1))
```

Adding Items to List

Add Items at end of the list

- To add an item to the end of the list, use the append() method:

Example

```
a= ["apple", "banana", "cherry“]
print(a)
a.append("orange")
print(a)
```

Output

```
['apple', 'banana', 'cherry']
['apple', 'banana', 'cherry', 'orange']
```

Add an item at the specified index

- To add an element at a specified index we use the insert() method:

Example

```
a= ["apple", "banana", "cherry"]
print(a)
a.insert(1,"orange")
print(a)
```

Output

```
['apple', 'banana', 'cherry']
['apple', 'orange', 'banana', 'cherry']
```

Deleting Items from List

There are several methods to remove items from a list:

Remove any specified item

The remove( ) method can be used to remove any specified item:

Example

```
a= ["apple", "banana", "cherry"]
print(a)
a.remove("banana")
print(a)
```

Output

```
['apple', 'banana', 'cherry']
['apple', 'cherry']
```

Remove item from specified index

The **pop( )** method is used to removes item from specified index.

if index is not specified the last item will be removed from the list.

Example

```
a= ["apple", "banana", "cherry", "grapes", "mango"]
print(a)
a.pop(1)    #removes item at index 1
print(a)
a.pop()     #removes the last item
print(a)
```

Output

```
['apple', 'banana', 'cherry', 'grapes', 'mango']
['apple', 'cherry', 'grapes', 'mango']
['apple', 'cherry', 'grapes']
```

The **del** keyword can also be used to remove item from the specified index.

If the index is not specified then the entire list is removed completely

Example

```
a= ["apple", "banana", "cherry", "grapes", "mango"]
print(a)
del a[2]    #removes item at index 2
print(a)
del a    #removes the entire list
print(a)
```

Output

```
['apple', 'banana', 'cherry', 'grapes', 'mango']
['apple', 'banana', 'grapes', 'mango']
NameError: name 'a' is not defined
```

del keyword can also be used to delete range of items

Example

```
a= ["apple", "banana", "cherry", "grapes", "mango", "pine apple", "orange"]
print(a)
del a[2:5]    #removes item from index 2 to 4
print(a)
```

Output

['apple', 'banana', 'cherry', 'grapes', 'mango', 'pine apple', 'orange']

['apple', 'banana', 'pine apple', 'orange']

The **clear()** method empties the list:

Example:

```
a = ["apple", "banana", "cherry"]
print(a)
a.clear()
print(a)
```

Output

['apple', 'banana', 'cherry']

[ ]

Accessing Items of a List

We can Access the items of a list by referring to the index number:

Example

Print the second item of the list:

```
a = ["apple", "banana", "cherry"]
print(a[1])
```

Output

```
banana
```

Negative Indexing

Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second last item etc.

Example

Print the last item of the list:

```
a= ["apple", "banana", "cherry"]
print(a[-1])
```

Access a range of items

- We can also access an range of items from a list by using a range of indexes by specifying where to start and where to end the acess.
- When specifying a range, the return value will be a new list with the specified items.

Example

Return the third, fourth, and fifth item:

```
a= ["apple", "banana", "cherry", "orange", "kiwi", "melon", "mango"]
print(a[2:5])
```

Output

```
['cherry', 'orange', 'kiwi']
```

Note: The search will start at index 2 but index 5 is not included in the range.

Changing item values in the List

Change single Item Value

To change the value of a specific item we refer to the index number:

Example

```
a = ["apple", "banana", "cherry"]
print(a)
a[1] = "mango"
print(a)
```

Output

```
['apple', 'banana', 'cherry']
['apple', 'mango', 'cherry']
```

Change a Range of Item Values:

- To change the value of items within a specific range,
- define a list with the new values, and
- refer to the range of index numbers where we want to insert the new values:

Example:

Change the values "banana" and "cherry" with the values “carrot" and “tomato":

```
a = ["apple","banana","cherry","orange","kiwi","mango"]
a[1:3] = ["carrot","tomato"]
print(a)
```

Output:

```
['apple', 'carrot', 'tomato', 'orange', 'kiwi', 'mango']
```

If we try to give more values than the specified range then , the new items will be inserted in the specified range, and the remaining items will move forward accordingly:

Example:

```
a = ["apple","banana","cherry","orange","kiwi","mango"]
a[1:2] = ["pineapple","grapes","papaya"]
print(a)
```

Output:

```
['apple', 'pineapple', 'grapes', 'papaya', 'cherry', 'orange', 'kiwi', 'mango']
```

If we try to give less values than the specified range then the given value is inserted throughout the range and the remaining items will move accordingly:

Example:

```
a = ["apple","banana","cherry","banana","grapes","papaya"]
a[1:3] = ["watermelon"]
print(a)
a[1:4] = ["watermelon","pineapple"]
print(a)
```

Output:

```
['apple', 'watermelon', 'banana', 'grapes', 'papaya']
['apple', 'watermelon', 'pineapple', 'grapes', 'papaya']
```

extend() method

This method append elements of one list at the end of the other .

Example

```
vaag1 = ["CSE","CSM","CSD"]
vaag2 = ["EEE","ECE","MECH","CIVIL"]
vaag1.extend(vaag2)
print(vaag1)
```

Output:

```
['CSE', 'CSM', 'CSD', 'EEE', 'ECE', 'MECH', 'CIVIL']
```

Note: The elements will be added to the *end* of the list1.

The extend() method not only appends *lists but it* can append any iterable object like (tuples, sets, dictionaries etc.)

Example

```
vaag1 = ["CSE","CSM","CSD"]
vaag2 = ("EEE","ECE","MECH","CIVIL")
print(type(vaag1))
print(type(vaag2))
vaag1.extend(vaag2)
print(vaag1)
```

Output:

```
<class 'list'>
<class 'tuple'>
['CSE', 'CSM', 'CSD', 'EEE', 'ECE', 'MECH', 'CIVIL']
```

Python List Methods	
Method	**Description**
append()	Adds an element at the end of the list
clear()	Removes all the elements from the list
copy()	Returns a copy of the list
count()	Returns the number of elements with the specified value
extend()	Add the elements of a list (or any iterable), to the end of the current list
index()	Returns the index of the first element with the specified value
insert()	Adds an element at the specified position
pop()	Removes the element at the specified position
remove()	Removes the item with the specified value
reverse()	Reverses the order of the list
sort()	Sorts the list

➢ **copy() method:** This method returns an exact true copy of the list.

A list can be copied using the = operator.

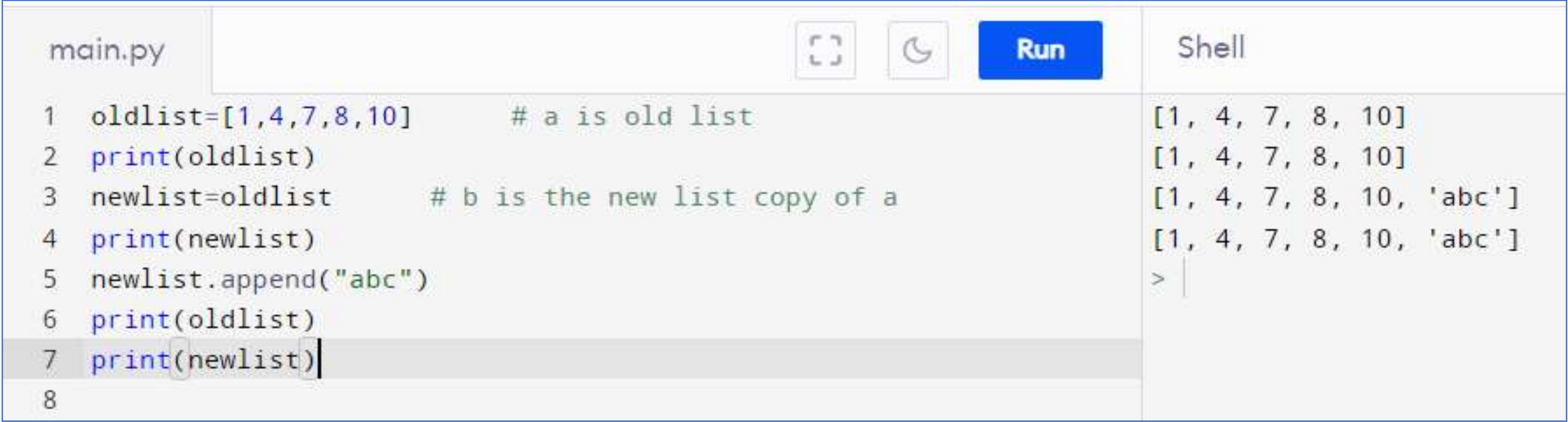
For example,

```
oldlist=[1,4,7,8,10]
print(oldlist)
newlist=oldlist
print(newlist)
```

Output:

```
[1, 4, 7, 8, 10]
[1, 4, 7, 8, 10]
```

- The problem with copying lists in this way is that if we modify newlist then oldlist is also modified.
- This is because the newlist is referencing or pointing to the same oldlist object.



- However, when we need the oldlist unchanged when the newlist is modified, we can use the copy() method.
- The syntax of the copy() method is:

  newlist = oldlist.copy()

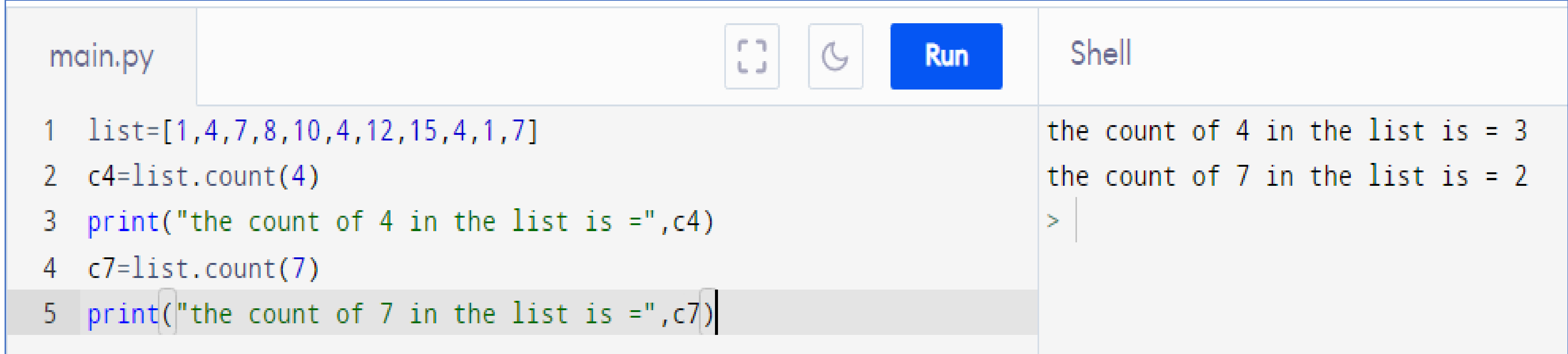
- The copy() method doesn't take any parameters.
- The copy() method returns a new list. It doesn't modify the original list.

```
main.py                                        Run    Shell
1 oldlist=[1,4,7,8,10]      # a is old list        [1, 4, 7, 8, 10]
2 print(oldlist)                                   [1, 4, 7, 8, 10]
3 newlist=oldlist.copy()      # b is the new list copy of a   [1, 4, 7, 8, 10]
4 print(newlist)                                   [1, 4, 7, 8, 10, 'abc']
5 newlist.append("abc")                            >
6 print(oldlist)
7 print(newlist)
8
```

OF
IENCE & ENGINEERING

- **count():** This method returns the number of times the specified element appears in the list.
- The syntax of the count() method is:

list.count(element)



➢**index() method:** This method returns the index of the specified element in the list.

- The syntax of the list index() method is:

    list.index(element, start, end)

- **element** - the element to be searched
- **start** (optional) - start searching from this index
- **end** (optional) - search the element up to this index
- The index() method returns the index of the given element in the list.
- If the element is not found, a ValueError exception is raised.
- **Note:** The index() method only returns the first occurrence of the matching element.

```
1  list=[1,4,7,8,10,4,12,15,4,1,7]
2  i=list.index(7)
3  print("position of 7 in the list=",i)
4  j=list.index(18)
5  print("position of 18 in the list=",j)
```

```
position of 7 in the list= 2
Traceback (most recent call last):
  File "<string>", line 4, in <module>
ValueError: 18 is not in list
>
```

index() With Start and End Parameters

main.py

```
1  # alphabets list
2  alphabets = ['a', 'e', 'i', 'o', 'g', 'l', 'i', 'u']
3
4  # index of 'i' in alphabets
5  index = alphabets.index('e')   # first index position
6  print('The index of e:', index)
7
8  # 'i' after the 4th index is searched
9  index = alphabets.index('i', 4)
10 print('The index of i:', index)
11
12 # 'i' between 3rd and 5th index is searched
13 index = alphabets.index('i', 3, 5)   # Error!
14 print('The index of i:', index)
15
```

Shell

```
The index of e: 1
The index of i: 6
Traceback (most recent call last):
  File "<string>", line 13, in <module>
ValueError: 'i' is not in list
>
```

➢ **reverse() method:** This method reverses the elements of the list.

- The syntax of the reverse() method is:

```
list.reverse()
```

Example

```
vaag=["CSE","CSM","CSD","EEE","ECE"]
print(vaag)
vaag.reverse()
print(vaag)
```

Output

```
['CSE', 'CSM', 'CSD', 'EEE', 'ECE']
['ECE', 'EEE', 'CSD', 'CSM', 'CSE']
```

➢ **sort() method:** method sorts the elements of a given list in a specific ascending or descending order.

- The syntax of the sort() method is:

  list.sort(reverse=...)

- when we do not use the reverse parameter then the list is sorted in ascending order.
- If we use the parameter as reverse=True then the list is sorted in descending order.

main.py

```
1 vowels=['i','a','o','e','u']
2 print("the original list=",vowels)
3 vowels.sort()
4 print("sorted list ascending order=",vowels)
5 vowels.sort(reverse=True)
6 print("sorted list descending order=",vowels)
```

Run

Shell

```
the original list= ['i', 'a', 'o', 'e', 'u']
sorted list ascending order= ['a', 'e', 'i', 'o', 'u']
sorted list descending order= ['u', 'o', 'i', 'e', 'a']
>
```

Customize Sort Function

- We can also customize our own function by using the keyword argument key = *function*.
- The function will return a number that will be used to sort the list (the lowest number first)

Example: Sort the list based on how close the number is to 50:

```
def myfunc(n):
  return abs(n - 50)

list = [100, 50, 65, 82, 23]
list.sort(key = myfunc)
print(list)
```

Output

```
[50, 65, 23, 82, 100]
```

- By default the sort() method is case sensitive, resulting in all capital letters being sorted before lower case letters:

Example

```
list = ["banana", "Orange", "Kiwi", "cherry"]
list.sort()
print(list)
```

Output:

```
['Kiwi', 'Orange', 'banana', 'cherry']
```

- We can customize the sort and print lower case first

main.py   Run   Shell

```
1 list = ["banana","Orange","Kiwi","cherry"]
2 list.sort(key=str.lower)
3 print(list)
4
```

```
['banana', 'cherry', 'Kiwi', 'Orange']
>
```

Loop through a list :Python - Loop Lists

We can loop or iterate through the list items by using a for loop:

Example: Print all items in the list, one by one:

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

Loop Through the Index Numbers:

We can also loop through the list items by referring to their index number. Here we use the range() and len() functions.

Example: Print all items by referring to their index number:

```
list = ["apple", "banana", "cherry"]
for i in range(len(list)):
        print(list[i])
```

Using a While Loop

- we can loop through the list items by using a while loop.
- We can use the len() function to determine the length of the list, then start at 0 and loop through the list items by referring to their indexes. increase the index by 1 after each iteration.

Example:

main.py

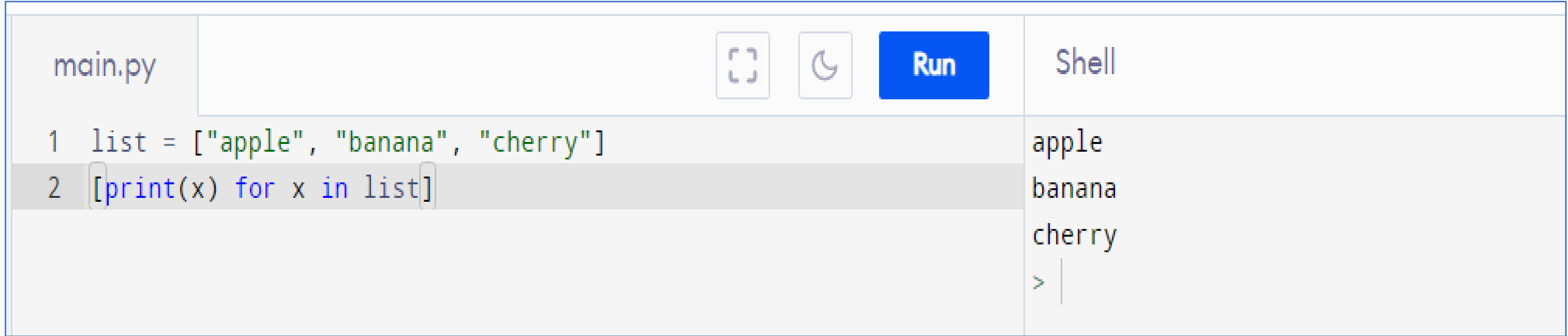
```
list = ["apple", "banana", "cherry"]
i = 0
while i < len(list):
  print(list[i])
  i = i + 1
```

Shell

```
apple
banana
cherry
>
```

Looping Using List Comprehension

List Comprehension offers the shortest syntax for looping through lists. This is also called as the short hand for loop that will print all items in a list.



Join Two Lists

- There are several ways to join, or concatenate, two or more lists in Python.
- One of the easiest way is by using the + operator.

Example:

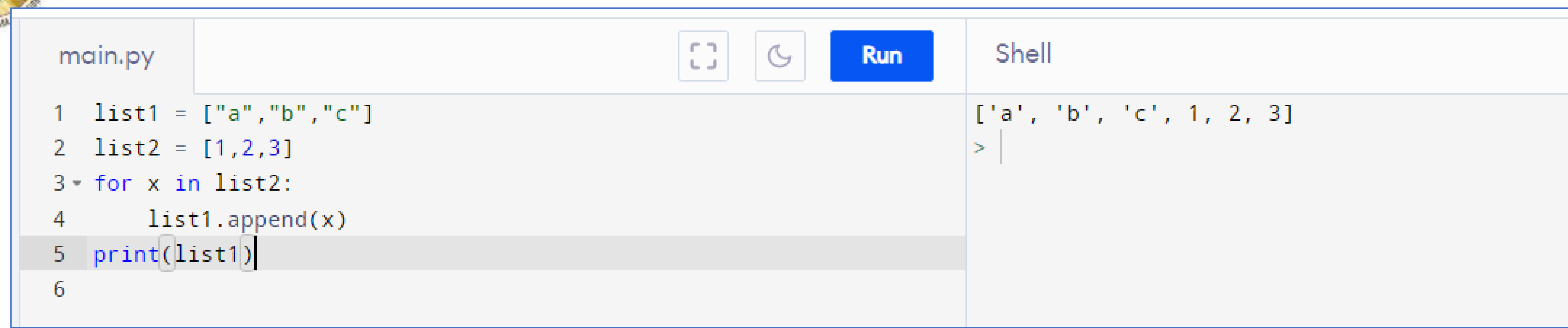
main.py

```
1  list1 = ["a", "b" , "c"]
2  list2 = [1, 2, 3]
3  list3 = list1 + list2
4  print(list3)
5
```

Shell

```
['a', 'b', 'c', 1, 2, 3]
>
```

- Another way to join two lists is by appending all the items from list2 into list1, one by one using loops.



- we can also use the extend() method to add elements from one list to another list:

main.py

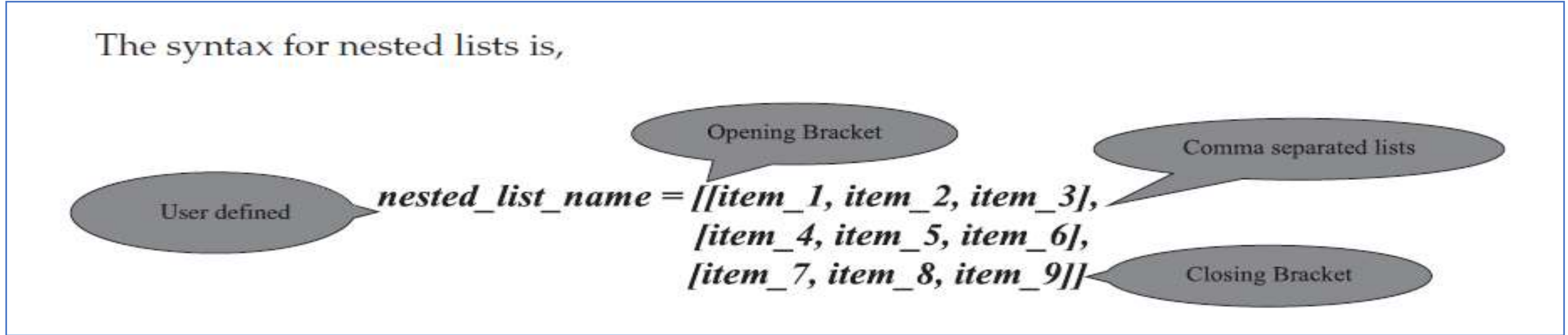
```
list1 = ["a","b","c"]
list2 = [1,2,3]
list1.extend(list2)
print(list1)
```

Shell

```
['a', 'b', 'c', 1, 2, 3]
```

Nested Lists

- A list inside another list is called a nested list.



- Each list inside another list is separated by a comma. Each list inside the nested list can be accessed using the index value.

main.py

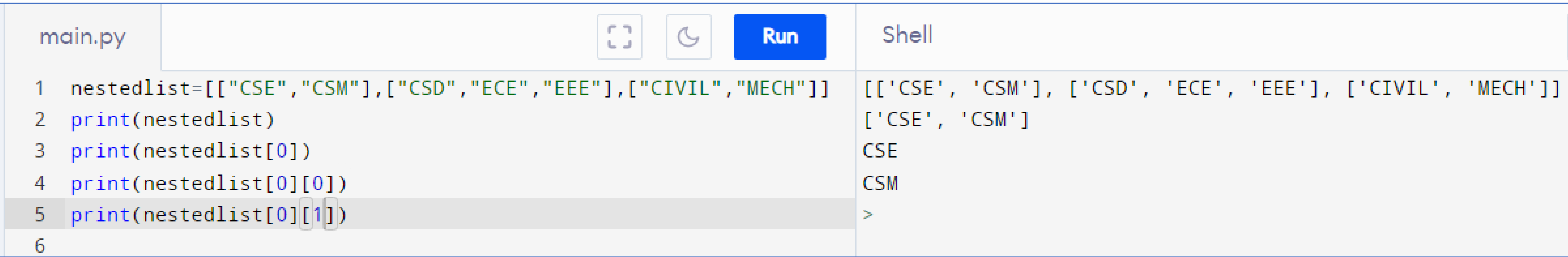
```
1  nestedlist=[["CSE","CSM"],["CSD","ECE","EEE"],["CIVIL","MECH"]]
2  print(nestedlist)
3  print(nestedlist[0])
4  print(nestedlist[1])
5  print(nestedlist[2])
6
```

Run

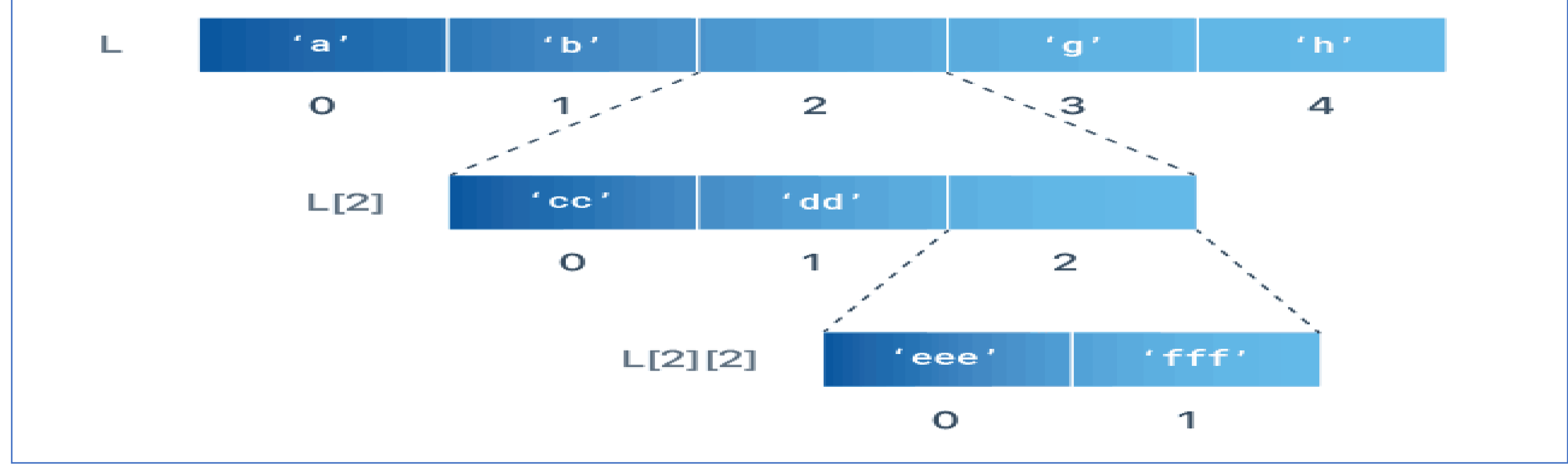
Shell

```
[['CSE', 'CSM'], ['CSD', 'ECE', 'EEE'], ['CIVIL', 'MECH']]
['CSE', 'CSM']
['CSD', 'ECE', 'EEE']
['CIVIL', 'MECH']
>
```

We can access the items of the sub list by using matrix index form.

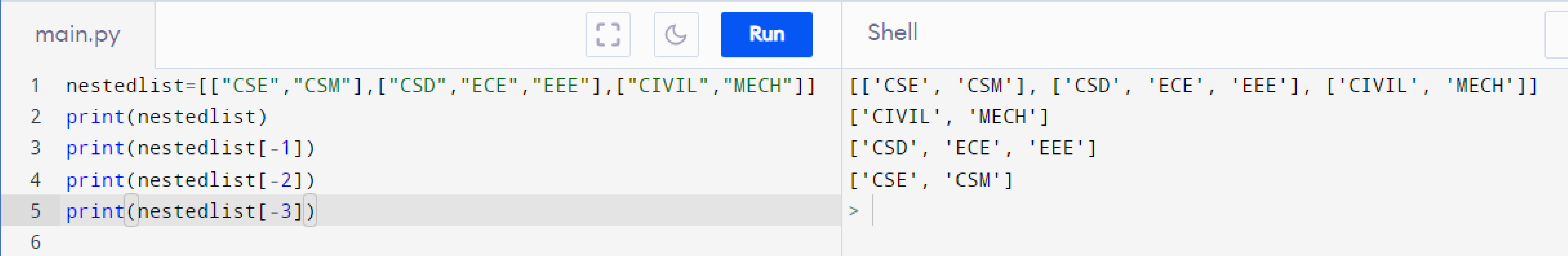


The indexes for the items in a nested list are illustrated as below:



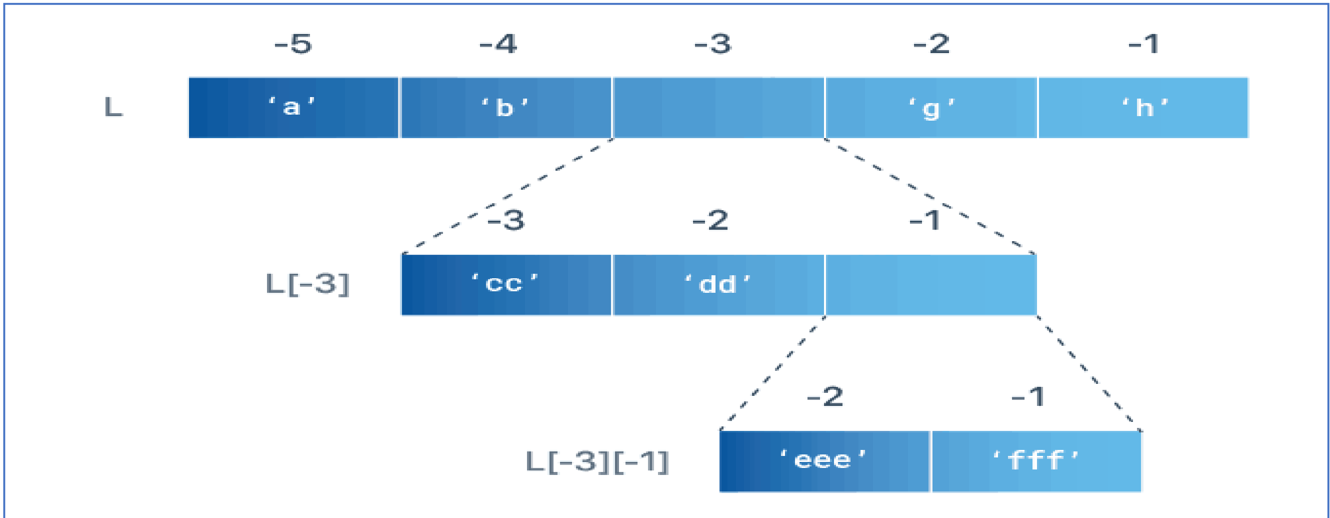
Negative List Indexing In a Nested List:

We can access a nested list by negative indexing also. Negative indexes count backward from the end of the list. So, L[-1] refers to the last item, L[-2] is the second-last, and so on.

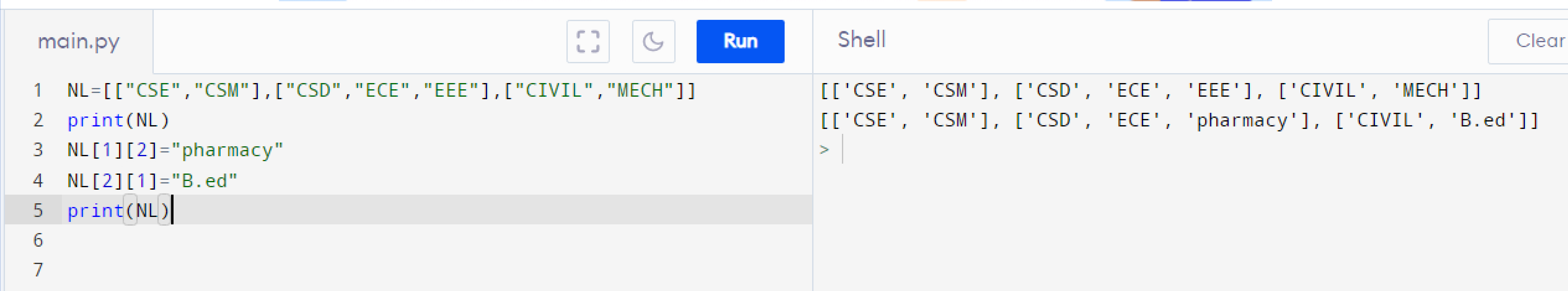


```
main.py
1 nestedlist=[["CSE","CSM"],["CSD","ECE","EEE"],["CIVIL","MECH"]]
2 print(nestedlist)
3 print(nestedlist[-1])
4 print(nestedlist[-2])
5 print(nestedlist[-3])
6
```

```
Shell
[['CSE', 'CSM'], ['CSD', 'ECE', 'EEE'], ['CIVIL', 'MECH']]
['CIVIL', 'MECH']
['CSD', 'ECE', 'EEE']
['CSE', 'CSM']
>
```



Change Nested List Item Value: we can change same like in list.



Add items to a Nested list:

To add new values to the end of the nested list we can use append() method same like in list

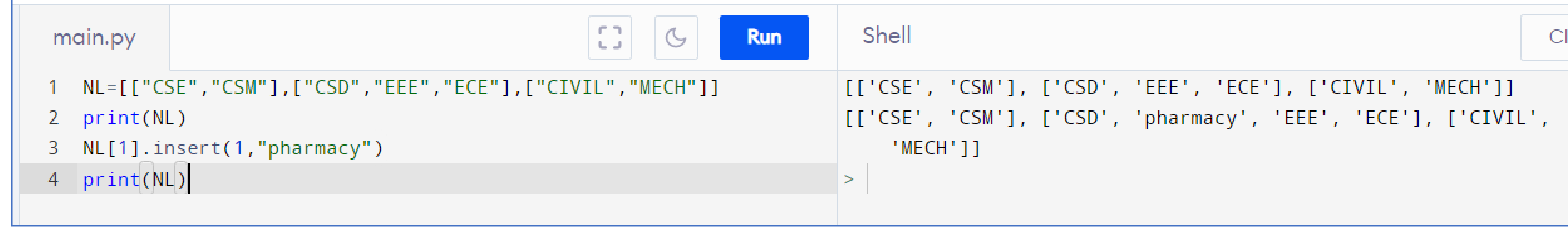
main.py

```
1 NL=[["CSE","CSM"],["CSD","ECE","EEE"],["CIVIL","MECH"]]
2 print(NL)
3 NL[1].append("pharmacy")
4 print(NL)
5
```

Shell

```
[['CSE', 'CSM'], ['CSD', 'ECE', 'EEE'], ['CIVIL', 'MECH']]
[['CSE', 'CSM'], ['CSD', 'ECE', 'EEE', 'pharmacy'], ['CIVIL',
    'MECH']]
>
```

We can also insert an item at a specific position in a nested list, use insert() method.



we can merge one list into another by using extend() method.

main.py

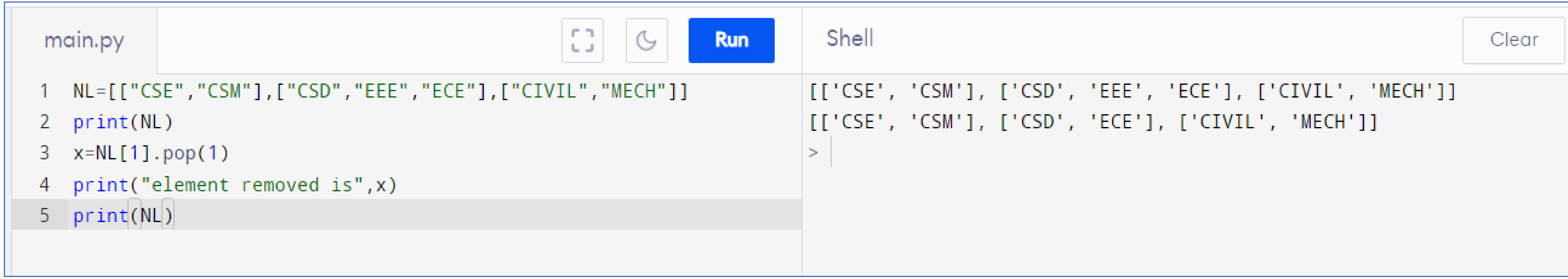
```
NL=[["CSE","CSM"],["CSD","EEE","ECE"],["CIVIL","MECH"]]
print(NL)
NL[1].extend([1,2,3])
print(NL)
```

Shell

```
[['CSE', 'CSM'], ['CSD', 'EEE', 'ECE'], ['CIVIL', 'MECH']]
[['CSE', 'CSM'], ['CSD', 'EEE', 'ECE', 1, 2, 3], ['CIVIL', 'MECH']]
```

Remove items from a Nested List:

If we know the index of the item to be removed, we can delete the item using pop() method.



If no index value is given then the last element is removed or popped from the sub list

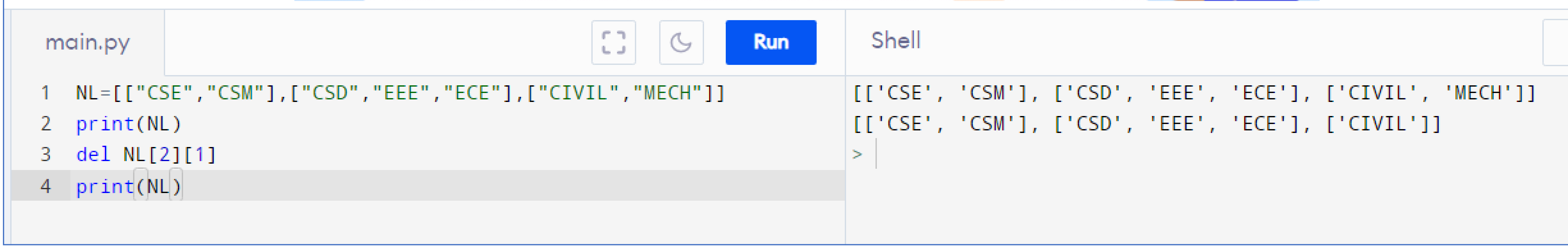
main.py

```
NL=[["CSE","CSM"],["CSD","EEE","ECE"],["CIVIL","MECH"]]
print(NL)
x=NL[1].pop()
print("element removed is",x)
print(NL)
```

Shell

```
[['CSE', 'CSM'], ['CSD', 'EEE', 'ECE'], ['CIVIL', 'MECH']]
element removed is ECE
[['CSE', 'CSM'], ['CSD', 'EEE'], ['CIVIL', 'MECH']]
```

We can also use the del statement to remove the value from the nested list. In this case the removed element value cannot be printed.



If the second index is not specified in del statement then the entire sub list is removed completely and If no index is specified then the entire nested list is removed completely

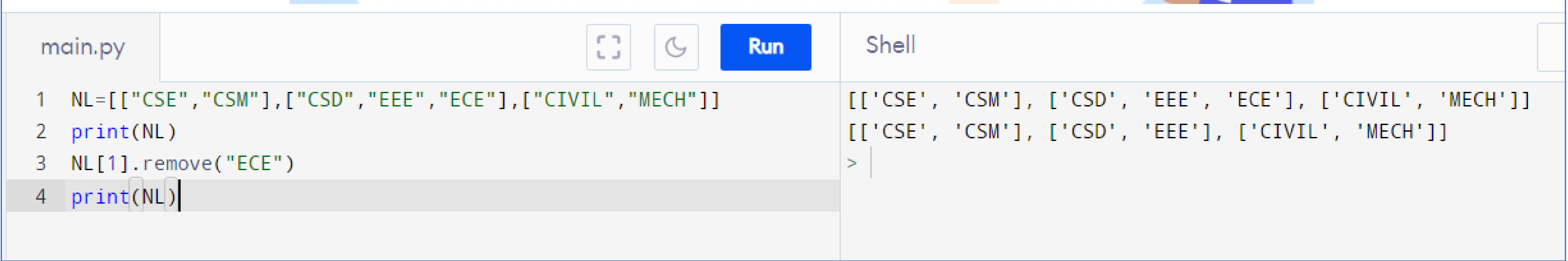
main.py

```
1 NL=[["CSE","CSM"],["CSD","EEE","ECE"],["CIVIL","MECH"]]
2 print(NL)
3 del NL[1]
4 print(NL)
5 del NL
6 print(NL)
```

Shell

```
[['CSE', 'CSM'], ['CSD', 'EEE', 'ECE'], ['CIVIL', 'MECH']]
[['CSE', 'CSM'], ['CIVIL', 'MECH']]
Traceback (most recent call last):
  File "<string>", line 6, in <module>
NameError: name 'NL' is not defined
>
```

When we are not sure about the index position of the item but we know the name of the item to be removed then we use remove() method to delete it by value.



Find Nested List Length:

main.py

```
1 NL=[["CSE","CSM"],["CSD","EEE","ECE"],["CIVIL","MECH"]]
2 print(NL)
3 print(len(NL))
4 print(len(NL[2]))
```

Shell

```
[['CSE', 'CSM'], ['CSD', 'EEE', 'ECE'], ['CIVIL', 'MECH']]
3
2
>
```

Iterate through a Nested List:

To iterate over the items of a nested list, use simple for loop.

main.py

```
1 NL=[["CSE","CSM"],["CSD","EEE","ECE"],["CIVIL","MECH"]]
2 print(NL)
3 for value in NL:
4     print(value)
5 for value in NL[1]:
6     print(value)
```

Shell

```
[['CSE', 'CSM'], ['CSD', 'EEE', 'ECE'], ['CIVIL', 'MECH']]
['CSE', 'CSM']
['CSD', 'EEE', 'ECE']
['CIVIL', 'MECH']
CSD
EEE
ECE
>
```

Nested List as a Matrix

- In Python, we can implement a matrix as a nested list (list inside a list). We can treat each element as a row of the matrix.
- For example X = [[1, 2], [4, 5], [3, 6]] would represent a 3x2 matrix. First row can be selected as X[0] and the element in first row, first column can be selected as X[0][0].
- The elements of the matrix can be accessed using the index values

main.py

```
list=[[1,2,3],[4,5,6],[7,8,9]]
print(list)
print("first row of matrix=",list[0])
print("second row of matrix=",list[1])
print("third row of matrix=",list[2])
print("first row second column element",list[0][1])
print("second row second column element",list[1][1])
print("third row second column element",list[2][1])
```

Shell

```
[[1, 2, 3], [4, 5, 6], [7, 8, 9]]
first row of matrix= [1, 2, 3]
second row of matrix= [4, 5, 6]
third row of matrix= [7, 8, 9]
first row second column element 2
second row second column element 5
third row second column element 8
>
```

We can also take the matrix element values from user as input

main.py

```
A = []
print("Enter 4 Elements for Matrix A: ")
for i in range(2):
    A.append([])
    for j in range(2):
        num = int(input())
        A[i].append(num)
print("\nMatrix A is:")
for i in range(2):
    for j in range(2):
        print(A[i][j], end=" ")
    print()
```

Shell

```
Enter 4 Elements for Matrix A:
5


8
9
6


Matrix A is:
5 8
9 6
>
```

Here number of rows and columns of the matrix are fixed as 2x2 matrix

We can also accept the no of rows and no of columns from user along with matrix element values.

main.py

```
A = []
print("enter no of rows and no of columns")
r=int(input())
c=int(input())
print("Enter rxc Elements for Matrix A: ")
for i in range(r):
    A.append([])
    for j in range(c):
        num = int(input())
        A[i].append(num)
print("\nMatrix A is:")
for i in range(r):
    for j in range(c):
        print(A[i][j], end=" ")
    print()
```

Shell

```
enter no of rows and no of columns
3
2
Enter rxc Elements for Matrix A:
1
2
3
4
5
6
Matrix A is:
1 2
3 4
5 6
>
```

Program to transpose a matrix using a nested list

```
A = []
print("enter no of rows and no of columns")
r=int(input())
c=int(input())
print("Enter rxc Elements for Matrix A: ")
for i in range(r):
    A.append([])
    for j in range(c):
        num = int(input())
        A[i].append(num)
```

```
print("\nMatrix A is:")
for i in range(r):
        for j in range(c):
                print(A[i][j], end=" ")
        print()

print("\ntranspose of Matrix A is:")
for i in range(c):
        for j in range(r):
                print(A[j][i], end=" ")
        print()
```

Python Program to Add Two Matrices

```
A = []
B = []
C = []
print("enter no of rows and no of columns of matrix A")
r1=int(input())
c1=int(input())
print("Enter rxc Elements for Matrix A: ")
for i in range(r1):
    A.append([])
    for j in range(c1):
        num = int(input())
        A[i].append(num)
print("\nMatrix A is:")
for i in range(r1):
    for j in range(c1):
        print(A[i][j], end=" “)
    print()
```

```
print("enter no of rows and no of columns of matrix B")
r2=int(input())
c2=int(input())
print("Enter rxc Elements for Matrix A: ")
for i in range(r2):
    B.append([])
    for j in range(c2):
        num = int(input())
        B[i].append(num)
print("\nMatrix B is:")
for i in range(r2):
    for j in range(c2):
        print(B[i][j], end=" ")
    print()
if ((r1!=r2) or (c1!=c2)):
    print("the matrices are not compatable for addition")
```

```
for i in range(r2):
    C.append([])
    for j in range(c2):
        C[i].append(A[i][j]+B[i][j])
print("matrix addition A+B =")
for i in range(r2):
    for j in range(c2):
        print(C[i][j], end=" ")
    print()
```

Transpose of a Matrix

```
A = []
T = []

print("enter no of rows and no of columns of matrix A")
r1=int(input())
c1=int(input())
print("Enter rxc Elements for Matrix A: ")
for i in range(r1):
    A.append([])
    for j in range(c1):
        num = int(input())
        A[i].append(num)
print("\nMatrix A is:")
for i in range(r1):
    for j in range(c1):
        print(A[i][j], end=" ")
    print()

for i in range(c1):
    T.append([])
    for j in range(r1):
        T[i].append(A[j][i])

print("transpose matrix  =")
for i in range(c1):
    for j in range(r1):
        print(T[i][j], end=" ")
    print()
```

Matrix addition

```
# Program to add two matrices
# 3x3 matrix
X = [[12,7,3],[4 ,5,6],[7 ,8,9]]
# 3x3 matrix
Y = [[5,8,1],[6,7,3],[4,5,9]]
# result is 3x3
result = [[0,0,0],[0,0,0],[0,0,0]]
print("Matrix X ")
for i in range(3):
   for j in range(3):
      print(X[i][j], end=" ")
   print()
print("Matrix Y ")
for i in range(3):
   for j in range(3):
      print(Y[i][j], end=" ")
   print()
# iterate through rows of X
for i in range(3):
  # iterate through columns of Y
  for j in range(3):
      result[i][j] = X[i][j] + Y[i][j]

print("addition Matrix  ")
for i in range(3):
   for j in range(3):
      print(result[i][j], end=" ")
   print()
```

Matrix multiplication

```
# Program to multiply two matrices

# 3x3 matrix
X = [[12,7,3],[4 ,5,6],[7 ,8,9]]
# 3x4 matrix
Y = [[5,8,1,2],[6,7,3,0],[4,5,9,1]]
# result is 3x4
result = [[0,0,0,0],[0,0,0,0],[0,0,0,0]]
print("Matrix X ")
for i in range(3):
    for j in range(3):
        print(X[i][j], end=" ")
    print()
print("Matrix Y ")
for i in range(3):
    for j in range(4):
        print(Y[i][j], end=" ")
    print()

# iterate through rows of X
for i in range(len(X)):
   # iterate through columns of Y
   for j in range(len(Y[0])):
       # iterate through rows of Y
       for k in range(len(Y)):
           result[i][j] += X[i][k] * Y[k][j]

print("Product Matrix  ")
for i in range(3):
    for j in range(4):
        print(result[i][j], end=" ")
    print()
```

Write a program to perform bubble sort on a list without using the sort(). (Record Program)

Bubble Sorting

First Pass

swapping
5 1 4 2 8
swapping
1 5 4 2 8
swapping
1 4 5 2 8
no swap
1 4 2 5 8
1 4 2 5 8

Second Pass

no swap
1 4 2 5 8
swapping
1 4 2 5 8
no swap
1 2 4 5 8
no swap
1 2 4 5 8
1 2 4 5 8

Third Pass

no swap
1 2 4 5 8
no swap
1 2 4 5 8
no swap
1 2 4 5 8
no swap
1 2 4 5 8
1 2 4 5 8

© w3resource.com

```
def bubbleSort(list):
# loop to access each array element
  for i in range(len(list)):
# loop to compare array elements
      for j in range(len(list) - i - 1):
# compare two adjacent elements
          if list[j] > list[j + 1]:
                  # swapping elements
              temp = list[j]
              list[j] = list[j+1]
              list[j+1] = temp
data = [2, 45, 0, 11, 9]
bubbleSort(data)
print('Sorted Array in Ascending Order:')
print(data)
```

Write a program to display the elements of a list in reverse order without using the reverse()

(Record Program)

```
list = [1, 2, 3, 4, 5]
print("the list in original order is")
for i in range(len(list)):
      print(list[i])
print("the list in reverse order is")
for i in range(len(list)-1, -1, -1):
     print(list[i])
```

program to create a list and eliminate the duplicate values from the list

```
list = [1, 3, 5, 6, 3, 5, 6, 1]
print ("The original list is : " ,list)

res = []
for i in list:
    if i not in res:
        res.append(i)
print ("The list after removing duplicates : ", res)
```

Unit - IV

- Lists: List, Creating List, Updating the Elements of a List, Sorting the List Elements. Storing Different Types of Data in a List, Nested Lists, Nested Lists as Matrices.
- Tuples: Creating Tuple, Accessing the Tuple Elements, Basic Operations on Tuples, Functions to Process Tuples, Inserting Elements in a Tuple, Modifying Elements of a Tuple, Deleting Elements from a Tuple.
- Sets: Creating Set, Basic Operations on Sets, Methods of Set.
- Dictionaries: Operations on Dictionaries, Dictionary Methods, Using for Loop with Dictionaries, Sorting the Elements of a Dictionary using Lambdas, Converting Lists into Dictionary.

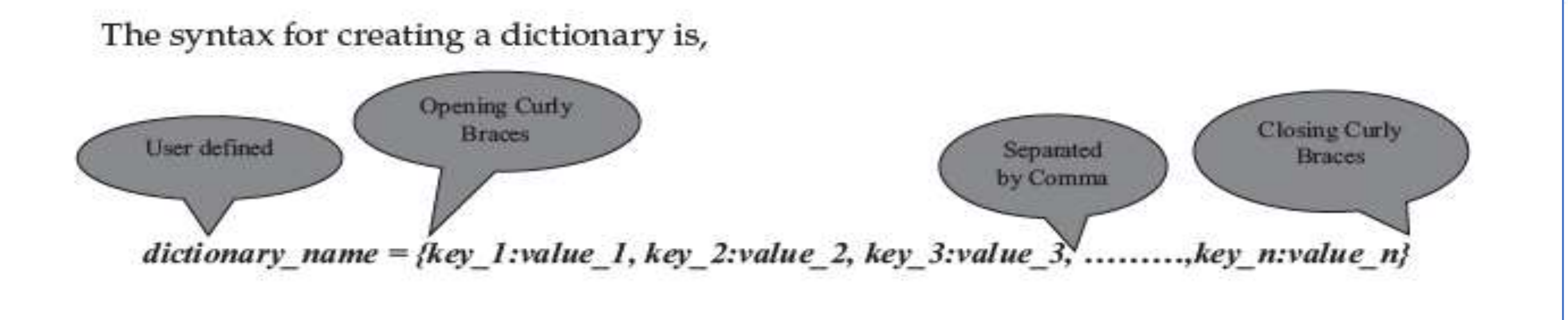
Dictionary

- A dictionary is a collection of an ordered, mutable set of *key: value pairs, with the requirement* that the keys are unique within a dictionary.
- As of Python version 3.7, dictionaries are *ordered*. In Python 3.6 and earlier, dictionaries are *unordered*.
- No duplicate values are allowed.
- In the real world, the Contacts list in our phone.
- It is practically impossible to memorize the mobile number of everyone.
- In the Contacts list, we store the name of the person as well as his number.
- This allows to identify the mobile number based on a person's name.
- We can think of a person's name as the key that retrieves his mobile number, which is the value associated with the key.

Creating Dictionary

- Dictionaries are constructed using curly braces { }, wherein you include a list of *key:value pairs separated by commas.*
- *Also, there* is a colon (:) separating each of these key and value pairs, where the words to the left of the colon operator are the keys and the words to the right of the colon operator are the values.
- Unlike lists, which are indexed by a range of numbers, dictionaries are indexed by keys.
- Here a key along with its associated value is called a *key:value pair.*
- *Dictionary* keys are case sensitive.

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING



Example:

```
dict1 = {
    "R": "Red",
    "Y": "Yellow",
    "G": "Green"
    }
print(dict1)
```

Output

```
{'R': 'Red', 'Y': 'Yellow', 'G': 'Green'}
```

- Each of the keys and values here is of string type.
- A value in the dictionary can be of any data type including string, number, list, or dictionary itself.

Accessing Items of Dictionary

- We can access the items of a dictionary by referring to its key name, inside square brackets:

Example:

```
dict1 = {
 "R": "Red",
"Y": "Yellow",
"G": "Green"
}

x=dict1["R"]
print(x)
```

Output

```
Red
```

There is also a method called get() that will give you the same result:

```
dict1 = {
          "R": "Red",
          "Y": "Yellow",
          "G": "Green"
          }
x = dict1.get("R")
print(x)
```

Get Keys

- The keys() method will return a list of all the keys in the dictionary.

```
x = dict1.keys()
print(x)
```

Add a new item to the dictionary

```
dict1 = {
                "R": "Red",
               "Y": "Yellow",
               "G": "Green"
       }
print(dict1)
dict1["P"]="Pink"
print(dict1)
```

output

Get Values

- The values() method will return a list of all the values in the dictionary.

```
dict1 = {
                    "R": "Red",
                    "Y": "Yellow",
                    "G": "Green"
          }
x=dict1.values()
print(x)
```

<u>Output</u>

Get Items

- The items() method will return each item in a dictionary, as tuples in a list.

```
dict1 = {
            "R": "Red",
            "Y": "Yellow",
            "G": "Green"
      }
print(dict1)
x=dict1.items()
print(x)
```

Check if Key Exists

- To determine if a specified key is present in a dictionary use the in keyword:

```
dict1 = {
                "R": "Red",
                "Y": "Yellow",
                "G": "Green"
        }
if "R" in dict1:
   print("This is the key of the dictionary")
```

output

This is the key of the dictionary

Python - Change Dictionary Items

Change Values

- We can change the value of a specific item by referring to its key name:

```
dict1 = {
                "R": "Red",
                "Y": "Yellow",
                "G": "Green"
        }
print(dict1)
dict1["R"]="Rocking"
print(dict1)
```

Output

Update Dictionary

- The update() method will update the dictionary with the items from the given argument.
- The argument must be a dictionary, or an iterable object with key:value pairs.

```
dict1 = {
            "R": "Red",
            "Y": "Yellow",
            "G": "Green"
      }
print(dict1)
dict1.update({"G":"Golden"})
print(dict1)
```

Python - Remove Dictionary Items

There are several methods to remove items from a dictionary:

- The **pop()** method removes the item with the specified key name:

```
dict1 = {
                "R": "Red",
                "Y": "Yellow",
                "G": "Green"
        }
print(dict1)
dict1.pop("R")
print(dict1)
```

<u>Output</u>

- The **popitem()** method removes a random item(before 3.7 version) and last (after 3.7) .

```
dict1 = {
                "R": "Red",
                "Y": "Yellow",
                "G": "Green"
        }
print(dict1)
dict1.popitem()
print(dict1)
```

- The **del** keyword removes the item with the specified key name same like **pop. (only syntax is different)**

```
dict1 = {
                "R": "Red",
                "Y": "Yellow",
                "G": "Green"
        }
print(dict1)
del dict1["G"]
print(dict1)
```

- The **del** keyword can also delete the dictionary completely:

```
del dict1
```

- The **clear()** method empties the dictionary:

```
dict1 = {
                    "R": "Red",
                    "Y": "Yellow",
                    "G": "Green"
          }
print(dict1)
dict1.clear()
print(dict1)
```

<u>Output</u>

Dictionary Methods

Python has a set of built-in methods that you can use on dictionaries.

Method	Description
clear()	Removes all the elements from the dictionary
copy()	Returns a copy of the dictionary
fromkeys()	Returns a dictionary with the specified keys and value
get()	Returns the value of the specified key
items()	Returns a list containing a tuple for each key value pair
keys()	Returns a list containing the dictionary's keys
pop()	Removes the element with the specified key
popitem()	Removes the last inserted key-value pair
setdefault()	Returns the value of the specified key. If the key does not exist: insert the key, with the specified value
update()	Updates the dictionary with the specified key-value pairs
values()	Returns a list of all the values in the dictionary

The **copy()** method returns a copy of the specified dictionary.

```
dict1 = {
                    "R": "Red",
                    "Y": "Yellow",
                    "G": "Green"
          }
x=dict1.copy()
print(x)
```

Output

- The **fromkeys()** method returns a dictionary with the specified keys and the specified value.

```
x = ('key1', 'key2', 'key3')
y = 0
thisdict = dict.fromkeys(x, y)
print(thisdict)
```

<u>output</u>

['key1': 0, 'key2': 0, 'key3': 0]

- The **setdefault()** method returns the value of the item with the specified key.
- If the key does not exist, insert the key, with the specified value.

```
dict1 = {
            "R": "Red",
            "Y": "Yellow",
            "G": "Green"
      }
x=dict1.setdefault("G", "Golden")
print(x)
```

<u>output</u>

```
      Green
```

```
dict1 = {
                    "R": "Red",
                    "Y": "Yellow"
          }
x=dict1.setdefault("G","Golden")
print(x)
```

<u>Output</u>

```
          Golden
```

- Since this key is not there in the dictionary, it is inserted and the value is displayed

Using for Loop with Dictionaries

- We can loop through a dictionary by using a for loop.
- When looping through a dictionary, the return value are the *keys* of the dictionary
- There are methods that can be used to return the *values* also.

```
dict1 = {
            "R": "Red",
            "Y": "Yellow",
            "G": "Green"
      }
for x in dict1:
  print(x)
```

Output

- We can also Print all *values* in the dictionary, one by one:

```
dict1 = {
                    "R": "Red",
                    "Y": "Yellow",
                    "G": "Green"
          }
for x in dict1:
   print(dict1[x])
```

<u>Output</u>

- We can also use the **values()** method to return values of a dictionary:

```
dict1 = {
                              "R": "Red",
                              "Y": "Yellow",
                              "G": "Green"
                    }
for x in dict1.values():
   print(x)
```

- We can use the **keys()** method to return the keys of a dictionary:

```
dict1 = {
                              "R": "Red",
                              "Y": "Yellow",
                              "G": "Green"
                    }
for x in dict1.keys():
   print(x)
```

- We can Loop through both *keys* and *values*, by using the **items()** method:

```
dict1 = {
                    "R": "Red",
                    "Y": "Yellow",
                    "G": "Green"
          }
for x,y in dict1.items():
   print(x,y)
```

Output

Converting Lists into Dictionary

- We can convert a Python list to a dictionary using three methods

1. dictionary comprehension
2. dict.fromkeys()
3. zip() method.

- All methods create a new dictionary. They do not modify the existing list.

Method - 1 Using Dictionary Comprehension

```
student = ["Raju", "Ravi", "Rama", "Roopa"]
student_dict = { course : "CSM" for course in student }
print(student_dict)
```

Output

- In the above example, we have created a student list with 4 names that is to be converted into the dictionary.
- Using the dictionary comprehension, the list is converted in to a dictionary in a single line.
- The list elements are turned into keys and "CSM" becomes the value.

```
list1 = [1, 2, 3, 4, 5]
square_dict = {n: n*n for n in list1}
print(square_dict)
```

Output

```
{1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Method - 2 Using dict.fromkeys()

```
student = ["Raju", "Ravi", "Rama", "Roopa"]
student_dict = dict.fromkeys(student,"CSM")
print(student_dict)
```

Output

- We can create dictionary from a list using the *dict.fromkeys()* method.
- This method accepts a list of keys that we want to convert into a dictionary.
- The value specified is assigned to every key.

Note: In above two methods we can assign same value to all the keys. If we want to assign different values to different keys we must use method 3: zip() method

Method - 3 Using zip()

```
student = ["Raju", "Ravi", "Rama", "Roopa"]
course=["CSM","CSD","CSM","CSE"]
student_dict = dict(zip(student, course))
print(student_dict)
```

Output

- First, we specify two lists: a list of students, and a list of their courses.
- Then, we use the *zip()* function to merge the two lists together.
- The *zip()* function returns a list of merged tuples.
- Because we want a dictionary, we use *dict()* to convert the tuples into a dictionary.

Python Lambda

- A lambda function is a small anonymous function in python that can take any number of arguments, but can only have one expression.

Syntax

lambda *arguments* : *expression*

- The expression is executed and the result is returned:

Example 1:Add 10 to argument a, and return the result:

```
x = lambda a : a + 10
print(x(5))
```

Example 2: Multiply argument a with argument b and return the result:

```
x = lambda a, b : a * b
print(x(5, 6))
```

Example 3: find sum of argument a, b, and c and return the result:

```
x = lambda a, b, c : a + b + c
print(x(5, 6, 2))
```

Sorting the Elements of a Dictionary using Lambdas

```
dict1 = [{ "name" : "Amar", "age" : 19},
{ "name" : "Rama", "age" : 20 },
{ "name" : "Prithvi" , "age" : 20 }]
print("the dict1 printed before soring")
print(dict1)
print("\r")
```

```
print("The dict1 printed sorting by age: ")
print(sorted(dict1, key = lambda i: i['age']))
print("\r")
print("The dict1 printed sorting by age and name: ")
print(sorted(dict1, key = lambda i: (i['age'], i['name'])))
print ("\r")
print("The dict1 printed sorting by age in descending order: ")
print(sorted(dict1, key = lambda i: i['age'],reverse=True))
```

Unit - V

Files:

- Working with Files and Directories
- File Processing
- Controlling File I/O.

FILES

- Files are named locations on disk to store related information.
- They are used to permanently store data in a non-volatile memory (e.g. hard disk).
- Since Random Access Memory (RAM) is volatile (which loses its data when the computer is turned off), we use files for future use of the data by permanently storing them.
- Python has several functions for creating, reading, updating, and deleting files.
- Hence, in Python, a file operation takes place in the following order:

➢ Open a file

➢ Read or write (perform operation)

➢ Close the file

Opening Files in Python

- Python has a built-in open() function to open a file.
- The open() function takes two parameters; *filename*, and *mode*.
- There are four different methods (modes) for opening a file:

➢ "r" - Read - Default value. Opens a file for reading, error if the file does not exist

➢ "a" - Append - Opens a file for appending, creates the file if it does not exist

➢ "w" - Write - Opens a file for writing, creates the file if it does not exist

➢ "x" - Create - Creates the specified file, returns an error if the file exists

- In addition we can specify if the file should be handled as binary or text mode

➢ "t" - Text - Default value. Text mode

➢ "b" - Binary - Binary mode

- To open a file for reading it is enough to specify the name of the file:

```
f = open("demofile.txt")
```

- The code above is the same as:

```
f = open("demofile.txt", "rt")
```

- Because "r" for read, and "t" for text are the default values, you do not need to specify them.
- Make sure the file exists, or else you will get an error.
- Assume we have the file demofile.txt, located in the same folder as Python:

demofile.txt

```
Hello! Students of CSM & CSD.
Good Luck!
```

- To open the file, use the built-in open() function.
- The open() function returns a file object, which has a read() method for reading the content of the file:

```
f = open("demofile.txt", "r")
print(f.read())
```

- If the file is located in a different location, you will have to specify the file path, like this:

```
f = open("D:\\myfiles\demofile.txt", "r")
print(f.read())
```

- By default the read() method returns the whole text, but you can also specify how many characters you want to return:

```
f = open("demofile.txt", "r")
print(f.read(5))
```

- You can return one line by using the readline() method:

```
f = open("demofile.txt", "r")
print(f.readline())
```

- By calling readline() two times, you can read the two first lines:

```
f = open("demofile.txt", "r")
print(f.readline())
print(f.readline())
```

- By looping through the lines of the file, you can read the whole file, line by line:

```
f = open("demofile.txt", "r")
for x in f:
  print(x)
```

- It is a good practice to always close the file when you are done with it.

```
f = open("demofile.txt", "r")
print(f.readline())
f.close()
```

```
print(f.readlines())
```

Write to an Existing File

- To write to an existing file, you must add a parameter to the open() function:
- "a" - Append - will append to the end of the file
- "w" - Write - will overwrite any existing content

Example

Open the file "demofile2.txt" and append content to the file:

```
f = open("demofile2.txt", "a")
f.write("Now the file has more content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

- Open the file "demofile2.txt" and overwrite the content:

```
f = open("demofile2.txt", "w")
f.write("Woops! I have deleted the content!")
f.close()

#open and read the file after the appending:
f = open("demofile2.txt", "r")
print(f.read())
```

- the "w" method will overwrite the entire file.

Create a New File

- To create a new file in Python, use the open() method, with one of the following parameters:
- "x" - Create - will create a file, returns an error if the file exist
- "a" - Append - will create a file if the specified file does not exist
- "w" - Write - will create a file if the specified file does not exist

Example

- Create a file called "myfile.txt":

```
f = open("myfile.txt", "x")
```

- Result: a new empty file is created!

Delete a File

- To delete a file, you must import the OS module, and run its os.remove() function:

Example

- Remove the file "demofile.txt":

```
import os
os.remove("demofile.txt")
```

Delete Folder

- To delete an entire folder, use the os.rmdir() method:

Example

- Remove the folder "myfolder":

```
import os
os.rmdir("myfolder")
```

Note: You can only remove *empty* folders.

Changing position in files

- Seek() method: seek() function is used to **change the position of the File Handle** to a given specific position. File handle is like a cursor, which defines from where the data has to be read or written in the file.

Syntax: *f.seek(offset, from_what), where f is file pointer*

Parameters:

Offset: *Number of positions to move forward*
from_what: *It defines point of reference.*

Returns: *Does not return any value*

- The reference point is selected by the **from_what** argument. It accepts three values:

0: sets the reference point at the beginning of the file
1: sets the reference point at the current file position
2: sets the reference point at the end of the file
By default from_what argument is set to 0.

Example:

```
f=open(“demofile.txt","r")
f.seek(10)
print(f.tell())
print(f.readline())
f.close()
```

The tell() method returns the current file position in a file stream.

Directories in Python

- All files are contained within various directories, and Python has the os module with several methods that help us to create, remove, and change directories.

➢ **The mkdir() Method**

- We can use the *mkdir()* method of the os module to create directories in the current directory. we need to supply an argument to this method which contains the name of the directory to be created.

Syntax

```
os.mkdir("newdir")
```

Example

```
import os
os.mkdir("test")        # Create a directory "test"
```

➢ The getcwd() Method

- The *getcwd()* method displays the current working directory.

Syntax

```
os.getcwd()
```

Example

```
import os
print(os.getcwd())
```

➢ Print List Directories and Files

```
import os
print(os.listdir())
```

➢Renaming a Directory or a File

- The rename() method can rename a directory or a file.
- For renaming any directory or file, the rename() method takes in two basic arguments:
- the old name as the first argument and the new name as the second argument.

First execute

```
import os
os.mkdir(“newdir”)
print(os.listdir())
```

Next Execute

```
import os
os.rename(“newdir”, “renamedir”)
print(os.listdir())
```

➢The rmdir() Method

- The *rmdir()* method deletes the directory, which is passed as an argument in the method.
- Before removing a directory, all the contents in it should be removed.

Syntax

```
os.rmdir('dirname')
```

Example

```
import os
os.rmdir(“renamedir”)
print(os.listdir())
```

Controlling File I/O

- Once we open any file we may require to control the way how the file is accessed and shared.
- To use the file control we need to first find the file descriptor number from the file object.
- We can use the fileno( ) method to do this.

```
f=open("demofile.txt","wb")
print("name of file:=",f.name)

fid=f.fileno()
print("file descriptor:=",fid)

f.close()
```

```
C:\ayesha>notepad fileprog.txt

C:\ayesha>python fileprog.txt
name of file:= demofile.txt
file descriptor:= 3
```

- **fcntl( ) method:** after finding the file descriptor, next the fcntl module sets or gets the configuration information for the file. The basic format for this method is

fcntl(fd , command , [, args]) fd : file descriptor

command: constant argument specifying what to send to the file control

args: optional

Command	Description
1. F_DUPFD	This command Duplicates the file descriptor
2. F_SETFD	This command sets the **close-on-exce** flag. If this id set as TRUE then file descriptor is closed and if set as False then file descriptor is kept open.
3. F_GETFD	Returns the current value of the **close-on-exce** flag.
4. F_SETFL	sets the **file status** flag.
5. F_GETFL	gets the current file status flag value

- **<u>ioctl( ) method:</u>** this method is similar to fcntl( ) except that it provides an interface to the ioctl subsystem for controlling I/O.
- **<u>File Locking:</u>** when working with files we perform reading and writing operations. When write operation is performed it is necessary to take care that no other process is writing to the same file. That is two processes must not write to one file at same time.

In this case the best way to handle is file locking. There are two functions for locking a file

1. flock( ) : this is a whole file locking method.
2. lockf ( ) : allow us to lock some specific portions of files.

flock( ) function is defined as

flock(fd, op) fd: file descriptor

op: lock operation

Operation	**description**
LOCK_EX	Exclusive lock. Other processes cannot even obtain a shared lock on the file.
LOCK_NB	Don’t block the locking process.
LOCK_SH	Get a shared lock. This allows only your process to read and write to the file. other processes can only read.
LOCK_UN	Remove the lock or unlock.

lockf( ) function is defined as:

lockf(fd, operation[, length[, start[, whence]]])

fd and op are the same as before. Len, start and whence arguments define the length and duration of the lock.

VAAGDEVI COLLEGES

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

MODULES

VAAGDEVI COLLEGES

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

VAAGDEVI COLLEGES

DEPARTMENT OF
COMPUTER SCIENCE & ENGINEERING

UNIT 5- MODULES

- A python module can be defined as a python program file which contains a python code including python functions, class, or variables. In other words, we can say that our python code file saved with the extension (.py) is treated as the module. We may have a runnable code inside the python module.
- Modules in Python provides us the flexibility to organize the code in a logical way.
- To use the functionality of one module into another, we must have to import the specific module.

Create a Module:

To create a module just save the code you want in a file with the file extension .py:

Example

Save this code in a file named file.py

```
def  display(name):
     print("Hello, " + name)
```

Use a Module:

Now we can use the module we just created, by using the import statement:

Example

Import the module named file, and call the display function:

```
import  file
   file.display("sai")       #modulename.functionname()
```

Loading the module in our python code:

We need to load the module in our python code to use its functionality. Python provides two types of statements as defined below.

1. The import statement
2. The from-import statement

1. The import statement:

The import statement is used to import all the functionality of one module into another. Here, we must notice that we can use the functionality of any python source file by importing that file as the module into another python source file.

We can import multiple modules with a single import statement, but a module is loaded once regardless of the number of times, it has been imported into our file.

The Syntax to use the import statement is:

import module1,module2,........ module n

Hence, if we need to call the function display() defined in the file file.py, we have to import that file as a module into our module as shown in the example below.

Example: mainfile.py

```
import  file
name = input("Enter the name:")
file.display(name)
```

file.py

```
def  display(name)
       print("Hi "+name)
```

Output:

Enter the name: Sai

Hi sai

Note: first run file.py then run mainfile.py

Execution:

Step1: Open a command prompt and create a new file that is file.py

```
Command Prompt
Microsoft Windows [Version 10.0.18363.1556]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\admin>cd\

C:\>cd RAMA

C:\RAMA>notepad file.py
```

Step2: Write some code in file.py.
#display prints a message to the name being passed.

```
file - Notepad
File   Edit   Format   View   Help
def display(name):
        print("HELLO "+name)
```

Step3: Next run that program.

```
Command Prompt
Microsoft Windows [Version 10.0.18363.1556]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\admin>cd\

C:\>cd RAMA

C:\RAMA>notepad file.py

C:\RAMA>python file.py

C:\RAMA>
```

Step4: Create a new file that is mainfile.py.

```
Command Prompt
Microsoft Windows [Version 10.0.18363.1556]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\admin>cd\

C:\>cd RAMA

C:\RAMA>notepad file.py

C:\RAMA>python file.py

C:\RAMA>notepad mainfile.py

C:\RAMA>
```

Step5: Write some code and import file.py in mainfile.py.

mainfile - Notepad

File Edit Format View Help

```
import file
file.display("SUMANYA")
```

Step6: run the mainfile.py program it give result that is HELLO SUMANYA.

Command Prompt

```
Microsoft Windows [Version 10.0.18363.1556]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\admin>cd\

C:\>cd RAMA

C:\RAMA>notepad file.py

C:\RAMA>python file.py

C:\RAMA>notepad mainfile.py

C:\RAMA>python mainfile.py
HELLO SUMANYA

C:\RAMA>_
```

The from-import statement:

Instead of importing the whole module into the namespace, python provides the flexibility to import only the specific attributes of a module. This can be done by using from? import statement. The syntax to use the from-import statement is given below.

from < module-name> import <name 1>, <name 2>..,<name n>

Consider the following module named as calculation which contains three functions as summation, multiplication, and divide.

calculation.py:

```
#place the code in the calculation.py
def summation(a,b):
    return a+b
def multiplication(a,b):
    return a*b;
def divide(a,b):
    return a/b;
```

Maincalc.py:

```
from calculation import summation
                    #it will import only the summation() from calculation.py
a = int(input("Enter the first number"))
b = int(input("Enter the second number"))
print("Sum = ",summation(a, b))
   #we do not need to specify the module name while accessing summatio
```

Output:

```
Enter the first number10
Enter the second number20
Sum =  30
```

The from...import statement is always better to use if we know the attributes to be imported from the module in advance. It doesn't let our code to be heavier. We can also import all the attributes from a module by using *. Consider the following syntax.

from <module> import *

Variables in Module

The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):

Example:

Save this code in the file mymodule.py

```
person1 = {
        "name": "John",
        "age": 36,
        "country": "Norway"
      }
```

Import the module named mymodule, and access the person1 dictionary:

```
import mymodule
  a = mymodule.person1["age"]
  print(a)
```

Renaming a module:

Python provides us the flexibility to import some module with a specific name so that we can use this name to use that module in our python source file.

The syntax to rename a module is given below.

import <module-name> as <specific-name>

Example:

import calculation **as** cal;

Example:

```
#the module calculation of previous example is imported in this example as cal.
import calculation as cal
a = int(input("Enter a?"))
b = int(input("Enter b?"))
print("Sum = ",cal.summation(a,b))
```

Output:

```
Enter a?10
Enter b?20
Sum =  30
```

Built-in Modules:

There are several built-in modules in Python, which you can import whenever you like.

Example:

Import and use the platform module:

```
import platform
  x = platform.system()
  print(x)
```

Output: Windows

Using dir() function:

The dir() function returns a sorted list of names defined in the passed module. This list contains all the sub-modules, variables and functions defined in this module.

Example:

There is a built-in function to list all the function names (or variable names) in a module. List all the defined names belonging to the platform module:

```
import platform
  x = dir(platform)
  print(x)
```

The reload() function:

As we have already stated that, a module is loaded once regardless of the number of times it is imported into the python source file. However, if you want to reload the already imported module to re-execute the top-level code, python provides us the reload() function.

The syntax is:

reload(<module-name>)

for example, to reload the module calculation defined in the previous example, we must use the following line of code.

reload(calculation)

Tricks for Importing Modules

Python also supports a number of different techniques and tricks that we can use to import modules within a python script.

1. Using import in a script
2. Trapping import statements
3. Identifying a module or a script

1.Using import in a script:

```
if(module=='test1'):
    import  test1
else:
    import   test2
```

2. Trapping import statements:

The import statement raises an **importError** exception if a module fails to load correctly. We can trap a failed module load directly within the script, allowing you to exit from the program.

Example:

```
try:
   import mymodule
except ImportError:
  print “The required module is missing here”
  import sys
  sys.exit()
```

3. Identifying a module or a script:

Each module defines a variable, _ _name_ _, that contains the name of the module. We can use this variable to determine which module a particular piece of code executing within. It is also used to determine whether a given module is running as a script or whether it has been imported. Modules running as scripts set _ _name_ _ to _ _main_ _

```
if _ _name_ _ == '_ _main_ _':
 # work as a script
else
   # work as a module
```

Python packages

The packages in python facilitate the developer with the application development environment by providing a hierarchical directory structure where a package contains sub-packages, modules, and sub-modules. The packages are used to categorize the application level code efficiently.

Let's create a package named Employees in your home directory. Consider the following steps.

1. Create a directory with name pack1.
2. Create a python source file with name module1.py on the path \pack1\module1.py.

C:\RAMA\pack1>notepad module1.py #Write a python code in module1.py

```
def show():
   print("HELLO, PYTHON")
```

3. Similarly, we can create any number of modules.

   ex: C:\RAMA\pack1>notepad module2.py

4. Now, the directory pack1 which we have created in the first step contains two python modules.

5. Now, the directory pack1 has become the package containing two python modules.

6. To use the modules defined inside the package pack1, we must have to import this in our python source file. Let's create a simple python source file at our home directory (/RAMA) which uses the modules defined in this package.

C:\>cd RAMA

C:\RAMA>mkdir pack1

C:\RAMA>cd pack1

C:\RAMA\pack1>notepad module1.py #Write a python code in module1.py

```
def  show():
   print("HELLO, PYTHON")
```

C:\RAMA\pack1>python module1.py # run module1.py

C:\RAMA\pack1>cd .. # come out from pack1

C:\RAMA>notepad test.py # create test.py

```
import pack1.module1
pack1.module1.show()
```

C:\RAMA>python test.py # run test.py

HELLO, PYTHON **# output**

DEPARTMENT OF

COMPUTER SCIENCE & ENGINEERING

PYTHON PROGRAMMING

B.Tech I Year II Sem

Dr Ayesha Banu

Assistant Professor

Department of CSE

Unit – I

Introduction to Python: What is Python?, What is Python Good For?, Python History, How does Python Execute a Program, Review of a Simple Program, Some of the Basic Commands, Variables, Statements, Input/Output Operations, Keywords, Variables, Assigning values, Standard Data Types, Strings, Operands and operators.

Unit – II

Understanding the Decision Control Structures: The if Statement, A Word on Indentation, The if … else Statement, The if … elif … else Statement,

Loop Control Statements: The while Loop, The for Loop, Infinite Loops, Nested Loops.

The break Statement, The continue Statement, The pass Statement, The assert Statement, The return Statement.

Unit – III

Functions- Function Definition and Execution, Scoping, Arguments: Arguments are Objects, Argument Calling by Keywords, Default Arguments, Function Rules, Return Values.

Advanced Function Calling: The apply Statement, The map Statement, Indirect Function Calls.

Unit - IV

Lists: List, Creating List, Updating the Elements of a List, Sorting the List Elements. Storing Different Types of Data in a List, Nested Lists, Nested Lists as Matrices.

Tuples: Creating Tuple, Accessing the Tuple Elements, Basic Operations on Tuples, Functions to Process Tuples, Inserting Elements in a Tuple, Modifying Elements of a Tuple, Deleting Elements from a Tuple.

Sets: Creating Set, Basic Operations on Sets, Methods of Set.

Dictionaries: Operations on Dictionaries, Dictionary Methods, Using for Loop with Dictionaries, Sorting the Elements of a Dictionary using Lambdas, Converting Lists into Dictionary.

Unit – V

Modules: Importing a Module, Tricks for Importing Modules, Packages.

Exceptions and Error Trapping: What is an Exception?, Exception Handling: try..except..else.., try..finally.., Exceptions Nest, Raising Exceptions, Built-In Exceptions.

UNIT – 5

Exceptions

There are (at least) two distinguishable kinds of errors:

syntax errors and *exceptions*.

- Syntax errors, also known as parsing errors, are the most common kind of errors . As the name suggests this error is caused by the wrong syntax in the code. It leads to the termination of the program.
- Even if a statement or expression is syntactically correct, it may cause an error when an attempt is made to execute it. These Errors detected during execution are called *exceptions* .
- Exceptions are raised when the program is syntactically correct, but the code resulted in an error. This error does not stop the execution of the program, however, it changes the normal flow of the program.

```
1 x=10
2 if(x>0)
3     print("x is positive")
```

```
File "main.py", line 2
    if(x>0)
          ^
SyntaxError: invalid syntax
```

syntax error

```
1 x=10
2 y=x/0
3 print(x)
4 print(y)
```

```
Traceback (most recent call last):
  File "main.py", line 2, in <module>
    y=x/0
ZeroDivisionError: division by zero
```

exception

```
>>> 10 * (1/0)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> 4 + spam*3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> '2' + 2
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: Can't convert 'int' object to str implicitly
```

- The last line of the error message indicates what happened.
- Exceptions come in different types, and the type is printed as part of the message:
- the types in the example are ZeroDivisionError, NameError and TypeError.
- The string printed as the exception type is the name of the built-in exception that occurred.
- This is true for all built-in exceptions, but need not be true for user-defined exceptions

Built-in Exceptions

- *exception* Exception: All built-in, non-system-exiting exceptions are derived from this class. All user-defined exceptions should also be derived from this class.
- *exception* ArithmeticError: This is base class for those built-in exceptions that are raised for various arithmetic errors: OverflowError, ZeroDivisionError.

- OverflowError :Raised when the result of an arithmetic operation is too large to be represented.
- ZeroDivisionError :Raised when the second argument of a division or modulo operation is zero.
- **IndentationError:** Raised when there is incorrect indentation.

- *exception* BufferError :Raised when a buffer related operation cannot be performed.
- *exception* LookupError :The base class for the exceptions that are raised when a key or index used on a mapping or sequence is invalid: IndexError, KeyError.

EXCEPTION HANDLING

- When an error or exception occurs, Python will normally stop and generate an error message.
- It is possible to write programs that handle selected exceptions using try .. except .. else clauses

- The try block lets us test a block of code for errors.
- The except block lets us handle the error.

Example: 1

```
1  print(x)
2
3
4
```

```
Traceback (most recent call last):
  File "main.py", line 1, in <module>
    print(x)
NameError: name 'x' is not defined
```

- in above example we print the value of x. But x is not defined and hence it raises an error when the program is executed.

- This can be handled using the try .. except clause
- The statement which causes error is written after the try clause
- When the try block raises an error, the except block will be executed.
- Therefore we can include the exception message to be displayed after the except clause.

```
1 try:
2     print(x)
3 except:
4     print("An exception occurred")
5
6
```

inp

```
An exception occurred
```

- Example 2:

```
x=int(input("enter x value"))
y=int(input("enter y value"))
z=x/y
print(z)
```

Output:

```
enter x value10
enter y value2
5.0

enter x value5
enter y value0
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    z=x/y
ZeroDivisionError: division by zero
```

```
x=int(input("enter x value"))
y=int(input("enter y value"))
try:
        z=x/y
        print(z)
except:
        print("this is an exception chk denominator value")
```

Output

```
enter x value10
enter y value2
5.0


enter x value3
enter y value0
this is an exception chk denominator value
```

The try statement works as follows.

- First, the *try clause* is executed.
- If no exception occurs, the *except clause* is skipped and execution of the try statement is finished.
- If an exception occurs during execution of the try clause, the rest of the clause is skipped.

- A single try statement can have multiple except statements.
- This is useful when the try block contains statements that may throw different types of exceptions.
- At most one except clause statement only will be executed.

Python try with else clause

```
x=int(input("enter x value"))
y=int(input("enter y value"))
try:
	z=x/y
	print(z)
except:
	print(" chk the remainder value")
else:
	print("the division is success")
```

<u>Output</u>

```
enter x value12
enter y value4
3.0
the division is sucess


enter x value2
enter y value0
 chk the remainder value
```

Multiple except clauses for one try clause

- We can define as many exception blocks as we want for one single try clause, when we want to execute a special block of code for a special kind of error:

Example:

```
# x=int(input("enter x value"))
y=int(input("enter y value"))
try:
    z=x/y
    print(z)
except NameError:
    print("chk if all variables are defined")
except ZeroDivisionError:
    print(" chk the remainder value")
else:
    print("the division is success")
```

```
try:
  a = int(input("Please enter the numerator: "))
  b = int(input("Please enter the denominator: "))
  print(a / b)
except ZeroDivisionError:
  print("Please enter a valid denominator.")
except ValueError:
  print("Both values have to be integers.")
```

Output1:

```
Please enter the numerator: 10
Please enter the denominator: 2
5.0
```

When both the inputs are correct

Output2:

```
Please enter the numerator: 10
Please enter the denominator: 0
Please enter a valid denominator.
```

When denominator value is zero

Output3:

```
Please enter the numerator: a
Both values have to be integers.
```

When any one of the inputs is not an integer

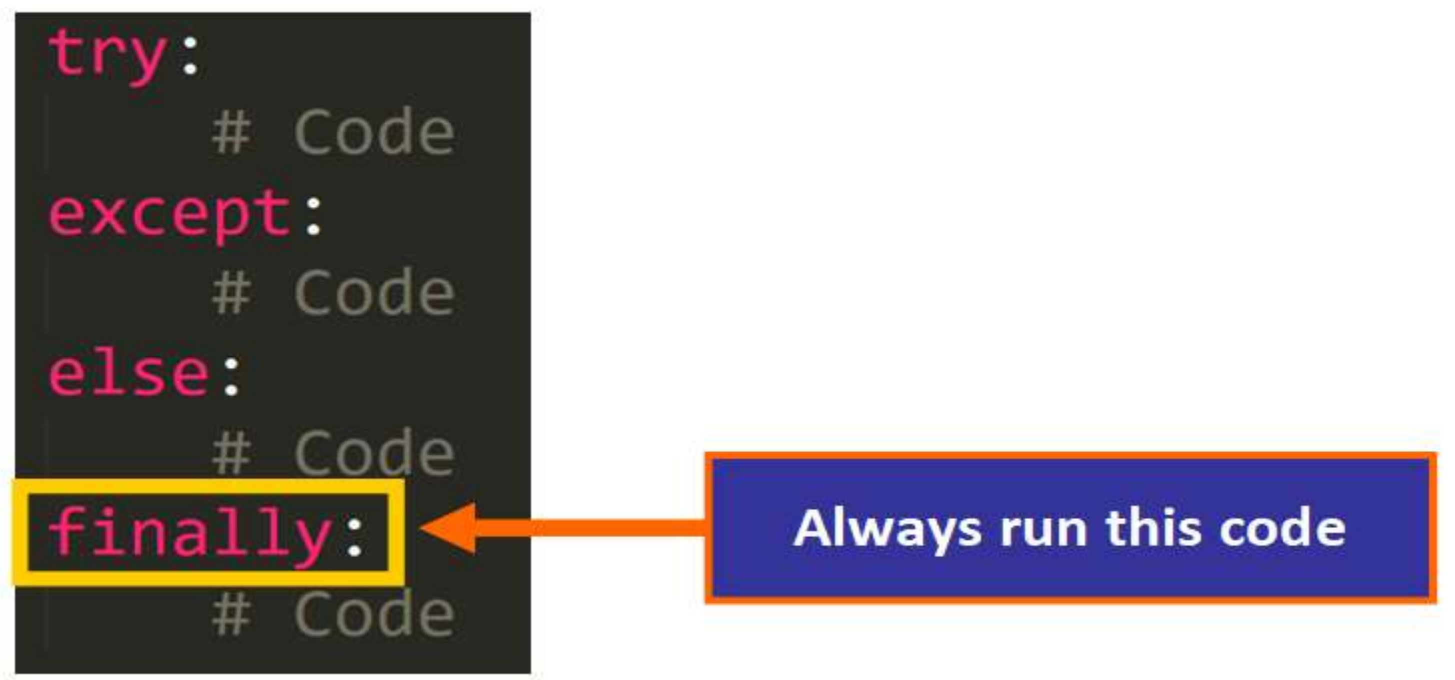
Multiple Exceptions, One Except Clause

- We can also choose to handle different types of exceptions with the same except clause.
- An except clause may name **multiple exceptions** as a parenthesized tuple.

```
try:
    a = int(input("Please enter the numerator: "))
    b = int(input("Please enter the denominator: "))
    print(a / b)
except (ZeroDivisionError, ValueError):
    print("Please enter valid integers.")
```

Python try...finally

- The finally clause is the last clause in this sequence.
- It is optional, but if you include it, it has to be the last clause in the sequence.
- The finally clause is always executed, even if an exception was raised in the try clause.



- If a finally clause is present, the finally clause will execute as the last task before the try statement completes.
- The finally clause **runs whether or not the try statement produces an exception.**
- The finally clause is usually used to perform "clean-up" actions that should always be completed.
- For example, if we are working with a file in the try clause, we will always need to close the file, even if an exception was raised when we were working with the data.

```
try:
    a = int(input("Please enter the numerator: "))
    b = int(input("Please enter the denominator: "))
    result = a / b
except (ZeroDivisionError, ValueError):
    print("Please enter valid integers. The denominator can't be zero")
else:
    print(result)
finally:
    print("Inside the finally clause")
```

Output1: This is the output when no exceptions were raised:

```
Please enter the numerator: 5
Please enter the denominator: 5
1.0
Inside the finally clause
```

Output 2: This is the output when an exception was raised:

```
Please enter the numerator: 5
Please enter the denominator: 0
Please enter valid integers. The denominator can't be zero
Inside the finally clause
```

Note: the else clause and the finally clause are optional, but if you decide to include both, the finally clause has to be the last clause in the sequence.

Raising Exceptions

- Python also provides the raise keyword to be used in the context of exception handling.
- It causes an exception to be generated explicitly.
- Built-in errors are raised implicitly.

Example: The following code accepts a number from the user. The try block raises a ValueError exception if the number is outside the allowed range.

```
try:
    x=int(input('Enter a number upto 100: '))
    if x > 100:
        raise ValueError(x)
    except ValueError:
        print(x, "is out of allowed range")
    else:
        print(x, "is within the allowed range")
```

Output1:

```
Enter a number upto 100: 60
60 is within the allowed range
```

Output 2:

```
Enter a number upto 100: 150
150 is out of allowed range
```

Example 2: enter amount to be withdrawn. If remaining bal <500 raise exception

```
try:
    bal=10000
    wamt=int(input('Enter amount to be withdrawn: '))
    bal=bal-wamt
    if bal<500:
        raise ValueError(bal)
    except ValueError:
        print("balance is less than 500, transaction cancelled")
    else:
        print("transaction succesfull. rem bal",bal)
```

Output1:

```
Enter amount to be withdrawn: 9700
balance is less than 500, transaction cancelled
```

Output 2:

```
Enter amount to be withdrawn: 6000
transaction succesfull. rem bal 4000
```

Nested Exceptions

- Exceptions can be nested.
- A nested exception is an exception that occurs while another exception is being handled.
- When this happens, the processing of the first exception (outer exception) is temporarily suspended.
- Exception handling begins again with the most recently generated exception (inner exception).

Nested try-except Blocks

- We can have nested try-except blocks in Python.
- In this case, if an exception is raised in the nested try block, the nested except block is used to handle it.
- In case the nested except is not able to handle it, the outer except blocks are used to handle the exception.

```
x = int(input("enter numerator x value"))
y = int(input("enter denominator y value"))
 try:
    print("outer try block")
    try:
       print("nested try block")
       print(x / y)
    except TypeError as te:
       print("nested except block")
       print(te)
except ZeroDivisionError as ze:
    print("outer except block")
    print(ze)
```

Output 1: when there is no exception only the outer and nested try blocks are executed but none of the except block is executed and we print output.

```
enter numerator x value10
enter denominator y value2
outer try block
nested try block
5.0
```

Output 2: outer and nested try blocks are executed and outer except block is executed.

enter numerator x value6
enter denominator y value0
outer try block
nested try block
outer except block
division by zero

Advanced Function Calling

Python supports advanced function handling:

1. map statement
2. apply statement

1. map statement: The map() function executes a specified function for each item in an iterable. The item is sent to the function as a parameter.

Syntax:

map(*function*, *iterables*)

- ***Function:*** A mandatory function to be given to map, that will be applied to all the items available in the iterator.
- ***Iterable:*** An iterable compulsory object. It can be a list, a tuple, etc. You can pass multiple iterator objects to map() function.

Return Value: The map() function is going to apply the given function on all the items inside the iterator and return an iterable map object i.e a tuple, a list, etc.

Example-1:

```
def myfunc(n):
    return len(n)

x = map(myfunc, ('apple', 'banana', 'cherry'))

print(x)

print(list(x))
```

Output:

```
<map object at 0x2b38791fb130>
 [5, 6, 6]
```

Example-2:

```
def myfunc(a, b):
    return a + b   a=apple, b=orange

x = map(myfunc, ('apple', 'banana', 'cherry'), ('orange', 'lemon', 'pineapple'))

print(x)

print(list(x))
```

Output:

```
<map object at 0x034244F0>

['appleorange', 'bananalemon', 'cherrypineapple']
```

Example-3: one object is tuple and other object is list

```
def myfunc(a, b):
   return a + b
x=map(myfunc,('apple','banana','cherry'),['orange','lemon','pineapple'])
print(x)
print(list(x))
```

Output:

```
<map object at 0x7f699e9487c0>
['appleorange', 'bananalemon', 'cherrypineapple']
```

2. apply statement: Pandas

This function, acts as a **map** function in python. It takes a function as an input and Applies this function to an entire dataframe. If you are working with tabular data you must specify an axis.

Syntax:

apply(function, axis, args, kwargs)

import pandas

Example:

```
import pandas as pd
df=pd.DataFrame({'A':[1,2,3],'B':[10,20,30]})
def square(x):
  return x*x
df1=df.apply(square)
print(df)
print(df1)
```

Output:

```
   A  B
0  1  10
1  2  20
2  3  30

   A   B
0  1  100
1  4  400
2  9  900
```