# 21CSS101J – PROGRAMMING FOR PROBLEM SOLVING

## UNIT – IV

REFERENCE BOOKS/WEB SITES:


7. Python Datascience Handbook, Oreilly,Jake VanderPlas, 2017. [Chapters 2 &3]

8. Python For Beginners, Timothy C.Needham,2019. [Chapters 1 to 4]

9. https://www.tutorialspoint.com/python/index.htm

10. https://www.w3schools.com/python/

# UNIT – IV

Python: Introduction to Python - Introduction to Google Colab - Basic Data Types: Integers, Floating Points, Boolean types - Working with String functions - Working with Input, Output functions - Python-Single and Multi line Comments/Error Handling - Conditional & Looping Statements : If, for, while statements - Working with List structures - Working with Tuples data structures - Working with Sets - Working with Dictionaries - Introduction to Python Libraries - Introduction to Numpy - High Dimensional Arrays

# Introduction to Python

- The Python programming language was developed in the 1980's by Guido Van Rossum at Centrum Wiskunde & Informatica (CWI) in Netherlands as a successor to the ABC language (itself inspired by SETL) capable of exception handling and interfacing with the Amoeba operating system.
- Programming languages can be classified into the below three categories:

1. Low-level programming languages
2. High-level programming languages
3. Middle-level programming languages

**Python versions:**

- Guido Van Rossum published the first version of Python code (**Python 0.9.0**) at alt.sources in February 1991.
- **Python 1.0** released in January 1994. It included filters, map, reduce and lambda.

  **Python 2.0** released in October 2000. Features include list comprehension and garbage collection with reference cycles.

  **Python 3.0** released in December 2008. Removed duplicate constructs and modules. Not backward compatible.

  **Do not panic.** The features mentioned above will be discussed in detail in the later sections.

  **Python** can be used for creating computer applications, computer games or even for testing microchips.

The instructions (called as source code) written in Python programming language can be executed in two modes:

1. **Interactive mode:** In this mode, lines of code is directly written and executed in the Python interpreter shell, which instantaneously provide the results.
2. **Normal or script mode:** In this mode the source code is saved to a file with .py extension, and the file is executed by the Python interpreter.

- The **Python** program statements have to be entered and saved in a file through a text editor, and then the file has to be executed to produce results.

  If there are any errors in **Python** code then system will inform the same so that they can be corrected.

  There are two types of errors: **Syntax errors** and **Runtime errors**.

- **Syntax errors** are like grammatical errors in English. They occur when the defined rules are not followed.
- **Runtime errors** occur due to incorrect algorithm/logic or wrong operations like divide by zero during the program execution.

# Below are a few interesting ways print(...) function works.

1. **With comma (,) as a separator:** When two or more strings are given as parameters to a print(...) function with a comma (,) as a separator, the strings are appended with a space and printed.

For example:

print("I", "Love", "Python")

will generate the output as

**I Love Python**

2. **With space as a separator:** When two or more strings are given as parameters to a print(...) function with space as a separator, the strings are appended without a space between them.

For example:

print("I"  "Love"  "Python")

will generate the output as

**ILovePython**

3. **With repeat character (*) :** We can print a string n times by using the repeat character (*) as shown below:

print("ABC" * 3);

will generate the output as

**ABCABCABC**

- **Write code to print the magic word Abracadabra, 7 times, using the repeat character (*).**

(Ans: next slide)

- Answer:

```
print("Abracadabra" * 7);
```

- The print statement in the below code is supposed to generate the following output:

where there is a will, there is a way

Answer:

```
print("where", "there", "is", "a", "will," ,"there", "is", "a", "way")
```

# Features of Python Programming Language

- Simple Python
- Easy to Learn Python
- Free and Open Source
- High Level Language
- Interpreted
- Portable Python
- Object Oriented Python
- Extensible
- Embeddable
- Extensive Libraries

# Basics of Python

- **Python** is an Interpreted, Interactive, Object Oriented Programming Language.

  **Python** provides Exception Handling, Dynamic Typing, Dynamic Data Types, Classes, many built-in functions and modules.

  We can also generate Byte Code for **Python**.

  **Python** is used to develop both Web Applications and Desktop Applications.

  **Python** is used in Artificial Intelligence, Scientific Computing and Data Analytics.

  **Python** standard libraries support XML, HTML, Email, IMAP etc..

  **Python** is used to control Firmware updates in Network Programming.

  **Python** is used for Computer Graphics, Cross Platform development, Documentation Development, Data Mining etc..

# Difference between Compiler & Interpreter

| Compiler | Interpreter |
|---|---|
| Compiler takes entire program as input | Interpreter takes single instruction as input |
| Intermediate object code is generated | No Intermediate object code is generated |
| Conditional control statements execute faster | Conditional control statements are slower in execution |
| Memory Requirement is More(Since Object code is generated) | Memory Requirement is Less |
| Program need not be compiled every time | Every time higher level program is converted into lower level program |
| Errors are displayed after entire program is checked | Errors are displayed for every instruction interpreted (if any) |
| Example : C compiler | Example : Python |

# Understanding Python Comments

A computer program is a collection of instructions or statements.

A **Python** program is usually composed of multiple statements. Each statement is composed of one or a combination of the following:

1. Comments
2. Whitespace characters
3. Tokens

In a computer program, a comment is used to mark a section of code as non-executable.

Comments are mainly used for two purposes :

1. To mark a section of source code as non-executable, so that the Python interpreter ignores it.
2. To provide remarks or an explanation on the working of the given section of code in plain English, so that a fellow programmer can read the comments and understand the code.

In **Python**, there are two types of comments :

1. **Single-line comment :** It starts with # (also known as the **hash** or **pound** character) and the content following # till the end of that line is a comment.
2. **Docstring comment :** Content enclosed between triple quotes, either ''' or """. (We will learn about it later).

# Understanding Docstring in Python

- Python provides a way to specify documentation comments, called docstrings. A docstring is a string literal (plain text usually in English) provided in the source code to document what a particular segment of code does.
- A docstring requires the text to be enclosed (surrounded) with triple-quotes, for example:

```
def add(a, b):
        # Comment lines can be present before docstrings.
        """Return the sum of given arguments."""
    return a + b
```

In the above code segment, *Return the sum of given arguments* is the **docstring.**

# Understanding Docstring in Python – Cont.,

- The docstrings are usually provided for Python objects like **modules, classes, functions, members, method definitions, etc.**
- Python allows the use of both """**triple-double-quotes**""" **or** '''**triple-single-quotes**''' to create docstrings. However, the Python coding conventions specification recommends us to use """**triple-double-quotes**""" for consistency.
- The main purpose of docstrings in Python is to provide information on what a particular Python object does and not how it does.
- According to the Python coding conventions, the docstring should always begin with a capital letter and end with a period (.)

We have two types of docstrings in Python, they are :

- **One-line docstring -** provides information about what an object in Python does in a single line.
- **Multi-line docstring** - provides information about what an object in Python does in multiple lines.

# Identifiers and Keywords

- An identifier is a name used to identify a variable, function, class, module, or object.
- In Python, an identifier is like a noun in English.
- Identifier helps in differentiating one entity from the other. For example, name and age which speak of two different aspects are called identifiers.
- Python is a case-sensitive programming language. Meaning, Age and age are two different identifiers in Python.

  Let us consider an example:

  Name = "codeTantra1"

  name = "CodeTantra2"

- Here the identifiers Name and name are different, because of the difference in their case.

# Identifiers and Keywords – Cont.,

Below are the rules for writing identifiers in Python:

1. Identifiers can be a combination of lowercase letters (a to z) or uppercase letters **(A to Z) or digits (0 to 9)** or **an underscore ( _ ).**
2. **myClass, var_1, print_this_to_screen, _number** are valid Python identifiers.
3. An identifier can start with an alphabet or an underscore **(_),** but not with a digit.
4. **1_variable** is invalid, but **variable_1** is perfectly fine.
5. **Keywords** cannot be used as identifiers. (Keywords are reserved words in Python which have a special meaning). (Keywords in Python will be explained later.)
6. Some of the keywords are def, and, not, for, while, if, else and so on.
7. **Special symbols** like !, @, #, $, % etc. are not allowed in identifiers. Only one special symbol underscore (_) is allowed.
8. **company#name, $name, email@id** are invalid Python identifiers.
9. **Identifiers** can be of any length.

# Understanding Python Keywords

- Every programming language usually has a set of words know as **keywords.**
- These are reserved words with **special meaning and purpose.** They are used only for the intended purpose.
- Note : We cannot use a keyword as a variable name, function name or as any other identifier name.
- Python also has its own set of **reserved words called keywords.**
- The interpreter uses the keywords to recognize the structure of the program.
- Python 2 has **32 keywords** while Python 3.5 has **33 keywords.** An extra keyword called nonlocal was added in Python 3.5.
- The **33 keywords** are as follows:

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

# Understanding Variables

- **In Python**, a variable is a reserved memory location used to store values.

For example in the below code snippet, age and city are variables which store their respective values.

```
age = 21
city = "Tokyo"
```

- Usually in programming languages like C, C++ and Java, we need to declare variables along with their types before using them. Python being a dynamically typed language, there is no need to declare variables or declare their types before using them.
- **Python** has no command for declaring a variable. A variable is created the moment a value is assigned to it.
- The **equal-to (=) operator** is used to assign value to a variable.
- **Note:** Operators are special symbols used in programming languages that represent particular actions. = is called the assignment operator.

For example :

```
marks = 100 # Here marks is the variable and 100 is the value assigned to it.
```

# Assigning Different Values to Variables

- Associating a **value** with a **variable** using the assignment operator **(=)** is called as **Binding**.
- In Python, **assignment** creates references, not copies.
- Python uses **reference semantics.**
- We cannot use Python variables without assigning any value.
- If we try to use a variable without assigning any value then, Python interpreter shows **error** like **"name is not defined".**

# Assignment of Variables

- **Python variables** do not need explicit declaration to reserve memory space.
- The declaration happens automatically when you assign a value to a variable.
- The equals to sign **(=)** is used to assign values to variables.
- The operand to the left of the **= operator** is the name of the variable and the operand (or expression) to the right of the **= operator** is the value stored in the variable.
- Let us consider the below example:

```
counter = 100 # An integer assignment
print(counter) # Prints 100
miles = 1000.50 # A floating point assignment
print(miles) # Prints 1000.5
name = "John" # A string assignment
print(name) # Prints John
```

- In the above example 100, 1000.50, and "John" are the values assigned to the variables counter, miles, and name respectively.

# Understanding Multiple Assignment

- Python allows you to assign a single value to several variables simultaneously.

  Let us consider an example:

  **number1 = number2 = number3 = 100**

- Here an **integer object** is created with the value 100, and all the three variables are references to the same memory location. This is called chained assignment.
- We can also assign **multiple objects to multiple variables**, this is called multiple assignment.

  Let us consider the below example:

  value1, value2, value3 = 1, 2.5, "Ram"

  Here the **integer object** with value **1** is assigned to variable **value1,** the **float object** with value 2.5 is assigned to variable value2 and the string object with the value "Ram" is assigned to the variable value3.

# Chained Assignment

- In Python, assignment statements do not return a value. Chained assignment is recognised and supported as a special case of the assignment statement.

- a = b = c = x is called a chained assignment in which the value of x is assigned to multiple variables a, b, and c.

- Let us consider a simple example:

```
a = b = c = d = 10
print(a) # will print result as 10
print(b) # will print result as 10
print(c) # will print result as 10
print(d) # will print result as 10Here,
```

we are initializing the variables a, b, c, d with value 10.

# Understanding Expressions

- An expression is a combination of values(Constants), variables and operators.

- An expression may also include call to functions and objects. We will learn about functions and objects in the later sections.

- Operators are symbols that represent particular actions. Operands participate in the action performed by the operator.

# Understanding Statements – Print Statements

- A **statement** in **Python** is a logical instruction which can be read and executed by **Python interpreter**.
- In **Python**, we have different kinds of statements :

  Print statements

  Assignment statements

  Selective statements

  Iterative statements

  Function declaration statements

- The purpose of Python's **assignment statement** is to associate **variables** with **values** in your program.
- It is the only statement that does not start with a keyword.
- An assignment statement contains at least one equal sign (**=**), to the left hand side of **=** we have a variable and to the right hand side we have an expression.

# Types of Assignment Statements

- There are two types of assignment statements in Python. They are:

    1. Basic assignment statements

    2. Augmented assignment statements

- The simple syntax for a basic assignment statement is:

```
variable_name = expression
```

- In the example given below we are assigning a string value "CodeTantra" to a variable called name.

```
name  =  "CodeTantra"
```

# Augmented Assignment Statement

- Augmented assignment is a **combination of an arithmetic or a binary operation** and an **assignment operation in a single statement.**
- We can combine **arithmetic operators in assignments** to form an augmented assignment statement.
- The combined operations of the augmented assignments are represented using the following operators : +=, -=, *=, /=, %=
- Let us consider a **simple example:** **a += b**
- The above statement is a **shorthand for the below simple statement**. a = a + b
- In augmented assignment statements, the left-hand side is evaluated before the right-hand side
- Let us discuss with a simple example: **a += b**
- **Here the left-hand side i.e. a is evaluated first, then valu**e of b is evaluated and then addition is performed, and finally the **addition result is written back to a.**
- An augmented assignment expression like x += 1 can be rewritten as x = x + 1
- Both the above statements give out **similar** result, but the effect is slightly different.

# Introduction to Google Colab

- Google is quite aggressive in AI research. Over many years, Google developed AI framework called **TensorFlow** and a development tool called **Colaboratory**.
- Colaboratory is now known as Google Colab or simply **Colab**.
- Colab supports GPU and it is totally free.
- Colab is a free Jupyter notebook environment that runs entirely in the cloud.
- Colab supports many popular machine learning libraries which can be easily loaded in your notebook.

# What Colab Offers You?

As a programmer, you can perform the following using Google Colab.

- Write and execute code in Python
- Document your code that supports mathematical equations
- Create/Upload/Share notebooks
- Import/Save notebooks from/to Google Drive
- Import/Publish notebooks from GitHub
- Import external datasets e.g. from Kaggle
- Integrate PyTorch, TensorFlow, Keras, OpenCV
- Free Cloud service with free GPU

# Basic Data Types

- Computer programming is all about processing data. In computer programming, the data is always represented in the binary form (0's and 1's), i.e. groups of **Bits** called as **Bytes**.
- In the real world, we come across different types of data like age of a person(**integer**), number of rooms in a house (**integer**), price(**floating-point number**), height (**floating-point number**), Names of people, places and things (**strings**), inventory list (**list**) etc.
- For easier and efficient processing, the **data** is classified into different types (**data types**)
- In **Python**, like in all programming languages, **data types** are used to classify data to one particular type of data. The data type of the data determines :
- possible values it can be assigned.
- possible operations that can be performed. (Arithmetic operations can be applied on numeric data and not strings)
- the format in which it is stored in the memory.
- the amount of memory allocated to store the data.

- Some of the built-in data types supported in Python are:

1. Number
2. String
3. List
4. Tuple
5. Set
6. Dictionary

- In Python, every value has a data type.
- A variable in Python can hold value of any data type.
- The same variable in Python can refer to data of different data types at different times.
- So variables in Python are not (data) typed.
- Let us consider an example:

```
a = 5 # variable a now refers to Integer data type
a = 90.45 # variable a now refers to Float data type
a = "India"  # variable a now refers to String data type
a = [5, 90.45, "India"]  # variable a now refers to List data type
```

# Numbers

- We have three different categories of numbers in Python. They are :

1. int
2. float
3. complex

1.int – int stands for integer. This Python data type stores signed integers. We can use the type() function to find which class it belongs to.

```
a = -7
print(type(a)) # will print output as follows
<class 'int'>
```

- In Python an integer can be of any length, with the only limitation being the available memory.

```
a = 12536984596965565656236534754587821564
print(type(a)) # will print output as follows
<class 'int'>
```

2.float – float stands for floating-point numbers. This Python data type stores floating-point real values. An int can only store the number 20, but float can store numbers with decimal fractions like 20.25 if you want.

```
a = 3.0
print(type(a)) # will print output as follows
<class 'float'>
```

3.complex – complex stands for complex numbers. This Python data type stores a complex number. A complex number is a combination of a real number and an imaginary number. It takes the form of a + bj Here, a is the real part and b*j is the imaginary part.

```
a = 2 + 3j # It is important to note that there should not be any space between 3 and j
print(type(a)) # will print output as follows
<class 'complex'>
```

# String data type

- In Python, **string** is a sequence of characters enclosed inside a pair of single quotes(‘) or double quotes(“). Even triple quotes (''') are used in Python to represent multi-line strings.
- The computer doesn’t see letters at all. Every letter you use is represented by a number in memory.
- For example, the letter A is actually the number 65. This is called **encoding.** There are two types of encoding for characters – **ASCII and Unicode.**
- **ASCII** uses 8 bits for encoding whereas Unicode uses 32 bits. **Python** uses **Unicode** for character representation.
- An individual character within a string is accessed using an **index**.
- Index starts from 0 to n-1, where n is the number of characters in the string.
- Python allows negative indexing in strings. The index of **-1** refers to the **last item** in the string, **-2** refers to the **second last item** and so on.

# Working with Input, Output functions

- In Python, to read input from the user, we have an in-built function called input().
- The syntax for input() function is :

```
input([prompt])
```

Here, the optional prompt string will be printed to the console for the user to input a value. The prompt string is used to indicate the type of value to be entered by the user.

Reading string inputs

- Let us consider the below example:

```
name = input("Enter your Name: ")
print("User Name:", name)
```

- If the input is given as CodeTantra, the result will be

```
User Name: CodeTantra
```

- We already discussed this function to print output on Screen.
- It is print() function, which is a built-in function in Python.
- We can pass zero or more number of expressions separated by commas(,) to print() function.
- The print() function converts those expressions into strings and writes the result to standard output which then displays the result on screen.
- Let us consider an example of print() with multiple values with comma(,) as separator.

```
print("Hi", "Hello", "Python") # will print output as follows
Hi Hello Python
```

- The print() function is used to print the values to the standard output.
- The general syntax of print() function is:

print(value1, value2..., sep = ' ', end = '\n', file = sys.stdout, flush = False)

where,

- value1,value2...,value n These are the actual data values to printed.

- sep This is the separator used between each value. If there is no separator, then by default 'whitespace' is taken.

- end This is the character which gets printed after all values have been printed. The newline character '\n' is the default.

- file This argument specifies where the output needs to be printed. The screen is the standard output by default

- Python uses C-style string formatting to create new, formatted strings. The % operator also called as 'string modulo operator' is used to format a set of variables enclosed in a "tuple" (a fixed size list), together with a format string, which contains normal text together with "argument specifiers", special symbols like %s and %d.

The general syntax for a format placeholder is:

**%[flags][width][.precision]type**

The following are some basic argument specifiers:

- %s - String (or any object with a string representation, like numbers)
- %d - Integers
- %f - Floating point numbers
- %.<number of digits>f - Floating point numbers with a fixed amount of digits for the decimal part.
- %x or %X - Integers in hexadecimal representation (uppercase/lowercase)
- %o - Octal representation

# Arithmetic Operators

- Python supports the following 7 arithmetic operators.

| Operator | Operation | Expression and its result |
|---|---|---|
| + | Adds values on either side of the operator. | 10 + 20 = 30 |
| - | Subtracts right hand operand from left hand operand. | 20 - 10 = 10 |
| * | Multiplies values on either side of the operator | 11 * 11 = 121 |
| / | Divides left hand operand by right hand operand | 23 / 2 = 11.5 |
| ** | Performs exponential (power) calculation on operators | 2 ** 3 = 8 |
| % | Divides left hand operand by right hand operand and returns remainder | 12 % 5 = 2 |
| // | Floor Division - The division of operands where the result is the quotient in which the digits after the | 23 // 2 = 11 (Integer division) |
| | decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity). | 9.0 // 2.0 = 4.0<br>-11 // 3 = -4<br>-11.0 // 3 = -4.0 |

# Comparison Operators

- **Python** supports the following comparison operators.The result of these comparison operators is either True or False.

| Operator | Operation | Expression (Result) |
|---|---|---|
| == | If the values of two operands are equal, then the condition becomes True. | 23 == 34 (False)<br>10 == 10 (True) |
| != | If the values of two operands are not equal, then the condition becomes True. | 23 != 34 (True)<br>10 != 10 (False) |
| < | If value of left operand is less than value of right operand, then condition becomes true. | 10 < 20 (True)<br>20 < 5 (False) |
| > | If value of left operand is greater than value of right operand, then condition becomes true. | 10 > 20 (False)<br>20 > 5 (True) |
| <= | If value of left operand is less than or equal to value of right operand, then condition becomes true. | 10 <= 20 (True)<br>20 <= 20 (True) |
| >= | If value of left operand is greater than or equal to of right operand, then condition becomes true. | 10 >= 20 (False)<br>20 >= 20 (True) |

- **All the comparison operators work on strings also**. The result is either True or False.
- Python compares strings using **Unicode value** of the characters (lexicographical).
- The comparison is made taking the **ordinal** values of each character in the string and compare it with the **ordinal** values of the character at the same position in the other string.
- If the ordinal value of the character in the first string is greater than the ordinal value of the character in the second string, then the comparison stops and the first string is declared greater than the second string. The length of the string does not matter.
- In Python, the ordinal value of a character can be found using the ord() function, which takes the character as an argument.
- Write a program to understand the use of comparison operators using conditional parameters. Take input from user using input() method.

# Assignment Operators

- Assignment Operators are used to assign values to variables.
- a = 52 is a simple assignment operator that assigns the value 52 on the right to the variable a on the left.

| Operator | Description | Expression |
|---|---|---|
| = | Assigns values from right side operands to left side operand. | c = a + b assigns value of a + b into c |
| += | Adds right operand to the left operand and assign the result to left operand. | c += a, equivalent to c = c + a |
| -= | Subtracts right operand from the left operand and assign the result to left operand. | c -= a, equivalent to c = c - a |
| *= | Multiplies right operand with the left and assign the result to left operand. | c *= a, equivalent to c = c * a |
| /= | Divides left operand with the right and assign the result to left operand. | c /= a, equivalent to c = c / a |
| **= | **Performs exponential calculation on left and right operand c **= a, equivalent to c = c **a and assign the result to left operand.** | |
| %= | **Performs modulo division of left and right operands and c %= a, equivalent to c = c % a ,assign the result to left operand.** | |
| //= | **Performs floor division on left and right operands and c //= a, equivalent to c = c // a ,assign the result to left operand.** | |

# Bitwise Operators

- Numbers can be used in many forms like **decimal**, **hexa**, **octal** and **binary**. Computers store the numbers in the binary format.

  **Numbers in binary format :**

  2 is "10"

  3 is "11"

  4 is "100"

  678 is "1010100110“

- **Python Bitwise Operators** take one or two operands, and operate on them bit by bit, instead of whole.
- Following are the bitwise operators in Python

  1. << (Left shift) - Multiply by 2 ** number of bits

  Example: **x = 12** and **x << 2** will return **48** i.e. **(12 * (2 ** 2))** This is similar to multiplication and more efficient than the regular method

  2. >> (Right shift) - divide by 2 ** number of bits

  Example: **x = 48** and **x >> 3** will return **6** i.e. **(48 / (2 ** 3))** This is similar to division by powers of 2 (2, 4, 8, 16 etc.)

Contn.,

3. & (AND) - If both bits in the compared position are 1, the bit in the resulting binary representation is 1 (1 × 1 = 1) i.e. True; otherwise, the result is 0 (1 × 0 = 0 and 0 × 0 = 0) i.e. False.

Example : x & y Does a "bitwise and". If the bit in x and the corresponding bit in y are 1, then the bit in the result will be 1. otherwise it will be zero.

2 & 5 will result in zero because (010 & 101 = 000). 3 & 5 will result in 1 because (011 & 101 = 001)

4. | (OR) - If the result in each position is 0(False) if both bits are 0, while otherwise the result is 1(True).

Example : x | y Does a "bitwise or". If the bit in both operands is zero, then the resulting bit is zero. otherwise it is 1.

2 | 5 will result in 7 because (010 |101 = 111) and 3 | 5 will also result in 7 because (011 | 101 = 111)

5. ~ (NOT) - The bitwise NOT, or complement, is a unary operation that performs logical negation on each bit, forming the ones' complement of the given binary value. Bits that are 0 become 1, and those that are 1 become 0.

Example : ~ Returns the complement of x . If the bit is 1 it will become zero and if the bit is zero it will become 1.

~5 will result in 2 because (~101 = 010) and ~2 will become 5 because (~010 = 101). This is true only for unsigned integers.

6. ^ (Bitwise XOR) - If the comparison of two bits, being 1 if the two bits are different, and 0 if they are the same.

Example: Does a "bitwise exclusive or". If the bit in in either operands is 1, but not in both, then the resultant bit will be 1. Otherwise it will be 0.

5 ^ 3 will result in 6 because (101 ^ 011 = 110)

# Logical Operators

- **Logical Operators** : A logical operator is derived from boolean algebra where the result is either True or False.

  It is generally used to represent logical comparison the way it is done in real life.

  1 represents True and 0 represents False. (Internally 0 represents False and anything else represents True)

  The logical operators supported in Python are similar to the logic gates and are as follows:

  1. and - Logical AND : The result is True, only if both the operands are True. ( 1 and 1 is the only condition where the result is 1, any other combination results in 0.)

  2. or - Logical OR : The result is True, if any one of the operands is True. ( 0 and 0 is the only condition where the result is 0, any other combination results in 1.)

  3. not - Logical NOT : The result negates the operand, i.e if the operand is True, the result is False and vice versa

# Membership Operators

- The operators in and not in test for membership. x in s evaluates to True if x is a member of s, and False otherwise.
- The Right Hand Side (RHS) can be a String, List, Tuple, Set or a Dictionary.
- For strings, the Left Hand Side (LHS) can be any string. If this string exists in the RHS string, then True is returned. Otherwise False is returned.
- For all other data types (Lists, Tuples, Sets, Dictionaries, etc.) the LHS should be a single element.
- Let's consider an example for strings:
- 'am' in 'I am working' will return True.
- 'am' not in 'I am working' will return False.

# Conditional & Looping Statements

- **Python** provides special constructs to control the execution of one or more statements depending on a condition. Such constructs are called as control statements or control-flow statements.
- The control-flow statements are of three types:
- Selection Statement - is a statement whose execution results in a choice being made as to which of two or more paths should be followed.
  - if construct
  - if-else construct
  - if-elif-else construct
  - Nested if-elif-else construct
- Iterative Statement - is a statement which executes a set of statements repeatedly depending on a condition.
  - while loop
  - for loop
  - else clause on loop statements
- Control flow Statement - is a statement which transfers control-flow to some other section of the program based on a condition.
  - break statement
  - continue statement
  - pass statement

# If statement: Understanding IF construct

- The general syntax of if statement in Python is,

  If test expression:

  statement(s)

- The if-construct is a selection statement, the statements within the block are executed only once when the condition evaluates to True, Otherwise, the control goes to the first statement after the if-construct.
- In Python, the body (block of statements) of the If statement is indicated by indentation.
- The body starts with indentation and the first unindented line marks the end.
- Python interprets non-zero values as True. None and 0 are interpreted as False.
- Here is a simple program to illustrate the simple if-statement:

```
num = int(input("num: "))
if (num % 3 == 0):
        print("divisible by 3") # Notice the Indentation
print("End of program")
```

# if-else statement

- The if-else statement provides two different paths of execution depending on the result of the condition.
- The body of if is executed when the condition associated with the expression is true.
- The body of else part is executed when the condition is evaluated to false.
- Indentation is used to separate both the if and else blocks.
- Below is the general syntax for the if-else statement :

```
if(expression):
        body of If
else:
        body of else
```

# Write a program….

- Write a program to check whether the marks obtained by the student got distinction or not. Take marks as input from the user which of type int.
- Follow the below conditions while writing the program.
  - If marks > distinction _marks print the message as distinction
  - Otherwise print the message not distinction
  - Print the result to the console as shown in the examples.
- Sample Input and Output 1:

  marks: 78

  distinction
- Sample Input and Output 2:

  marks: 55

  not distinction

# Program:

```
distinction_marks = 75
marks= (int) (input("Enter marks obtained: "))
if(marks>75):
        print("User secured distinction")
else:
        print("User did not secure distinction")
```

# if-elif-else construct

- The if-elif-else construct extends the if-else construct by allowing to chain multiple if constructs as shown below:

```
if test expression:
        body of if
elif test expression:
        body of elif
elif test expression:
        body of elif
   ...
elif test expression:
        body of elif
else:
        body of else
```

- The if elif else construct is used when we have multiple mutually exclusive expressions.
- If the condition for it is false, then the condition for the next elif is evaluated, and so on up to the next elif.
- If all the conditions are false, then the body of else is executed.
- Only one block among if elif else blocks is executed based on the condition.
- The if block can have only one else block, but it can have multiple elif blocks.
- Indentation is used for each of the if-elif-else blocks

# Write a program....

- Take character as input from the console using input() function. Write a program to check whether the given input is a character or a digit, if the input is 0 exit the program, otherwise print the result to the console as shown in the examples.
- Sample Input and Output 1:

'0' for exit.

ch: 7

digit

- Sample Input and Output 2:

'0' for exit.

ch: D

alphabet

- Sample Input and Output 3:

'0' for exit.

ch: @

neither alphabet nor digit

# Program

```
print("Enter '0' for exit.")
# take th input from the user
ch=input('Enter any character: ‘)
if ch == '0’:
        exit()
else:
        if(ch.isalpha()==True):
                print("Given character",ch,"is an alphabet")
        elif(ch.isdigit()==True):
                print("Given character",ch,"is a digit")
        else:
                print(ch,"is not an alphabet nor a digit")
# write your logic here to find the given input is character or digit
```

# While Loop

- Syntax of while loop in Python:

```
while test expression:
    body of while
```

- A while statement is used to execute some block of code repeatedly as long as a condition evaluates to True.
- In the While loop, the value of the expression is evaluated first.
- The body of the loop is entered only when the expression evaluates to true.
- After one iteration, the expression is checked again.
- This process continues until the expression evaluates to False.
- The body of the loop is identified using the indentation with the expression condition as the first statement.

# Write a Program…

- Take an integer num as input from the user. Write a code to find sum of all even numbers up to num using while construct, print the result to the console as shown in the example.
- Sample Input and Output:

  num: 100

  sum: 2550
- Hint:

  sum = 2 + 4 + 8 + ……. + n (if n is even)

  sum = 2 + 4 + 8 + ……. + n - 1 (if n is odd)

# Program...

```
n = (int)(input("Enter a positive integer: "))
sum=0
i = 2
while (i <= n):
        sum = sum + i
        i = i + 2
print("The sum of even numbers is:",sum)
```

# While loop with else:

- We can have an optional else block with while loop as well.
- The else part is executed if the condition in the while loop evaluates to False.
- The while loop can be terminated with a break statement. In such a case, the else part is ignored.
- Hence, a while loop's else part runs if no break occurs and the condition is False.

```
while condition:
        statement_1
        ...
        statement_n
else:
        statement_1
        ...
        statement_n
```

# Write a program….

- Take an integer num as input from the console using input() function.
- Write a program to find the sum of all integers between 0 and num, both inclusive.
- Print the result to the console as shown in the examples.
- Sample Input and Output 1:

  num: 250

  sum: 31375

- Sample Input and Output 2:

  num: -660

  sum: -218130

# Program...

```
# Python program to find the sum of integers between 0 and n where n is provided by user
n = int(input("Enter a number: "))
inp = n
i = 1
n = abs(n)
s = 0
while i<=n:
        s=s+I
        i=i+1
else:
        if inp >= 0:
                print("The sum is",s)
        else:
                print("The sum is", -s)
```

# For Loop

- A for-loop is used to iterate over a range of values using a loop counter, which is a variable taking a range of values in some orderly sequence (e.g., starting at 0 and ending at 10 in increments of 1).
- The value stored in a loop counter is changed with each iteration of the loop, providing a unique value for each individual iteration. The loop counter is used to decide when to terminate the loop.
- A for-loop construct can be termed as an entry controlled loop.
- For loop Syntax:

```
for val in sequence:
    Body of for
```

- Here val is the variable that takes the value of the item in the sequence for each iteration.
- The Loop continues until the last item in the sequence is reached.
- The body of the for loop is separated from the rest of the code using indentation.

# Example:

- Let us consider a simple example:

```
items = "Python"
index = 0
for item in items:
    print(index, item)
    index += 1
```

- Here "Python" is a string, we can iterate a string using for loop.
- we take 'index' variable to print the index value of each character.
- The output for the above for loop:

```
(0, 'P')
(1, 'y')
(2, 't')
(3, 'h')
(4, 'o')
(5, 'n')
```

# Using range function with For loop

- The range() function :
- We can generate a sequence of numbers using range() function.
- We can use the range() function in for loop to iterate through a sequence of numbers.
- It can be combined with the len() function to iterate through a sequence using indexing.
- Syntax:

```
range(stop) :
```

- Returns a sequence of numbers starting from 0 to (stop - 1)
- Returns an empty sequence if stop is negative or 0.
- range(6) will generate numbers from 0 to 5 (up to 6 not including 6).
- Let us consider a simple example:

```
for i in range(1, 6):
    print(i)
```

- Observe the following output:

```
1
2
3
4
5
```

# For loop with else

- A for loop can have an optional else block as well.
- The else part is executed if the items in the sequence used in for-loop exhausts.
- A break statement can be used to stop a for-loop. In such case, the else part is ignored.
- Hence, a for-loop's else part runs if no break occurs.
- A common construct to search for an item is to use a for-loop which can terminate in two scenarios,
- If the item is found and a break is encountered, exit out of the for-loop.
- If the item is not found, then the loop completes.
- So, in order to find out how the loop exits, the else clause is useful.
- Basic structure of a for-else construct

```
for i in range(1, 10):
        print(i)
else:  # Executed because no break in for
        print("No Break")
```

- In the above example, it prints the numbers from **1** to **9** and then else part is executed, and prints No Break.

# Let us consider another example……

```
for i in range(1, 10):
        print(i)
        break
else:  # else part is  not executed, because there is break statement in the for loop
        print("No Break")
```

- In the above example the output is 1 only.

# Working with List structures

- List is a data type of Python and is used to store sequence of items.
- The items of a list need not be of the same data type.
- Lists are ordered sequence of items.
- Let us consider an example:

  L1 = [56, 78.94, "India"]

- In the above example, L1 is a list, which contains 3 elements. The first element is 56 (integer), the second is 78.94 (float), third is "India" (string).
- Lists are ordered, we can retrieve the elements of a List using "index".
- A List in general is a collection of objects. A List in **Python** is an ordered group of items or elements.
- List can be arbitrary mixture of types like numbers, strings and other lists as well.

# The main properties of Lists are :

1. Elements in a list are ordered
2. They can contain objects of different types
3. Elements of a list can be accessed using an **index** similar to accessing an element in an array
4. A List can contain other lists as their elements
5. They are of variable size i.e they can **grow** or **shrink** as required
6. They are **mutable** which means the elements in the list can be changed/modified

## Basic List Operations:

| Operations | Example | Description |
|---|---|---|
| **Create a List** | a = [2 ,3, 4, 5, 6, 7, 8, 9, 10] print(a) [2 ,3, 4, 5, 6, 7, 8, 9, 10] | Create a comma separated list of elements and assign to variable a |
| **Indexing** | print(a[0]) 2 print(a[8]) 10 print(a[-1]) 10 | Access the item at position 0<br>Access the item at position 8<br>Access the last item using negative indexing |
| **Slicing** | print(a[0:3]) [2, 3, 4] print(a[0:]) [2, 3, 4, 5, 6, 7, 8, 9, 10] | Print a part of list starting at index 0 till index 2<br>Print a part of the list starting at index 0 till the end of list |
| **Concatenation** | b = [20, 30] print(a + b) [2, 3, 4, 5, 6, 7, 8, 9, 10, 20, 30] | Concatenate two lists and show its output |
| **Updating** | a[2] = 100 print(a) [2, 3, 100, 5, 6, 7, 8, 9, 10] | Update the list element at index 2 |
| **Membership** | a = [2, 3, 4, 5, 6, 7, 8, 9, 10] 5 in a True 100 in a False 2 not in a False | Returns True if element is present in list.<br>Otherwise returns false. |
| **Comparison** | a = [2, 3, 4, 5, 6, 7, 8, 9, 10] b = [2, 3, 4] a == b False a != b True | Returns True if all elements in both lists are same.<br>Otherwise returns false |
| **Repetition** | a = [1, 2, 3] a * 0 [ ] a * 2 [1, 2, 3, 1, 2, 3] print(a * 3) [1, 2, 3, 1, 2, 3, 1, 2, 3] | Here the * operator repeats a list for the given number of times. |

# Understanding List slicing:

| Slices | Example | Description |
|---|---|---|
| **a[0:3]** | a = [9, 8, 7, 6, 5, 4]<br>a[0:3]<br>[9, 8, 7] | Print a part of list from index 0 to 2 |
| **a[:4]** | a[:4]<br>[9, 8, 7, 6] | Default start value is 0.<br>Prints the values from index 0 to 3 |
| **a[1:]** | a[1:]<br>[8, 7, 6, 5, 4] | Prints values from index 1 onwards |
| **a[:]** | a[:]<br>[9, 8, 7, 6, 5, 4] | Prints the entire list |
| **a[2:2]** | a[2:2]<br>[] | Prints an empty slice |
| **a[0:6:2]** | a[0:6:2]<br>[9, 7, 5] | Slicing list values with step size 2 |
| **a[::-1]** | a[::-1]<br>[4, 5, 6, 7, 8, 9] | Prints the list in reverse order |
| **a[-3:]** | a[-3:]<br>[6, 5, 4] | Prints the last 3 items in list |
| **a[:-3]** | a[:-3]<br>[9, 8, 7] | Prints all except the last 3 items in list |

# Built in List functions:

- Here is a list of functions that can be applied to list:
- all() - Returns True if all the items in the list have a True value.

  print(all([' ', ',', '1', '2’])) - True
- any() - Returns True if even one item in the list has a True value.

  print(any([' ', ',', '1', '2’])) - True
- enumerate() - Returns an enumerate object consisting of the index and value of all items of the list as a tuple pair.

  print(list(enumerate(['a','b','c','d','e'])))

  [(0, 'a'), (1, 'b'), (2, 'c'), (3, 'd'), (4, 'e')]
- len() - This function calculates the length i.e., the number of elements in the list.

  print(len(['a', 'b', 'c', 'd', 'e']))

  5
- list() - This function converts an iterable (tuple, string, set, dictionary) to a list

  print(list("abcdef"))

  ['a', 'b', 'c', 'd', 'e', 'f’]

  print(list(('a', 'b', 'c', 'd', 'e')))

  ['a', 'b', 'c', 'd', 'e']

- max() - This function returns the item with the highest value from the list

```
print(max([1, 2, 3, 4, 5]))
5
```

- min() - This function returns the item with the lowest value from the list.

```
print(min([1, 2, 3, 4, 5]))
1
```

- sorted() - This function returns a sorted result of the list, with the original list unchanged.

```
origlist = [1, 5, 3, 4, 7, 9, 1, 27]
sortedlist = sorted(origlist)
print(sortedlist)
[1, 1, 3, 4, 5, 7, 9, 27]
print(origlist)
[1, 5, 3, 4, 7, 9, 1, 27]
```

- sum() - This function returns the sum of all elements in the list.
- This function works only on numeric values in the list and will error out if the list contains a mix of string and numeric values.

```
print(sum([1, 5, 3, 4, 7, 9, 1, 27]))
57
print(sum([1, 3, 5, 'a', 'b', 4, 6, 7]))
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str'
```

# Working with Tuples data structures

- A tuple is a sequence which is similar to a list, except that these set of elements are enclosed in parenthesis ()
- Once a tuple has been created, addition or deletion of elements to a tuple is not possible(immutable).
  A tuple can be converted into a list and a list can be converted into a tuple

| Functions | Example | Description |
|---|---|---|
| list() | a = (1, 2, 3, 4, 5) a = list(a) print(a)<br>[1, 2, 3, 4, 5] | Convert a given tuple into list using list() function. |
| tuple() | a = [1, 2, 3, 4, 5] a = tuple(a) print(a)<br>(1, 2, 3, 4, 5) | Convert a given list into tuple using tuple() function |

# Benefits of Tuple:

- Tuples are faster than lists.
- Since a Tuple is immutable, it is preferred over a list to have the data write-protected.
- Tuples can be used as keys in dictionaries unlike lists.
- Tuples can contain list as an element and the elements of the list can be modified, because lists are mutable.
- Let's consider a example:

```
t1 = (1, 2, 3, [4, 5], 5)
t1[3][0] = 42 # will result as follows
(1, 2, 3, [42, 5], 5)  # because list elements can be changed
```

# Understanding basic tuple operations

| Operations | Example | Description |
|---|---|---|
| **Creating a Tuple** | a = (20, 40, 60, "apple", "ball") t1 = (1,) print(t1) (1,) | Creating tuple with different data types.<br>To create a tuple with a single element, there should be a final comma. |
| **Indexing** | print(a[0]) 20 print(a[2]) 60 | Accessing the item at position 0.<br>Accessing the item at position 2. |
| **Slicing** | print(a[1:3]) (40,60) | Displaying items from position 1 till position 2. |
| **Concatenation** | b = (2, 4) print(a + b) (20, 40, 60, "apple", "ball", 2, 4) | print the concatenation of two tuples. |
| **Repetition** | b = (2, 4) print(b * 2) (2, 4, 2, 4) | repeating the tuple 'n' no. of times. |
| **Membership** | a = (2, 3, 4, 5, 6, 7, 8, 9, 10) print(5 in a) True print(100 in a) False print(2 not in a) False | Returns True if element is present in tuple. Otherwise returns false. |
| **Comparison** | a = (2, 3, 4, 5, 6, 7, 8, 9, 10) b = (2, 3, 4) print(a == b) False print(a != b) True | Returns True if all elements in both tuples are same otherwise returns false. |

# Understanding the Built in Tuple Functions

- min() - This function returns the item with the lowest value in the tuple

```
print(min((1, 2, 3, 4, 5, 6)))
1
```

- sorted() - This function returns a sorted result of the tuple which is a list, with the original tuple unchanged.

```
origtup = (1, 5, 3, 4, 7, 9, 1, 27)
sorttup = sorted(origtup)
print(sorttup)
[1, 1, 3, 4, 5, 7, 9, 27]
print(origtup)
(1, 5, 3, 4, 7, 9, 1, 27)
```

- sum() - This function returns the sum of all elements in the tuple.
- This function works only on numeric values in the tuple and will error out if the tuple contains a mix of string and numeric values.

```
print(sum((1, 5, 3, 4, 7, 9, 1, 27)))
57
print(sum((1, 3, 5, 'a', 'b', 4, 6, 7)))
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'int' and 'str’
```

Contn.

- tuple() - This function converts an iterable (list, string, set, dictionary) to a tuple

```
x = list("abcdef")
print(x)
['a', 'b', 'c', 'd', 'e', 'f']
print(tuple(x))
('a', 'b', 'c', 'd', 'e', 'f')
```

# Working with Sets

- A set is a disordered collection with unique elements.
- The elements in a Set cannot be changed (immutable).
- Hence, a set cannot have a mutable element, like a list, set or dictionary as its element.
- The set is mutable which means elements can be added, removed or deleted from it.
- The main operations that can be performed on a set are:
  - Membership test
  - Eliminating duplicate entries.
  - Mathematical set operations like union, intersection, difference and symmetric difference.

- A Set can be created by placing all the items or elements inside curly braces {} each separated by a comma (,).
- The set() built-in function can also be used for this purpose.
- The Set can contain any number and different types of items - **integer**, **float**, **tuple**, **string** etc.

```
numset = {1, 2, 3, 4, 5, 3, 2}
print(numset)
{1, 2, 3, 4, 5}

numset2 = set([1, 2, 3, 2, 4, 5])
print(numset2)
{1, 2, 3, 4, 5}

emptyset = set()
print(type(emptyset))
<class 'set'>
```

- Let's discuss how to create a set using user-given elements.
- Follow the given instructions and write the code.
- Steps to be followed:

1. Take an input from user using input() function.(with comma separated).
2. Using split() function convert the given input into list using a separator(Here is ,(comma)).
3. Convert the list into set using set()
4. Print the obtained set in sorted order using sorted() function to convert a set into sorted order. When we try to convert a set into sorted order it returns a sorted list

- Let's consider a simple example:

```
set1 = {45, 89, 65, 3, 47, 400, 2, 963, 1, 963}
print(set1) # will print the result as {65, 2, 3, 963, 1, 45, 47, 400, 89}
print(sorted(set1)) # will print the result as [1, 2, 3, 45, 47, 65, 89, 400, 963]
```

Here when we convert the set into sorted order using sorted() it returns sorted list.

# Add Elements to a Set

- Create a set with the user given inputs. Take an element from the user and add that element to the set. Similarly, create a list by taking the inputs from the user and update the list with the set, print the result as shown in the example.
- Sample Input and Output:

data1: 1,2,3,4,5

element: 456

sorted set after adding: ['1', '2', '3', '4', '456', '5']

data2: 77,88,99

sorted set after updating: ['1', '2', '3', '4', '456', '5', '77', '88', '99']

- Note: Please print the set in sorted order (sorted(setname)), otherwise the test cases will fail because sets are not ordered and can be printed in random order.

# Remove Elements in a Set

- Create a set with the user given input values. Write a program to remove the elements from the set using the methods discard(), remove() and pop(), print the result as shown in the example. If the element is not present in the set print the error message as shown in the example.
- Sample Input and Output 1:

  data1: 10,20,30,40,50
  element to discard: 30
  sorted set after discarding: ['10', '20', '40', '50']
  element to remove: 40
  sorted set after removing: ['10', '20', '50']
- Sample Input and Output 2:

  data1: Blue,Green,Red
  element to discard: Orange
  not in set
- Note: We are not using pop() in the below program as it randomly removes an element.

# Introduction to Frozenset

- As we have seen that a set is mutable which can allow addition/deletion/removal of elements in it.
- The elements in a set should be immutable i.e., they cannot be modified/changed.
- Hence, Lists cannot be used as elements of a set.

```
set1 = set((("C", "C++"), ("Java", "OOPS")))
print(type(set1))
<class 'set'>
set2 = set((["C", "C++"], ["Java", "OOPS", "Scala"]))
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
set2 = set((["C", "C++"], ["Java", "OOPS", "SCALA"]))
TypeError: unhashable type: 'list'
```

- Another type of set exists called the frozenset which is an immutable set (which means addition/deletion/removal of elements is not possible.)

```
cities = frozenset(["Hyderabad", "Bengaluru", "Pune", "Kochi"])
print(type(cities))
<class 'frozenset'>
print(cities.add("London"))
Traceback (most recent call last):
 File "<stdin>", line 1, in <module>
cities.add("London")
AttributeError: 'frozenset' object has no attribute 'add'
```

# Membership Test

- Create a set by taking inputs from the user. Write a program to check if the user-given element is existing in the set or not, and print the result as shown in the examples.
- Sample Input and Output 1:

  data1: 1,2,3,4,5,6,7
  sorted set: ['1', '2', '3', '4', '5', '6', '7']
  element: 7
  is 7 in set: True
- Sample Input and Output 2:

  data1: 11,22,33,44,55,66
  sorted set: ['11', '22', '33', '44', '55', '66']
  element: 77
  is 77 in set: False
- Note: Please print the set in sorted order (sorted(setname)), otherwise the test cases will fail because sets are not ordered and can be printed in random order. The output will be printed as a list as sorted(set) returns a list.

# Mathematical Set Union

- Create three sets set1, set2 and set3 by taking the inputs from the user. Find the Uninon of three sets using the method union() and also using the operator |, print the result as shown in the example.
- Sample Input and Output:

```
data1: 1,2,3
data2: 4,5,6
data3: 7,8,9
set1 sorted: ['1', '2', '3']
set2 sorted: ['4', '5', '6']
set3 sorted: ['7', '8', '9']
union of set1, set2: ['1', '2', '3', '4', '5', '6']
union of set1, set2, set3: ['1', '2', '3', '4', '5', '6', '7', '8', '9']
set1 | set2: ['1', '2', '3', '4', '5', '6']
set1 | set2| set3: ['1', '2', '3', '4', '5', '6', '7', '8', '9']
```

- Note: Please print the set in sorted order (sorted(setname)), otherwise the test cases will fail because sets are not ordered and can be printed in random order. The output will be printed as a list as sorted(set) returns a list.

# Build in Set Functions & Methods

- Given below are few of the methods and functions that are applicable to a set.

```
len(set) -  Returns the number of elements in set
aset = {'a', 'b', 'a', 'c', 'd', 'e','a'}
print(aset)
{'e', 'b', 'd', 'a', 'c'}
print(len(aset))
5
```

- copy() - Return a new set with a shallow copy of set

```
x = {1, 2, 3, 4, 5, 6, 7}
print(len(x))
7
y = x.copy()
print(y)
{1, 2, 3, 4, 5, 6, 7}
```

- Copy is different from assignment. Copy will create a separate set object, assigns the reference to y and = will assign the reference to same set object to y

```
print(x.clear())
print(y)
{1, 2, 3, 4, 5, 6, 7}     # if we used y=x then y will also be cleared
```

- isdisjoint(otherset) - Returns True if the Set calling this method has no elements in common with the other Set specified by otherset.

  a = {1,2,3,4}
  b= {5, 6}
  a.isdisjoint(b)

  will return True because there are are no common elements

  which means a & b (intersection of a, b is a null set)

- issubset(otherset) - Returns True if every element in the set is in other set specified by otherset.

```
x = {"a","b","c","d","e"}
y = {"c", "d"}
print(x.issubset(y))
False
print(y.issubset(x))
True
print(x < y)
False


print(y < x)
True
print(x < x)
False
print(x <= x)
True
```

- issuperset(otherset) - Returns True if every element in other set specified by otherset is in the set calling this method.

```
x = {"a","b","c","d","e"}
y = {"c", "d"}
print(x.issuperset(y))
True
print(x > y)
True
print(x >= x)
True
print(x > x)
False
print(x.issuperset(x))
True
```

# Working with Dictionaries

- Dictionary is an unordered collection of key and value pairs.
- We use Dictionaries to store key and value pairs such as countries and capitals, cities and population, goods and prices etc.
- The keys should be unique, but the values can change (The price of a commodity may change over time, but the name of the commodity will not change).
- That is why we use immutable data types (Number, string, tuple etc.) for the key and any type for the value.
- Dictionary is an disordered collection of elements. Dictionary contains elements in the form of (key and value) pairs.
  - Unlike sequences which are indexed by a range of numbers, dictionaries are indexed by keys.
  - The element in a dictionary is of type key:value pair.
  - Hence, a dictionary is a set of disordered collection of comma separated key:value pairs which are enclosed within {} braces.
  - The requirement for the Key is it should be immutable and unique and can be of types - strings, numbers, tuples.
  - Tuples can also be Keys if they contain only strings, numbers or tuples.
  - If any tuple contains a mutable object (such as a list), it cannot be used as a Key.
  - Usually a pair of braces { } represents an empty dictionary.
  - Elements can be added, changed or removed by using the key.

- The typical operations that can be performed on a dictionary are:

- Adding elements i.e., Key:Value pairs.
- Accessing elements using the Key with the Index operator [] or using get() method.
- Updating the value of a particular element given in the Key.
- The elements can also be deleted which results in a deletion of both the key:value pair or deleting the entire dictionary using **del()** method.
- Iterating through a dictionary for processing the elements in it.

- Deleting an element from a dictionary.
- An element can be removed from the dictionary using the pop() method which takes a Key as argument and returns the value associated.
- If an invalid/non-existent key is provided with pop(), then a TypeError is shown.
- The popitem() method can be used to remove and return an arbitrary item (key, value) from the dictionary.
- All items can be removed from the dictionary using the clear() method.
- This in fact makes the dictionary empty and doesn't delete the variable associated.
- The associated variable reference can be reused.
- The del keyword is used to delete an item using index operator [] with key or delete the entire dictionary.
- Trying to reference the dictionary variable after it has been deleted results in a NameError.

- As a Dictionary is mutable,
- New elements can be added and an existing element can be changed using the assignment operator [ ].
- The Key is used with the assignment operator.
- If the key already exists in the dictionary, the associated value gets updated.
- or else a new key:value pair gets added to dictionary.

# Introduction to NumPy

List of points regarding NumPy:

- NumPy(Numerical Python) is a critical data science library in Python
- It is an open-source library for working efficiently with arrays.
- Developed in 2005 by Travis Oliphant
- Mathematical operations on NumPy's nd-array objects are up to 50x faster than iterating over native Python lists using loops.
- NumPy has huge efficiency gains primarily due to storing array elements in an ordered single location within memory.
- It eliminates redundancies by having all elements of the same type and making full use of modern CPUs.
- Operating on arrays with thousands or millions of elements are pretty easy in NumPy.
- It offers an Indexing syntax for easily accessing portions of data within an array.
- The built-in functions in NumPy improves quality of life when working with arrays and math, such as functions for linear algebra, array transformations, and matrix math.

Cont.,

- After NumPy, the next logical choices for growing data science and scientific computing capabilities are SciPy and pandas.

- NumPy provides a foundation on which other data science packages are built, including SciPy, Scikit-learn, and Pandas.

- SciPy provides a menu of libraries for scientific computations. It extends NumPy by including integration, interpolation, signal processing, more linear algebra functions, descriptive and inferential statistics, numerical optimizations, and more.

- Scikit-learn extends NumPy and SciPy with advanced machine-learning algorithms.

- Pandas extends NumPy by providing functions for exploratory data analysis, statistics, and data visualization( similar to Microsoft Excel spreadsheets) for working with and exploring tabular data.

- NumPy with other Python libraries like Matplotlib can be considered as a fully-fledged alternative to MATLAB's core functionality.

# High Dimensional Arrays

Lists vs NumPy Arrays:

- Users of Python may wonder why NumPy arrays are needed when Python lists already exist.
- Lists in Python operate as an array that can contain various types of elements. A perfectly reasonable question has a logical answer hidden in how Python stores objects in the memory. **A Python object is actually a pointer to a memory location where all the object's details, such as its bytes and value, are stored.** This additional information is what makes Python a dynamically typed language, it also comes at a cost which becomes apparent when storing a large collection of objects, like in an array.
- Python lists are an array of pointers where each pointer pointing to a location contains the relevant information to the element. This thing will significantly increase the memory and calculation overhead. When all the objects stored in the list are of the same type then the majority of this information is rendered redundant.
- To overcome this issue, we use NumPy arrays, which only include **homogeneous data (Elements with the same data type).**
- This makes it more efficient at storing and manipulating the array.
- This difference becomes evident **when the array contains numerous elements, say thousands or millions.** Additionally, you can do **element-wise operations with NumPy arrays,** which is not feasible with Python lists.
- This is the reason why **NumPy arrays** are preferred over **Python lists** when performing mathematical operations on a large amount of data.

- **How to check python and numpy version:**

```
import numpy as np
import platform
print('Python version is: ' + platform.python_version())
print('Numpy version is: ' + np.__version__)
```

- Output is:

  Python version is: 3.6.6

  Numpy version is: 1.15.0