

1. INTRODUCTION & COMPONENTS OF PYTHON.

1.1. Understanding Python

Python is an interactive programming language. Simple syntax of the language makes Python programs easy to read and write. **Python was developed by Guido Van Rossum in 1991 at the National Research Institute for Mathematics and Computer Science in the Netherlands.** Guido Van Rossum named Python by getting inspired from his favourite comedy show **Monty Python's Flying Circus**.

Ever since the language was developed, it is becoming popular day by day amongst the programmers. Various versions of Python have been released till date ranging from version 1.0. Python is used in various application areas such as the web, gaming, scientific and numeric computing, text processing, and network programming.

Python is a computer programming language often used to build websites and software, automate tasks, and conduct data analysis. Python is a general-purpose language, meaning it can be used to create a variety of different programs and isn't specialized for any specific problems.

Python is a high-level, general-purpose, and very popular programming language. Python programming language (latest Python 3) is being used in web development, Machine Learning applications, along with all cutting-edge technology in Software Industry.

Python language is being used by almost all tech-giant companies like – Google, Amazon, Facebook, Instagram, Dropbox, Uber... etc.

The biggest strength of Python is huge collection of standard library which can be used for the following:

- Machine Learning
- GUI Applications (like Kivy, Tkinter, PyQt etc.)
- Web frameworks like Django (used by YouTube, Instagram, Dropbox)
- Image processing (like OpenCV, Pillow)
- Web scraping (like Scrapy, BeautifulSoup, Selenium)
- Test frameworks
- Multimedia
- Scientific computing

- Text processing and many more..

Python is a widely used general-purpose, high level programming language. It was created by Guido van Rossum in 1991 and further developed by the Python Software Foundation. It was designed with an emphasis on code readability, and its syntax allows programmers to express their concepts in fewer lines of code.

Python is a programming language that lets you work quickly and integrate systems more efficiently.

There are two major Python versions: Python 2 and Python 3. Both are quite different.

Reason for increasing popularity:

1. Emphasis on code readability, shorter codes, ease of writing
2. Programmers can express logical concepts in fewer lines of code in comparison to languages such as C++ or Java.
3. Python supports multiple programming paradigms, like object-oriented, imperative and functional programming or procedural.
4. There exist inbuilt functions for almost all of the frequently used concepts.
5. Philosophy is "Simplicity is the best".

LANGUAGE FEATURES

1. Interpreted

- There are no separate compilation and execution steps like C and C++.
- Directly run the program from the source code.
- Internally, Python converts the source code into an intermediate form called bytecodes which is then translated into native language of specific computer to run it.
- No need to worry about linking and loading with libraries, etc.

2. Platform Independent

- Python programs can be developed and executed on multiple operating system platforms.
- Python can be used on Linux, Windows, Macintosh, Solaris and many more.

3. Free and Open Source; Redistributable

4. High-level Language

- In Python, no need to take care about low-level details such as managing the memory used by the program.

5. Simple

- Closer to English language; Easy to Learn
- More emphasis on the solution to the problem rather than the syntax

6. Embeddable

- Python can be used within C/C++ program to give scripting capabilities for the program's users.

7. Robust:

- Exceptional handling features
- Memory management techniques in built

8. Rich Library Support

- The Python Standard Library is very vast.
- Known as the “batteries included” philosophy of Python; It can help do various things involving regular expressions, documentation generation, unit testing, threading, databases, web browsers, CGI, email, XML, HTML, WAV files, cryptography, GUI and many more.
- Besides the standard library, there are various other high-quality libraries such as the Python Imaging Library which is an amazingly simple image manipulation library.

Python vs JAVA

Python	Java
Dynamically Typed <ul style="list-style-type: none"> • No need to declare anything. An assignment statement binds a name to an object, and the object can be of any type. • No type casting is required when using container objects 	Statically Typed <ul style="list-style-type: none"> • All variable names (along with their types) must be explicitly declared. Attempting to assign an object of the wrong type to a variable name triggers a type exception. • Type casting is required when using container objects.
Concise Express much in limited words	Verbose Contains more words
Compact	Less Compact

The classical Hello World program illustrating the relative verbosity of a Java Program and Python Program

Java Code: `public class HelloWorld`

```
{  
    public static void main (String[] args)  
    {  
        System.out.println("Hello, world!");  
    }  
}
```

Python Code: `print("Hello, world!")`

Softwares making use of Python

Python has been successfully embedded in a number of software products as a scripting language.

- GNU Debugger uses Python as a pretty printer to show complex structures such as C++ containers.
- Python has also been used in artificial intelligence
- Python is often used for natural language processing tasks.

Current Applications of Python

- A number of Linux distributions use installers written in Python example in Ubuntu we have the Ubiquity
- Python has seen extensive use in the information security industry, including in exploit development.
- Raspberry Pi- single board computer uses Python as its principal user-programming language.
- Python is now being used Game Development areas also.

Pros:

- Ease of use
- Multi-paradigm Approach

Cons:

- Slow speed of execution compared to C, C++
- Absence from mobile computing and browsers
- For the C, C++ programmers switching to python can be irritating as the language requires proper indentation of code. Certain variable names commonly used like sum are functions in python. So C, C++ programmers have to look out for these.

Industrial Importance:

Most of the companies are now looking for candidates who know about Python Programming. Those having the knowledge of python may have more chances of impressing the interviewing panel. So, I would suggest that beginners should start learning python and excel in it.

Python is a high-level, interpreted, and general-purpose dynamic programming language that focuses on code readability. It has fewer steps when compared to Java and C. It was founded in 1991 by developer Guido Van Rossum. Python ranks among the most popular and fastest-growing languages in the world. Python is a powerful, flexible, and easy-to-use language. In addition, the community is very active there. It is used in many organizations as it supports multiple programming paradigms. It also performs automatic memory management.

Advantages:

1. Presence of third-party modules
2. Extensive support libraries(NumPy for numerical calculations, Pandas for data analytics etc)
3. Open source and community development
4. Versatile, Easy to read, learn and write
5. User-friendly data structures
6. High-level language
7. Dynamically typed language (No need to mention data type based on the value assigned, it takes data type)
8. Object-oriented language
9. Portable and Interactive
10. Ideal for prototypes – provide more functionality with less coding

11. Highly Efficient (Python's clean object-oriented design provides enhanced process control, and the language is equipped with excellent text processing and integration capabilities, as well as its own unit testing framework, which makes it more efficient.)
12. (IoT)Internet of Things Opportunities
13. Interpreted Language
14. Portable across Operating systems

Applications :



1. GUI based desktop applications

2. Graphic design, image processing applications, Games, and Scientific/computational Applications
3. Web frameworks and applications
4. Enterprise and Business applications
5. Operating Systems
6. Education
7. Database Access
8. Language Development
9. Prototyping
10. Software Development

Organizations using Python:

1. Google (Components of Google spider and Search Engine)
2. Yahoo (Maps)
3. YouTube
4. Mozilla
5. Dropbox
6. Microsoft
7. Cisco
8. Spotify
9. Quora

List of all Python versions

The following table lists all the versions of Python. Which contains version and their documentation released date. The most recent version is 3.12.1, which was released on 8 December 2023.

PYTHON DOCUMENTATION BY VERSION
Python 3.12.1, documentation released on 8 December 2023.
Python 3.12.0, documentation released on 2 October 2023.

Python 3.11.7, documentation released on 4 December 2023.
Python 3.11.6, documentation released on 2 October 2023.
Python 3.11.5, documentation released on 24 August 2023.
Python 3.11.4, documentation released on 6 June 2023.
Python 3.11.3, documentation released on 5 April 2023.
Python 3.11.2, documentation released on 8 February 2023.
Python 3.11.1, documentation released on 6 December 2022.
Python 3.11.0, documentation released on 24 October 2022.
Python 3.10.13, documentation released on 24 August 2023.
Python 3.10.12, documentation released on 6 June 2023.
Python 3.10.11, documentation released on 5 April 2023.
Python 3.10.10, documentation released on 8 February 2023.
Python 3.10.9, documentation released on 6 December 2022.
Python 3.10.8, documentation released on 8 October 2022.
Python 3.10.7, documentation released on 6 September 2022.
Python 3.10.6, documentation released on 8 August 2022.
Python 3.10.5, documentation released on 6 June 2022.
Python 3.10.4, documentation released on 24 March 2022.
Python 3.10.3, documentation released on 16 March 2022.
Python 3.10.2, documentation released on 14 January 2022.
Python 3.10.1, documentation released on 6 December 2021.
Python 3.10.0, documentation released on 4 October 2021.
Python 3.9.18, documentation released on 24 August 2023.

Python 3.9.17, documentation released on 6 June 2023.
Python 3.9.16, documentation released on 6 December 2022.
Python 3.9.15, documentation released on 11 October 2022.
Python 3.9.14, documentation released on 6 September 2022.
Python 3.9.13, documentation released on 17 May 2022.
Python 3.9.12, documentation released on 24 March 2022.
Python 3.9.11, documentation released on 16 March 2022.
Python 3.9.10, documentation released on 14 January 2022.
Python 3.9.9, documentation released on 15 November 2021.
Python 3.9.8, documentation released on 05 November 2021.
Python 3.9.7, documentation released on 30 August 2021.
Python 3.9.6, documentation released on 28 June 2021.
Python 3.9.5, documentation released on 3 May 2021.
Python 3.9.4, documentation released on 4 April 2021.
Python 3.9.3, documentation released on 2 April 2021.
Python 3.9.2, documentation released on 19 February 2021.
Python 3.9.1, documentation released on 8 December 2020.
Python 3.9.0, documentation released on 5 October 2020.
Python 3.8.18, documentation released on 24 August 2023.
Python 3.8.17, documentation released on 6 June 2023.
Python 3.8.16, documentation released on 6 December 2022.
Python 3.8.15, documentation released on 11 October 2022.
Python 3.8.14, documentation released on 6 September 2022.

Python 3.8.13, documentation released on 16 March 2022.
Python 3.8.12, documentation released on 30 August 2021.
Python 3.8.11, documentation released on 28 June 2021.
Python 3.8.10, documentation released on 3 May 2021.
Python 3.8.9, documentation released on 2 April 2021.
Python 3.8.8, documentation released on 19 February 2021.
Python 3.8.7, documentation released on 21 December 2020.
Python 3.8.6, documentation released on 23 September 2020.
Python 3.8.5, documentation released on 20 July 2020.
Python 3.8.4, documentation released on 13 July 2020.
Python 3.8.3, documentation released on 13 May 2020.
Python 3.8.2, documentation released on 24 February 2020.
Python 3.8.1, documentation released on 18 December 2019.
Python 3.8.0, documentation released on 14 October 2019.
Python 3.7.17, documentation released on 6 June 2023.
Python 3.7.16, documentation released on 6 December 2022.
Python 3.7.15, documentation released on 11 October 2022.
Python 3.7.14, documentation released on 6 September 2022.
Python 3.7.13, documentation released on 16 March 2022.
Python 3.7.12, documentation released on 4 September 2021.
Python 3.7.11, documentation released on 28 June 2021.
Python 3.7.10, documentation released on 15 February 2021.
Python 3.7.9, documentation released on 17 August 2020.

Python 3.7.8, documentation released on 27 June 2020.
Python 3.7.7, documentation released on 10 March 2020.
Python 3.7.6, documentation released on 18 December 2019.
Python 3.7.5, documentation released on 15 October 2019.
Python 3.7.4, documentation released on 08 July 2019.
Python 3.7.3, documentation released on 25 March 2019.
Python 3.7.2, documentation released on 24 December 2018.
Python 3.7.1, documentation released on 20 October 2018.
Python 3.7.0, documentation released on 27 June 2018.
Python 3.6.15, documentation released on 4 September 2021.
Python 3.6.14, documentation released on 28 June 2021.
Python 3.6.13, documentation released on 15 February 2021.
Python 3.6.12, documentation released on 17 August 2020.
Python 3.6.11, documentation released on 27 June 2020.
Python 3.6.10, documentation released on 18 December 2019.
Python 3.6.9, documentation released on 02 July 2019.
Python 3.6.8, documentation released on 24 December 2018.
Python 3.6.7, documentation released on 20 October 2018.
Python 3.6.6, documentation released on 27 June 2018.
Python 3.6.5, documentation released on 28 March 2018.
Python 3.6.4, documentation released on 19 December 2017.
Python 3.6.3, documentation released on 03 October 2017.
Python 3.6.2, documentation released on 17 July 2017.

Python 3.6.1, documentation released on 21 March 2017.
Python 3.6.0, documentation released on 23 December 2016.
Python 3.5.10, documentation released on 5 September 2020.
Python 3.5.8, documentation released on 1 November 2019.
Python 3.5.7, documentation released on 18 March 2019.
Python 3.5.6, documentation released on 8 August 2018.
Python 3.5.5, documentation released on 4 February 2018.
Python 3.5.4, documentation released on 25 July 2017.
Python 3.5.3, documentation released on 17 January 2017.
Python 3.5.2, documentation released on 27 June 2016.
Python 3.5.1, documentation released on 07 December 2015.
Python 3.5.0, documentation released on 13 September 2015.
Python 3.4.10, documentation released on 18 March 2019.
Python 3.4.9, documentation released on 8 August 2018.
Python 3.4.8, documentation released on 4 February 2018.
Python 3.4.7, documentation released on 25 July 2017.
Python 3.4.6, documentation released on 17 January 2017.
Python 3.4.5, documentation released on 26 June 2016.
Python 3.4.4, documentation released on 06 December 2015.
Python 3.4.3, documentation released on 25 February 2015.
Python 3.4.2, documentation released on 4 October 2014.
Python 3.4.1, documentation released on 18 May 2014.
Python 3.4.0, documentation released on 16 March 2014.

Python 3.3.7, documentation released on 19 September 2017.
Python 3.3.6, documentation released on 12 October 2014.
Python 3.3.5, documentation released on 9 March 2014.
Python 3.3.4, documentation released on 9 February 2014.
Python 3.3.3, documentation released on 17 November 2013.
Python 3.3.2, documentation released on 15 May 2013.
Python 3.3.1, documentation released on 7 April 2013.
Python 3.3.0, documentation released on 29 September 2012.
Python 3.2.6, documentation released on 11 October 2014.
Python 3.2.5, documentation released on 15 May 2013.
Python 3.2.4, documentation released on 7 April 2013.
Python 3.2.3, documentation released on 10 April 2012.
Python 3.2.2, documentation released on 4 September 2011.
Python 3.2.1, documentation released on 10 July 2011.
Python 3.2, documentation released on 20 February 2011.
Python 3.1.5, documentation released on 9 April 2012.
Python 3.1.4, documentation released on 11 June 2011.
Python 3.1.3, documentation released on 27 November 2010.
Python 3.1.2, documentation released on 21 March 2010.
Python 3.1.1, documentation released on 17 August 2009.
Python 3.1, documentation released on 27 June 2009.
Python 3.0.1, documentation released on 13 February 2009.
Python 3.0, documentation released on 3 December 2008.

Python 2.7.18, documentation released on 20 April 2020
Python 2.7.17, documentation released on 19 October 2019
Python 2.7.16, documentation released on 02 March 2019
Python 2.7.15, documentation released on 30 April 2018
Python 2.7.14, documentation released on 16 September 2017
Python 2.7.13, documentation released on 17 December 2016
Python 2.7.12, documentation released on 26 June 2016.
Python 2.7.11, documentation released on 5 December 2015.
Python 2.7.10, documentation released on 23 May 2015.
Python 2.7.9, documentation released on 10 December 2014.
Python 2.7.8, documentation released on 1 July 2014.
Python 2.7.7, documentation released on 31 May 2014.
Python 2.7.6, documentation released on 10 November 2013.
Python 2.7.5, documentation released on 15 May 2013.
Python 2.7.4, documentation released on 6 April 2013.
Python 2.7.3, documentation released on 9 April 2012.
Python 2.7.2, documentation released on 11 June 2011.
Python 2.7.1, documentation released on 27 November 2010.
Python 2.7, documentation released on 4 July 2010.
Python 2.6.9, documentation released on 29 October 2013.
Python 2.6.8, documentation released on 10 April 2012.
Python 2.6.7, documentation released on 3 June 2011.
Python 2.6.6, documentation released on 24 August 2010.

Python 2.6.5, documentation released on 19 March 2010.
Python 2.6.4, documentation released on 25 October 2009.
Python 2.6.3, documentation released on 2 October 2009.
Python 2.6.2, documentation released on 14 April 2009.
Python 2.6.1, documentation released on 4 December 2008.
Python 2.6, documentation released on 1 October 2008.
Python 2.5.4, documentation released on 23 December 2008.
Python 2.5.3, documentation released on 19 December 2008.
Python 2.5.2, documentation released on 21 February 2008.
Python 2.5.1, documentation released on 18 April 2007.
Python 2.5, documentation released on 19 September 2006.
Python 2.4.4, documentation released on 18 October 2006.
Python 2.4.3, documentation released on 29 March 2006.
Python 2.4.2, documentation released on 28 September 2005.
Python 2.4.1, documentation released on 30 March 2005.
Python 2.4, documentation released on 30 November 2004.
Python 2.3.5, documentation released on 8 February 2005.
Python 2.3.4, documentation released on 27 May 2004.
Python 2.3.3, documentation released on 19 December 2003.
Python 2.3.2, documentation released on 3 October 2003.
Python 2.3.1, documentation released on 23 September 2003.
Python 2.3, documentation released on 29 July 2003.
Python 2.2.3, documentation released on 30 May 2003.

Python 2.2.2, documentation released on 14 October 2002.
Python 2.2.1, documentation released on 10 April 2002.
Python 2.2p1, documentation released on 29 March 2002.
Python 2.2, documentation released on 21 December 2001.
Python 2.1.3, documentation released on 8 April 2002.
Python 2.1.2, documentation released on 16 January 2002.
Python 2.1.1, documentation released on 20 July 2001.
Python 2.1, documentation released on 15 April 2001.
Python 2.0.1, documentation released on 22 June 2001.
Python 2.0, documentation released on 16 October 2000.
Python 1.6, documentation released on 5 September 2000.
Python 1.5.2p2, documentation released on 22 March 2000.
Python 1.5.2p1, documentation released on 6 July 1999.
Python 1.5.2, documentation released on 30 April 1999.
Python 1.5.1p1, documentation released on 6 August 1998.
Python 1.5.1, documentation released on 14 April 1998.
Python 1.5, documentation released on 17 February 1998.
Python 1.4, documentation released on 25 October 1996.

And many more versions of Python will be released in the future, each with minor additions or changes to features or functions in the language

1.2. Role of Python in AI and Data science

1. Python is a popular programming language used for Data Science. Here are some of the key reasons why Python is widely used for Data Science:

2. **Easy to Learn:** Python has a simple and intuitive syntax that is easy to learn and read. This makes it a great language for beginners to learn data science.
 3. **Large and Active Community:** Python has a large and active community that supports and contributes to the development of various libraries and tools for data science. This community has created many useful libraries, including Pandas, NumPy, Matplotlib, and SciPy, which are widely used in data science.
 4. **Powerful Libraries:** Python has several powerful libraries that make data analysis and visualization easy. Pandas is a library for data manipulation and analysis, NumPy is a library for numerical computation, and Matplotlib is a library for data visualization.
 5. **Machine Learning Libraries:** Python has several popular libraries for machine learning, including Scikit-learn, TensorFlow, and Keras, which make it easy to build and train machine learning models.
 6. **Integration with other languages:** Python can easily integrate with other programming languages like R, Java, and C++, making it easy to use Python for data science in a variety of settings.
- Python is a popular language for data science because it is easy to learn, has a large and active community, offers powerful libraries for data analysis and visualization, and has excellent machine learning libraries.
 - Python is an open-source, interpreted, high-level language and provides a great approach to data science, machine learning, and research purposes. It is one of the best languages for data science to use for various applications & projects. When it comes to dealing with mathematical, statistical, and scientific functions, Python has great utility.

Following are some useful features of Python that make it suitable for AI and data science :

- It uses elegant syntax, hence the programming is easier to read.
- It is an easy-to-access language.
- The large standard library and community support for mathematics and Data science purposes.
- Python's interactive mode makes it easy to evaluate codes.

- By adding new modules that are implemented in other compiled languages like C++ or C, it's quite easy to expand the code in Python.
- Python is a powerful language that can be integrated into programs to provide a customizable interface.
- It allows developers to run the code on Windows, Mac OS X, UNIX, and Linux.
- It is open-source interpreted, high-level language. Use, download, and program addition of Pythons are all free of charge.

Data Science With Python

Let's understand some popular Python libraries which make Data science easy with Python:

Data Analysis Tools in Python:

- **NumPy:** Numpy is a Python library that provides a mathematical function to handle large-dimension arrays. It provides various functions for Array, linear algebra, and statistical analysis. NumPy stands for Numerical Python. It offers many practical functions for n-array and matrix operations in Python. Large multidimensional arrays and matrices can have their mathematical operations vectorized, which improves efficiency and accelerates execution.
- **Pandas:** One of the most used Python tools for data manipulation and analysis is Pandas. Pandas provide useful functions to manipulate large amounts of structured data. Pandas provide the easiest method to perform analysis. It provides large data structures and manipulates numerical tables and time series data. For handling data, Pandas are the ideal instrument. It is intended for fast and simple data aggregation and manipulation. Pandas have two types of data structures: Series – It Handle and stores data in one-dimensional data. DataFrame – It Handle and stores Two-dimensional data.
- **Scipy:** Scipy is a well-liked Python library for scientific computing and data analysis. Scipy stands for Scientific Python. It provides great functionality for scientific mathematics and computing programming. SciPy contains sub-modules for optimization, linear algebra, integration, interpolation, special functions, and other tasks common in science and engineering.

Data Visualization Tools in Python:

- **Matplotlib:** Matplotlib is a useful Python library for Data Visualization. Descriptive analysis and visualizing data are very important for any organization. Matplotlib provides various methods to Visualize data in a more effective way. It allows you to quickly make line graphs, pie charts, histograms, and other professional-grade figures. Using Matplotlib, one can customize every aspect of a figure. Matplotlib has interactive features like zooming and panning and saving the Graph in graphics format.
- **Seaborn:** Seaborn is another library for Data visualization and statistical plotting in Python. It can be used with Matplotlib for statistical plotting and to make the graph beautiful and attractive.
- **Plotly:** Plotly is another library for Data visualization and understanding the data simply. It plots a more visually attractive graph. it is used by various institutions for scaling and deployment purposes.

Machine Learning Tools in Python

- **Scikit – learn:** Sklearn is a Python library for machine learning. Sklearn offers various algorithms and functions that are used in machine learning. Sklearn is built on Matplotlib, SciPy, and NumPy. It offers simple and straightforward tools for data analysis and predictions. It offers users a collection of standard machine-learning algorithms via a reliable user interface. Popular algorithms can be rapidly applied to datasets to solve real problems.
- **Statsmodels:** Statsmodel is great python tools. It has a number of functions which is helpful in statistical test and building statistical model in python.
- **Tensorflow:** Tensorflow is an open-source platform for Deep learning tasks. It is developed by Google. It uses Keras api's. It is suitable for computer vision and Natural language processing tasks. It was created using Python, CUDA, and C++. It is compatible with TPUs, GPUs, and CPUs.
- **Keras:** Keras is a high-level open-source API for neural network tasks. It is developed by François Chollet. it can be used with Tensorflow. it can also run on CPUs and GPUs.

- **Pytorch:** Pytorch is a popular open-source Deep Learning Platform. it is based on the torch library. It has a large number of data structures for multi-dimensional tensors and various mathematical operations. It is developed by Facebook Meta AI. It is easy and very much suitable for computer vision and Natural language processing tasks.

Advantages of Python for Data Science:

1. Easy to learn and read syntax
2. Large and active community with extensive documentation and support
3. Powerful libraries for data analysis and visualization, such as Pandas and Matplotlib
4. Excellent libraries for machine learning, such as Scikit-learn and TensorFlow
5. Integration with other languages and tools
6. Open-source and free to use

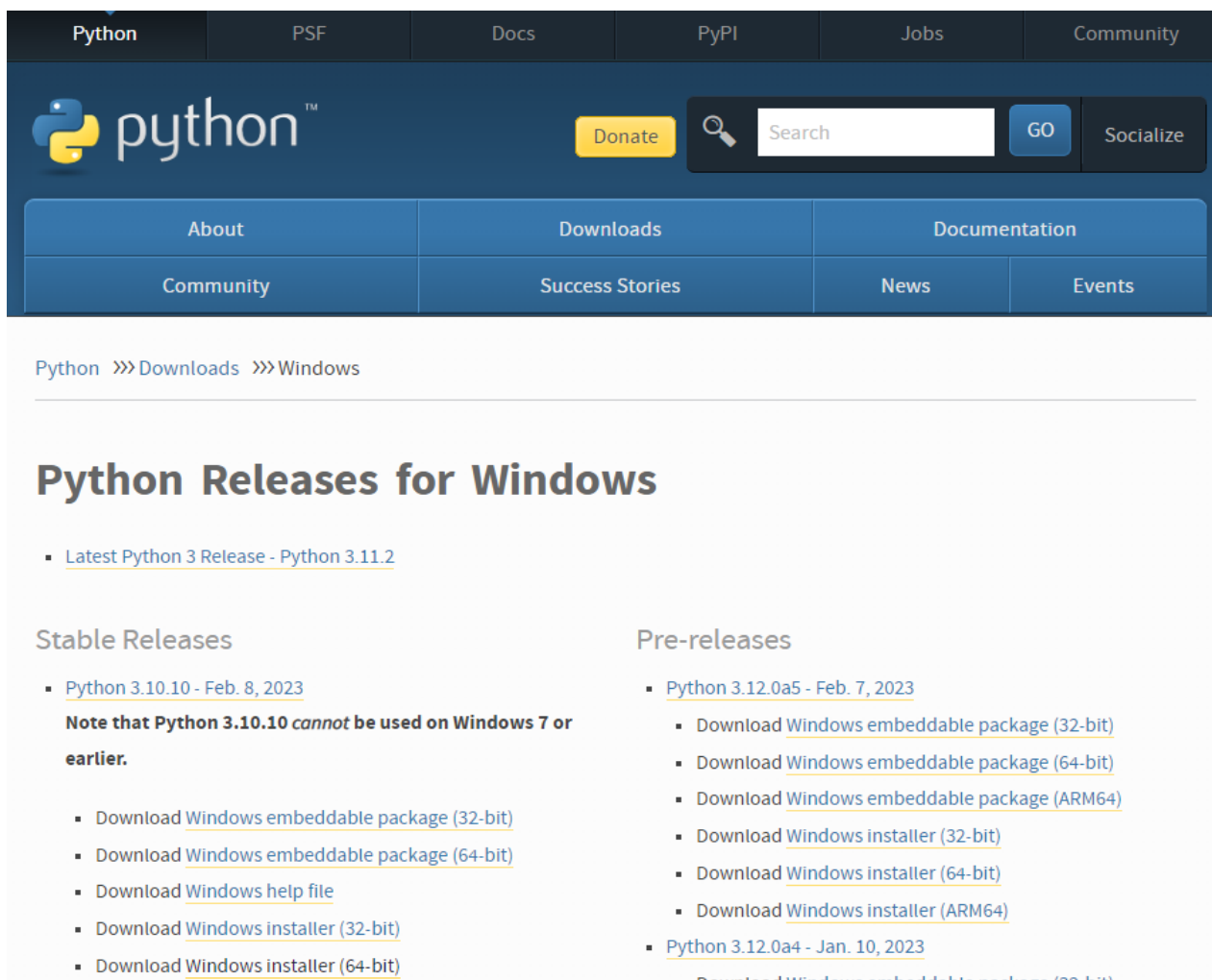
Disadvantages of Python for Data Science:

1. Slow execution time compared to low-level programming languages like C++
2. Limited support for multi-threading
3. Limited memory management control
4. Weak in handling big data compared to distributed computing frameworks like Hadoop
5. Steep learning curve for more advanced topics such as distributed computing and parallelism
6. In summary, while Python has several advantages for Data Science, including its ease of use, powerful libraries, and integration with other languages, it also has
7. some disadvantages, such as slow execution time and limited support for multi-threading. However, these drawbacks can be overcome with the use of additional tools and frameworks, making Python an excellent choice for many data science tasks.

1.3. Installation and Working with Python

Step 1 — Downloading the Python Installer

1. Go to the official Python download page for Windows.
2. Find a stable Python 3 release. This tutorial was tested with Python version 3.10.10.
3. Click the appropriate link for your system to download the executable file: Windows installer (64-bit) or Windows installer (32-bit).



Step 2 — Running the Executable Installer

1. After the installer is downloaded, double-click the .exe file, for example python-3.10.10-amd64.exe, to run the Python installer.
2. Select the Install launcher for all users checkbox, which enables all users of the computer to access the Python launcher application.

3. Select the Add python.exe to PATH checkbox, which enables users to launch Python from the command line.



4. If you're just getting started with Python and you want to install it with default features as described in the dialog, then click Install Now and go to Step 4 - Verify the Python Installation. To install other optional and advanced features, click Customize installation and continue.
5. The Optional Features include common tools and resources for Python and you can install all of them, even if you don't plan to use them.



Optional Features

☒ Documentation

Installs the Python documentation file.

☒ pip

Installs pip, which can download and install other Python packages.

☒ tcl/tk and IDLE

Installs tkinter and the IDLE development environment.

☒ Python test suite

Installs the standard library test suite.

☒ py launcher ☒ for all users (requires admin privileges)

Installs the global 'py' launcher to make it easier to start Python.

Back

Next

Cancel

Select some or all of the following options:

- Documentation: recommended
- pip: recommended if you want to install other Python packages, such as NumPy or pandas
- tcl/tk and IDLE: recommended if you plan to use IDLE or follow tutorials that use it
- Python test suite: recommended for testing and learning
- py launcher and for all users: recommended to enable users to launch Python from the command line

6. Click Next.

7. The Advanced Options dialog displays.



Advanced Options

- ☐ Install Python 3.10 for all users
- ☒ Associate files with Python (requires the 'py' launcher)
- ☒ Create shortcuts for installed applications
- ☒ Add Python to environment variables
- ☐ Precompile standard library
- ☐ Download debugging symbols
- ☐ Download debug binaries (requires VS 2017 or later)

Customize install location

C:\Users\4andi\AppData\Local\Programs\Python\Python310

Browse

Back

Install

Cancel

Select the options that suit your requirements:

- Install for all users: recommended if you're not the only user on this computer
- Associate files with Python: recommended, because this option associates all the Python file types with the launcher or editor
- Create shortcuts for installed applications: recommended to enable shortcuts for Python applications
- Add Python to environment variables: recommended to enable launching Python
- Precompile standard library: not required, it might down the installation
- Download debugging symbols and Download debug binaries: recommended only if you plan to create C or C++ extensions

Make note of the Python installation directory in case you need to reference it later.

8. Click Install to start the installation.

9. After the installation is complete, a Setup was successful message displays.



Setup was successful

New to Python? Start with the [online tutorial](#) and [documentation](#). At your terminal, type "py" to launch Python, or search for Python in your Start menu.

See [what's new](#) in this release, or find more info about [using Python on Windows](#).



Disable path length limit

Changes your machine configuration to allow programs, including Python, to bypass the 260 character "MAX_PATH" limitation.

Close

Step 3 — Adding Python to the Environment Variables (optional)

Skip this step if you selected **Add Python to environment variables** during installation.

If you want to access Python through the command line but you didn't add Python to your environment variables during installation, then you can still do it manually.

Before you start, locate the Python installation directory on your system. The following directories are examples of the default directory paths:

- C:\Program Files\Python310: if you selected **Install for all users** during installation, then the directory will be system wide
- C:\Users\Sammy\AppData\Local\Programs\Python\Python310: if you didn't select **Install for all users** during installation, then the directory will be in the Windows user path

Note that the folder name will be different if you installed a different version, but will still start with Python.

1. Go to **Start** and enter advanced system settings in the search bar.
2. Click **View advanced system settings**.

3. In the **System Properties** dialog, click the **Advanced** tab and then click **Environment Variables**.
4. Depending on your installation:
 - If you selected **Install for all users** during installation, select **Path** from the list of **System Variables** and click **Edit**.
 - If you didn't select **Install for all users** during installation, select **Path** from the list of **User Variables** and click **Edit**.
5. Click **New** and enter the Python directory path, then click **OK** until all the dialogs are closed.

Step 4 — Verify the Python Installation

You can verify whether the Python installation is successful either through the command line or through the Integrated Development Environment (IDLE) application, if you chose to install it.

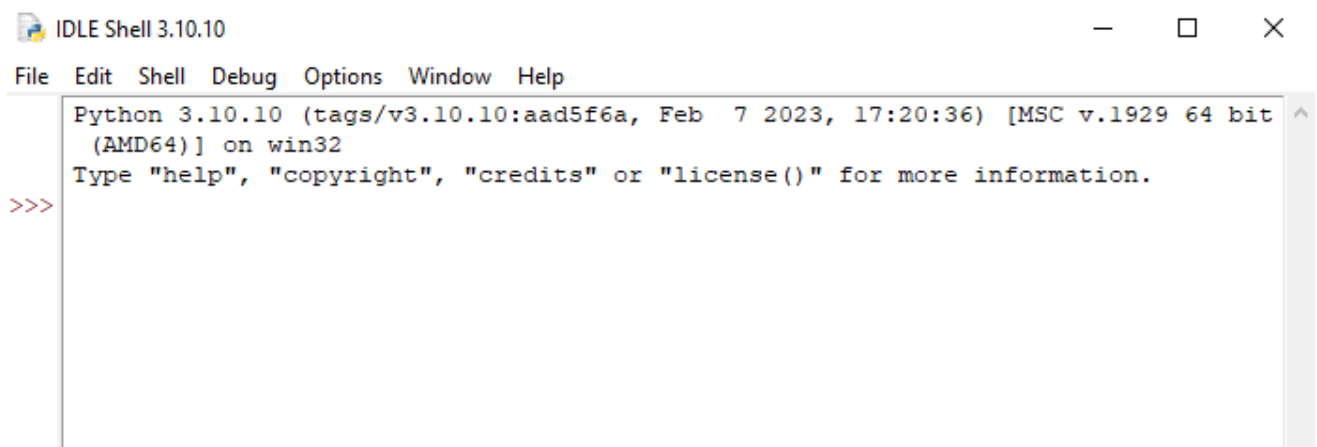
Go to **Start** and enter cmd in the search bar. Click **Command Prompt**.

Enter the following command in the command prompt:

```
python -version
```

Output

```
Python 3.10.10
```

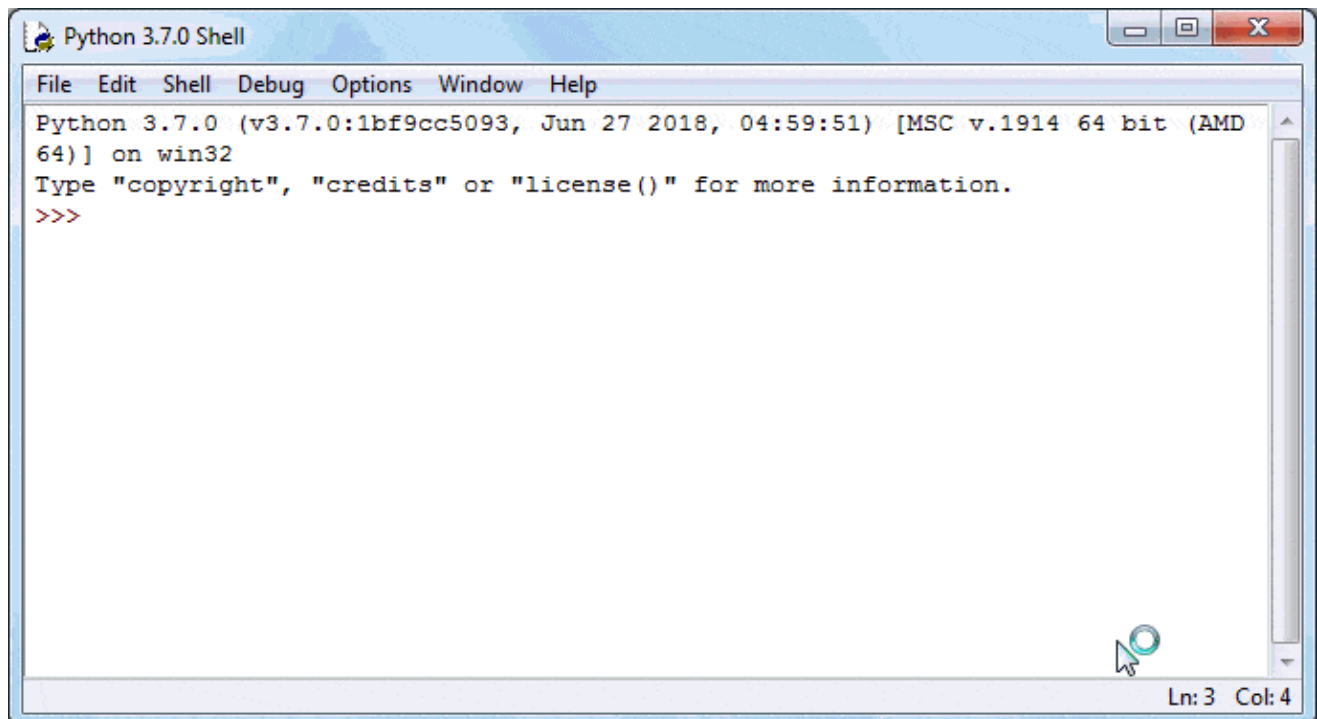


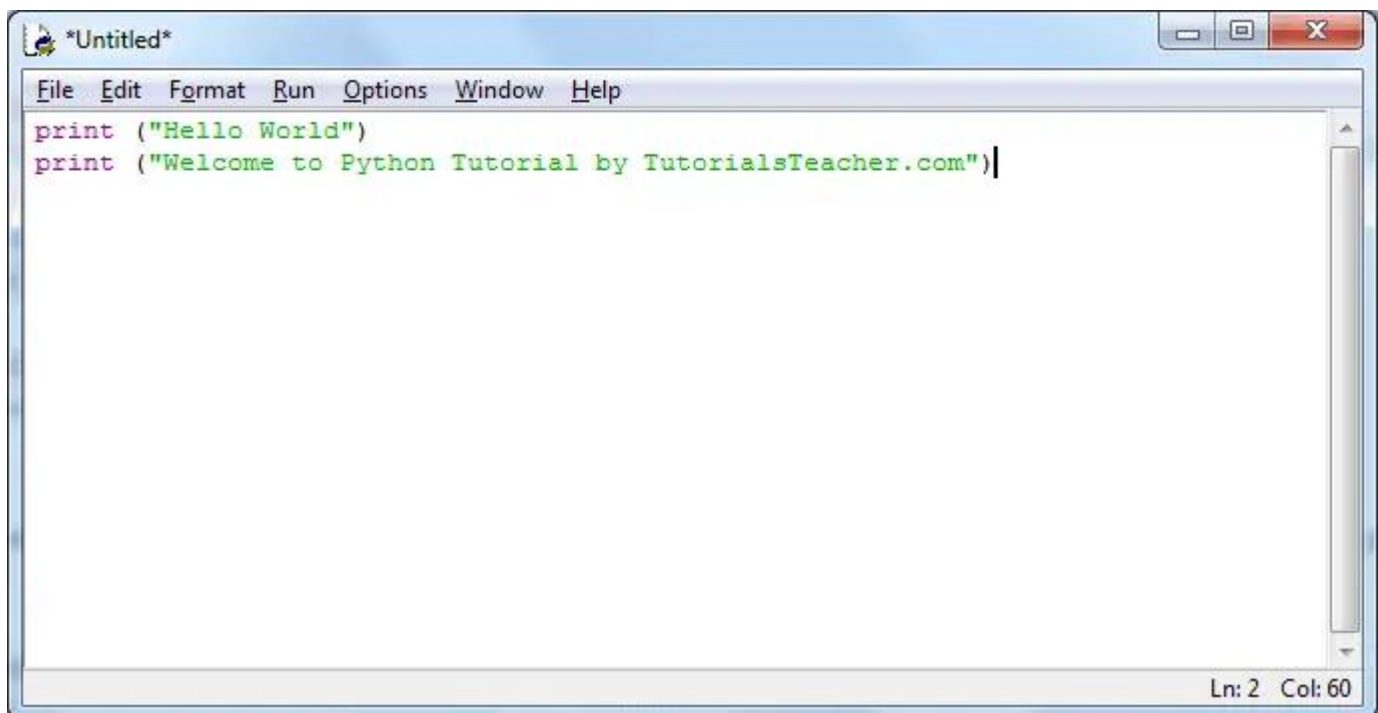
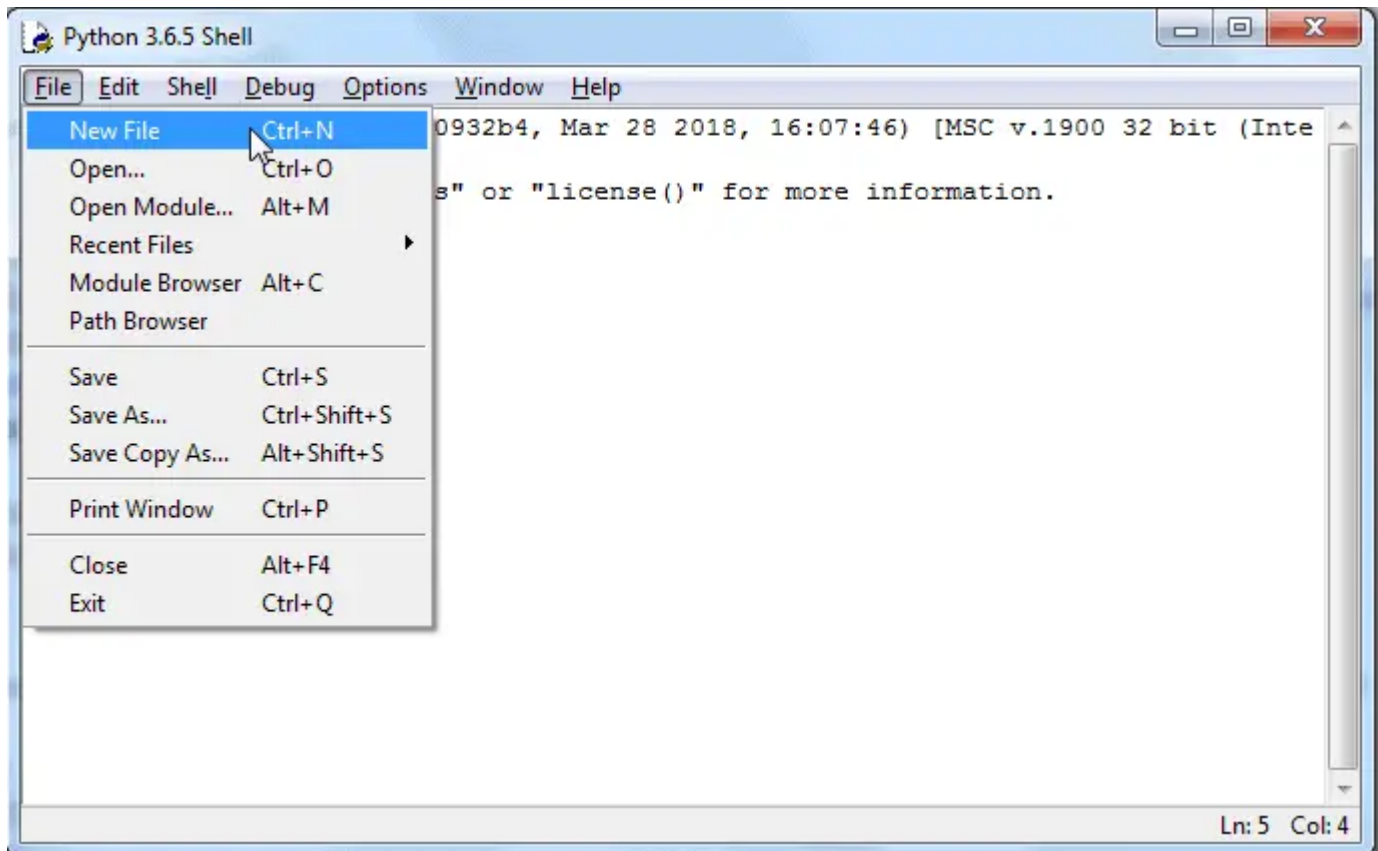
1.4. The default graphical development environment for Python – IDLE

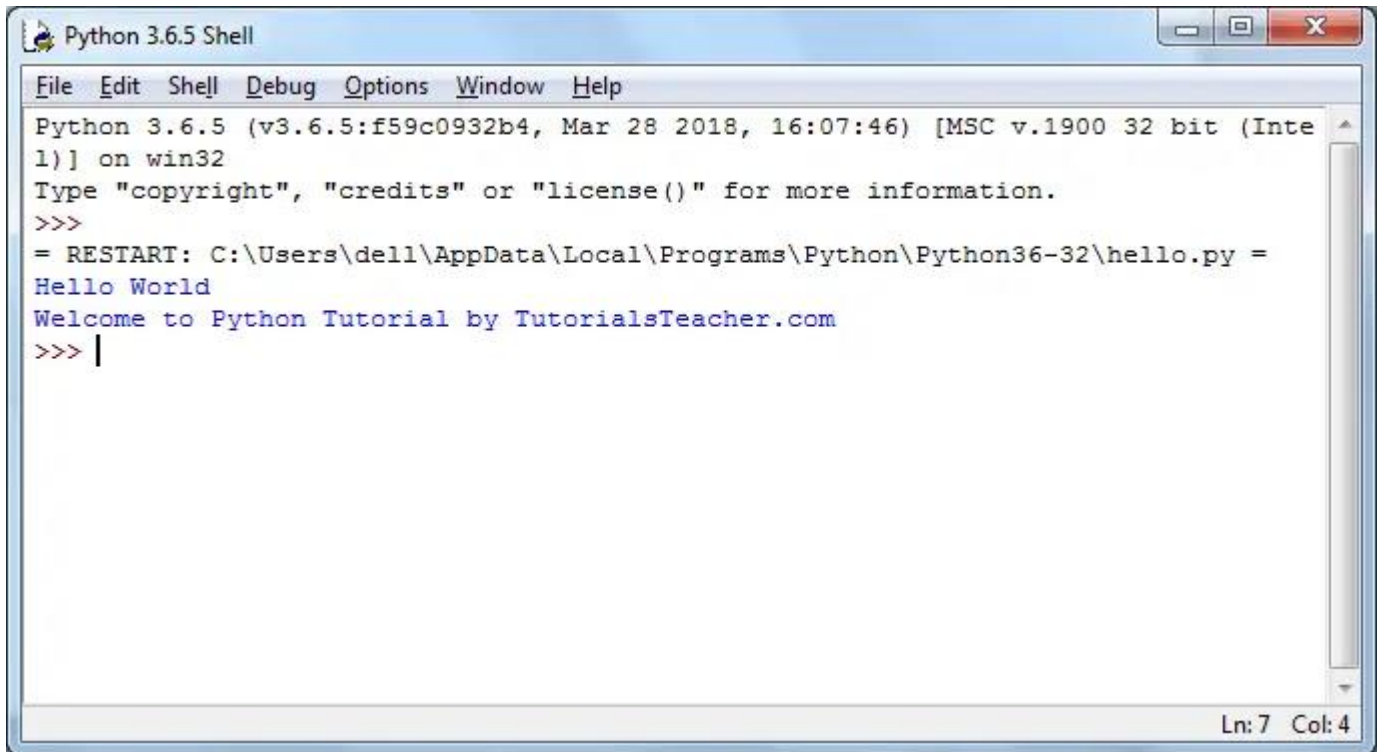
IDLE is Python's Integrated Development and Learning Environment.

IDLE has the following features:

- coded in 100% pure Python, using the tkinter GUI toolkit
- cross-platform: works mostly the same on Windows, Unix, and macOS
- Python shell window (interactive interpreter) with colorizing of code input, output, and error messages
- multi-window text editor with multiple undo, Python colorizing, smart indent, call tips, auto completion, and other features
- search within any window, replace within editor windows, and search through multiple files (grep)
- debugger with persistent breakpoints, stepping, and viewing of global and local namespaces
- configuration, browsers, and other dialogs





A screenshot of a Python 3.6.5 Shell window. The window has a menu bar with 'File', 'Edit', 'Shell', 'Debug', 'Options', 'Window', and 'Help'. The main text area shows the following content: 'Python 3.6.5 (v3.6.5:f59c0932b4, Mar 28 2018, 16:07:46) [MSC v.1900 32 bit (Intel)] on win32', 'Type "copyright", "credits" or "license()" for more information.', '>>>', '= RESTART: C:\Users\dell\AppData\Local\Programs\Python\Python36-32\hello.py =', 'Hello World', 'Welcome to Python Tutorial by TutorialsTeacher.com', and '>>> |'. The status bar at the bottom right indicates 'Ln: 7 Col: 4'.

for more information visit official website: <https://docs.python.org/3/library/idle.html>

1.5. Types and Operation

In Python programming, Operators in general are used to perform operations on values and variables. These are standard symbols used for the purpose of logical and arithmetic operations. In this article, we will look into different types of Python operators.

- OPERATORS: These are the special symbols. Eg- + , * , / , etc.
- OPERAND: It is the value on which the operator is applied.

Types of Operators in Python

- Arithmetic Operators
- Comparison Operators
- Logical Operators
- Bitwise Operators
- Assignment Operators
- Identity Operators and Membership Operators

1. Arithmetic Operators in Python

Python Arithmetic operators are used to perform basic mathematical operations like addition, subtraction, multiplication, and division.

In Python 3.x the result of division is a floating-point while in Python 2.x division of 2 integers was an integer. To obtain an integer result in Python 3.x floored (// integer) is used.

Operator	Description	Syntax
+	Addition: adds two operands	$x + y$
-	Subtraction: subtracts two operands	$x - y$
*	Multiplication: multiplies two operands	$x * y$
/	Division (float): divides the first operand by the second	x / y
//	Division (floor): divides the first operand by the second	$x // y$
%	Modulus: returns the remainder when the first operand is divided by the second	$x \% y$
**	Power: Returns first raised to power second	$x ** y$

Precedence of Arithmetic Operators in Python

The precedence of Arithmetic Operators in python is as follows:

P – Parentheses

E – Exponentiation

M – Multiplication (Multiplication and division have the same precedence)

D – Division

A – Addition (Addition and subtraction have the same precedence)

S – Subtraction

Example:

```
# Examples of Arithmetic Operator
```

```
a = 9
```

```
b = 4
```

```
# Addition of numbers
```

```
add = a + b
```

```
# Subtraction of numbers
```

```
sub = a - b
```

```

# Multiplication of number
mul = a * b

# Modulo of both number
mod = a % b

# Power
p = a ** b

# print results
print(add)
print(sub)
print(mul)
print(mod)
print(p)

```

Output:

```

13
5
36
1
6561

```

2. Comparison Operators in Python

In Python Comparison of Relational operators compares the values. It either returns True or False according to the condition.

Operator	Description	Syntax
>	Greater than: True if the left operand is greater than the right	$x > y$
<	Less than: True if the left operand is less than the right	$x < y$
==	Equal to: True if both operands are equal	$x == y$
!=	Not equal to – True if operands are not equal	$x != y$
>=	Greater than or equal to True if the left operand is greater than or equal to the right	$x \geq y$
<=	Less than or equal to True if the left operand is less than or equal to the right	$x \leq y$

= is an assignment operator and == comparison operator.

Precedence of Comparison Operators in Python

In python, the comparison operators have lower precedence than the arithmetic operators. All the operators within comparison operators have same precedence order.

Example:

```
# Examples of Relational Operators
```

```
a = 13
```

```
b = 33
```

```
# a > b is False
```

```
print(a > b)
```

```
# a < b is True
```

```
print(a < b)
```

```
# a == b is False
```

```
print(a == b)
```

```
# a != b is True
```

```
print(a != b)
```

```
# a >= b is False
```

```
print(a >= b)
```

```
# a <= b is True
```

```
print(a <= b)
```

Output

```
False
```

```
True
```

```
False
```

```
True
```

```
False
```

```
True
```

3. Logical Operators in Python

Python Logical operators perform Logical AND, Logical OR, and Logical NOT operations. It is used to combine conditional statements.

Operator	Description	Syntax
and	Logical AND: True if both the operands are true	x and y
or	Logical OR: True if either of the operands is true	x or y
not	Logical NOT: True if the operand is false	not x

Precedence of Logical Operators in Python

The precedence of Logical Operators in python is as follows:

1. Logical not
2. logical and
3. logical or

Example:

Examples of Logical Operator

```
a = True
b = False
```

```
# Print a and b is False
print(a and b)
```

```
# Print a or b is True
print(a or b)
```

```
# Print not a is False
print(not a)
```

Output

```
False
True
False
```

4. Bitwise Operators in Python

Python Bitwise operators act on bits and perform bit-by-bit operations. These are used to operate on binary numbers.

Operator	Description	Syntax
&	Bitwise AND	x & y
	Bitwise OR	x y
~	Bitwise NOT	~x
^	Bitwise XOR	x ^ y
>>	Bitwise right shift	x>>
<<	Bitwise left shift	x<<

Precedence of Bitwise Operators in Python

The precedence of Bitwise Operators in python is as follows:

1. Bitwise NOT
2. Bitwise Shift
3. Bitwise AND
4. Bitwise XOR
5. Bitwise OR

Example:

```
# Examples of Bitwise operators
```

```
a = 10
```

```
b = 4
```

```
# Print bitwise AND operation
```

```
print(a & b)
```

```
# Print bitwise OR operation
```

```
print(a | b)
```

```
# Print bitwise NOT operation
```

```
print(~a)
```

```
# print bitwise XOR operation
```

```
print(a ^ b)
```

```
# print bitwise right shift operation
```

```
print(a >> 2)
```

```
# print bitwise left shift operation
```

```
print(a << 2)
```

Output

0

14

-11

14

2

40

5. Assignment Operators in Python:

Python Assignment operators are used to assign values to the variables.

Operator	Description	Syntax
=	Assign the value of the right side of the expression to the left side operand	x = y + z
+=	Add AND: Add right-side operand with left-side operand and then assign to left operand	a+=b a=a+b
-=	Subtract AND: Subtract right operand from left operand and then assign to left operand	a-=b a=a-b
=	Multiply AND: Multiply right operand with left operand and then assign to left operand	a=b a=a*b
/=	Divide AND: Divide left operand with right operand and then assign to left operand	a/=b a=a/b
%=	Modulus AND: Takes modulus using left and right operands and assign the result to left operand	a%=b a=a%b
//=	Divide(floor) AND: Divide left operand with right operand and then assign the value(floor) to left operand	a//=b a=a//b
=	Exponent AND: Calculate exponent(raise power) value using operands and assign value to left operand	a=b a=a**b
&=	Performs Bitwise AND on operands and assign value to left operand	a&=b a=a&b
=	Performs Bitwise OR on operands and assign value to left operand	a =b a=a b
^=	Performs Bitwise xOR on operands and assign value to left operand	a^=b a=a^b
>>=	Performs Bitwise right shift on operands and assign value to left operand	a>>=b a=a>>b
<<=	Performs Bitwise left shift on operands and assign value to left operand	a<<=b a=a<<b

6. Assignment Operators in Python

Examples of Assignment Operators

```
a = 10
```

Assign value

```
b = a
```

```
print(b)
```

Add and assign value

```
b += a
```

```
print(b)
```

Subtract and assign value

```
b -= a
```

```
print(b)
```

multiply and assign

```
b *= a
```

```
print(b)
```

bitwise lishift operator

```
b <<= a
```

```
print(b)
```

Output

```
10
```

```
20
```

```
10
```

```
100
```

```
102400
```

7. Identity Operators in Python

In Python, `is` and `is not` are the identity operators both are used to check if two values are located on the same part of the memory. Two variables that are equal do not imply that they are identical.

is True if the operands are identical

is not True if the operands are not identical

Example:

```
a = 10
b = 20
c = a

print(a is not b)
print(a is c)
```

Output

```
True
True
```

8. Membership Operators in Python

In Python, `in` and `not in` are the membership operators that are used to test whether a value or variable is in a sequence.

in True if value is found in the sequence
not in True if value is not found in the sequence

Python program to illustrate

not 'in' operator

x = 24

y = 20

list = [10, 20, 30, 40, 50]

```
if (x not in list):
    print("x is NOT present in given list")
else:
    print("x is present in given list")
```

```
if (y in list):
    print("y is present in given list")
else:
    print("y is NOT present in given list")
```

Output

```
x is NOT present in given list
y is present in given list
```

9. Ternary Operator in Python

In Python, Ternary operators also known as conditional expressions are operators that evaluate something based on a condition being true or false. It was added to Python in version 2.5.

It simply allows testing a condition in a single line replacing the multiline if-else making the code compact.

Syntax : [on_true] if [expression] else [on_false]

Example:

```
# Program to demonstrate conditional operator
```

```
a, b = 10, 20
```

```
# Copy value of a in min if a < b else copy b
```

```
min = a if a < b else b
```

```
print(min)
```

Output: 10

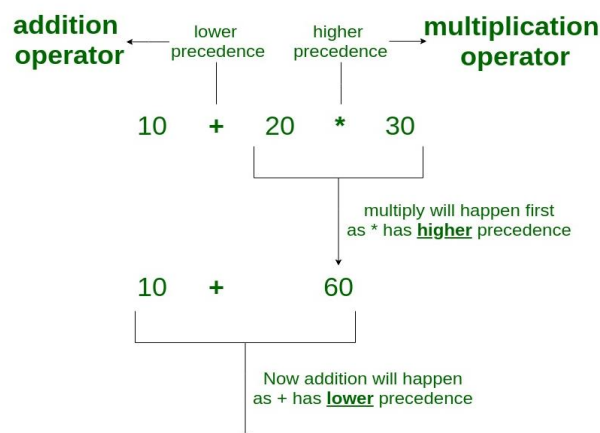
Precedence and Associativity of Operators in Python

When dealing with operators in Python we have to know about the concept of Python operator precedence and associativity as these determine the priorities of the operator otherwise, we'll see unexpected outputs.

Operator Precedence: This is used in an expression with more than one operator with different precedence to determine which operation to perform first.

Example: $10 + 20 * 30$

Operator Precedence



10 + 20 * 30 is calculated as **10 + (20 * 30)**
and not as **(10 + 20) * 30**

```
# Left-right associativity
# 100 / 10 * 10 is calculated as
# (100 / 10) * 10 and not
# as 100 / (10 * 10)
print(100 / 10 * 10)
```

```
# Left-right associativity
# 5 - 2 + 3 is calculated as
# (5 - 2) + 3 and not
# as 5 - (2 + 3)
print(5 - 2 + 3)
```

```
# left-right associativity
print(5 - (2 + 3))
```

```
# right-left associativity
# 2 ** 3 ** 2 is calculated as
# 2 ** (3 ** 2) and not
# as (2 ** 3) ** 2
print(2 ** 3 ** 2)
```

Output:

```
100
6
0
512
```

Operator	Description	Associativity
()	Parentheses	left-to-right
**	Exponent	right-to-left
* / %	Multiplication/division/modulus	left-to-right
+ -	Addition/subtraction	left-to-right
<< >>	Bitwise shift left, Bitwise shift right	left-to-right

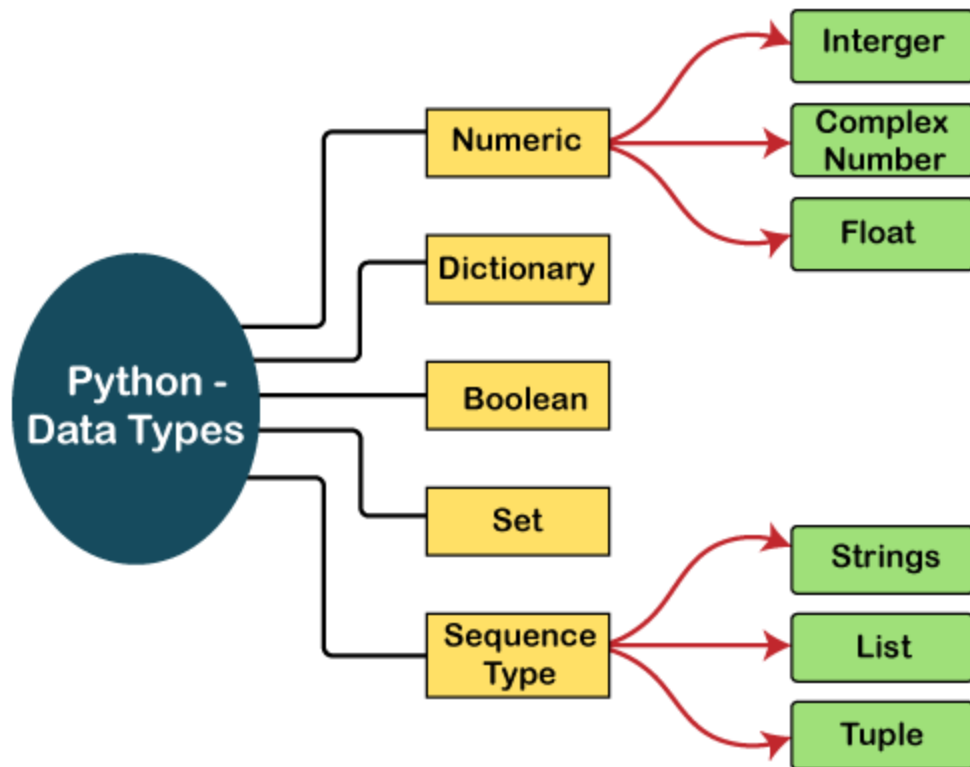
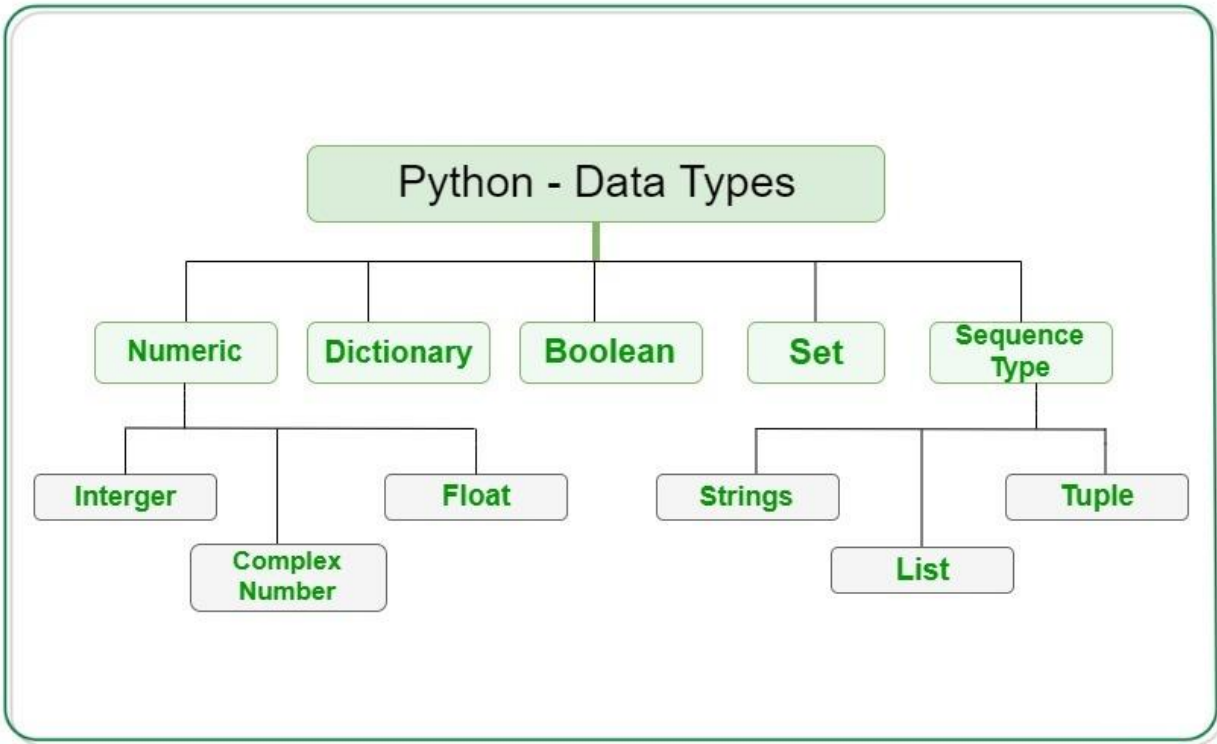
< <= > >=	Relational less than/less than or equal to Relational greater than/greater than or equal to	left-to-right
== !=	Relational is equal to/is not equal to	left-to-right
is, is not in, not in	Identity Membership operators	left-to-right
&	Bitwise AND	left-to-right
^	Bitwise exclusive OR	left-to-right
	Bitwise inclusive OR	left-to-right
not	Logical NOT	right-to-left
and	Logical AND	left-to-right
or	Logical OR	left-to-right
= += -= *= /= %= &= ^= = <<= >>=	Assignment Addition/subtraction assignment Multiplication/division assignment Modulus/bitwise AND assignment Bitwise exclusive/inclusive OR assignment Bitwise shift left/right assignment	right-to-left

1.6. Python Object Types-Number, Strings, Lists, Dictionaries, Tuples, Files, User Defined Classes

Data types are the classification or categorization of data items. It represents the kind of value that tells what operations can be performed on a particular data. Since everything is an object in Python programming, data types are actually classes and variables are instances (object) of these classes. The following are the standard or built-in data types in Python:

- Numeric
- Sequence Type
- Boolean
- Set

- Dictionary
- Binary Types (memoryview, bytearray, bytes)



What is Python type() Function?

To define the values of various data types and check their data types we use the type() function. Consider the following examples.

```
# DataType Output: str
```

```
x = "Hello World"
```

```
# DataType Output: int
```

```
x = 50
```

```
# DataType Output: float
```

```
x = 60.5
```

```
# DataType Output: complex
```

```
x = 3j
```

```
# DataType Output: list
```

```
x = ["ATES", "TC", "AKOLE"]
```

```
# DataType Output: tuple
```

```
x = ("ATES", "TC", "AKOLE")
```

```
# DataType Output: range
```

```
x = range(10)
```

```
# DataType Output: dict
```

```
x = {"name": "Suraj", "age": 24}
```

```
# DataType Output: set
```

```
x = {"ATES", "TC", "AKOLE"}
```

```
# DataType Output: frozenset
```

```
x = frozenset({"ATES", "TC", "AKOLE"})
```

```
# DataType Output: bool
```

```
x = True
```

```
# DataType Output: bytes
```

```
x = b"ATESTC"
```

```
# DataType Output: bytearray
```

```
x = bytearray(4)
```

```
# DataType Output: memoryview
```

```
x = memoryview(bytes(6))
```

```
# DataType Output: NoneType
```

```
x = None
```

Numeric Data Type in Python:

The numeric data type in Python represents the data that has a numeric value. A numeric value can be an integer, a floating number, or even a complex number. These values are defined as Python int, Python float, and Python complex classes in Python.

- **Integers** – This value is represented by int class. It contains positive or negative whole numbers (without fractions or decimals). In Python, there is no limit to how long an integer value can be.
- **Float** – This value is represented by the float class. It is a real number with a floating-point representation. It is specified by a decimal point. Optionally, the character e or E followed by a positive or negative integer may be appended to specify scientific notation.
- **Complex Numbers** – Complex number is represented by a complex class. It is specified as (real part) + (imaginary part)j. For example – 2+3j

Note – type() function is used to determine the type of data type.

```
# Python program to
# demonstrate numeric value

a = 5
print("Type of a: ", type(a))

b = 5.0
print("\nType of b: ", type(b))

c = 2 + 4j
print("\nType of c: ", type(c))
```

Output:

```
Type of a: <class 'int'>
Type of b: <class 'float'>
Type of c: <class 'complex'>
```

Sequence Data Type in Python:

The sequence Data Type in Python is the ordered collection of similar or different data types. Sequences allow storing of multiple values in an organized and efficient fashion. There are several sequence types in Python –

- Python String
- Python List
- Python Tuple

STRING DATA TYPE:

Strings in Python are arrays of bytes representing Unicode characters. A string is a collection of one or more characters put in a single quote, double-quote, or triple-quote. In python there is no character data type, a character is a string of length one. It is represented by str class.

1. Creating String:

Strings in Python can be created using single quotes or double quotes or even triple quotes.

Example:

```
# Python Program for
# Creation of String

# Creating a String
# with single Quotes
String1 = 'Welcome to the ATESTC'
print("String with the use of Single Quotes: ")
print(String1)

# Creating a String
# with double Quotes
String1 = "I'm a Arun"
print("\nString with the use of Double Quotes: ")
print(String1)
print(type(String1))

# Creating a String
# with triple Quotes
String1 = '''I'm a Arun and I live in a world of Book'''
print("\nString with the use of Triple Quotes: ")
print(String1)
print(type(String1))
```

```

# Creating String with triple
# Quotes allows multiple lines
String1 = '''Books
    For
    Life'''
print("\nCreating a multiline String: ")
print(String1)

```

Output:

String with the use of Single Quotes:

Welcome to the Atestc

String with the use of Double Quotes:

I'm a Arun

<class 'str'>

String with the use of Triple Quotes:

'I'm a Arun and I live in a world of Book'

<class 'str'>

Creating a multiline String:

Books

For

Life

2. Accessing elements of String:

In Python, individual characters of a String can be accessed by using the method of Indexing. Indexing allows negative address references to access characters from the back of the String, e.g. -1 refers to the last character, -2 refers to the second last character, and so on.

While accessing an index out of the range will cause an IndexError. Only Integers are allowed to be passed as an index, float or other types that will cause a TypeError.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
W	E	L	C	O	M	E	T	O	A	T	E	S	T	C
-15	-14	-13	-12	-11	-10	-9	-8	-7	-6	-5	-4	-3	-2	-1

```
# Python Program to Access  
# characters of String
```

```
String1 = "Welcome To ATESTC"  
print("Initial String: ")  
print(String1)
```

```
# Printing First character  
print("\\nFirst character of String is: ")  
print(String1[0])
```

```
# Printing Last character  
print("\\nLast character of String is: ")  
print(String1[-1])
```

Output:

```
Initial String:  
Welcome To ATESTC  
First character of String is:  
W  
Last character of String is:  
C
```

3. Reversing a Python String

With Accessing Characters from a string, we can also reverse them. We can Reverse a string by writing [::-1] and the string will be reversed.

```
#Program to reverse a string  
str = "Welcome to Python"  
print(str[::-1])
```

Output:

```
nohtyP ot mocleW
```

We can also reverse a string by using built-in join and reversed function.

```
# Program to reverse a string
```

```
str = " Welcome to Python "  
# Reverse the string using reversed and join function  
str = "".join(reversed(str))  
print(str)
```

Output: nohtyP ot mocleW

4. String Slicing:

Python slicing is about obtaining a sub-string from the given string by slicing it respectively from start to end.

For understanding slicing we will use different methods, here we will cover 2 methods of string slicing, one using the in-built slice() method and another using the [:] array slice. String slicing in Python is about obtaining a sub-string from the given string by slicing it respectively from start to end.

Python slicing can be done in two ways:

- Using a slice() method
- Using the array slicing [:] method

Index tracker for positive and negative index: String indexing and slicing in python. Here, the Negative comes into consideration when tracking the string in reverse.

0	1	2	3	4	5	6
A	S	T	R	I	N	G

-7	-6	-5	-4	-3	-2	-1
A	S	T	R	I	N	G

1: Using the slice() method.

The slice() constructor creates a slice object representing the set of indices specified by range(start, stop, step).

Syntax:

slice(stop)

slice(start, stop, step)

Parameters:

start: Starting index where the slicing of object starts.

stop: Ending index where the slicing of object stops.

step: It is an optional argument that determines the increment between each index for slicing.

Return Type: Returns a sliced object containing elements in the given range only.

```
# Python program to demonstrate
# string slicing
```

```
# String slicing
String = 'ASTRING'
```

```
# Using slice constructor
s1 = slice(3)
s2 = slice(1, 5, 2)
s3 = slice(-1, -12, -2)
```

```
print("String slicing")
print(String[s1])
print(String[s2])
print(String[s3])
```

Output: String slicing

AST

SR

GITA

2: Using the List/array slicing [::] method

In Python, indexing syntax can be used as a substitute for the slice object. This is an easy and convenient way to slice a string using list slicing and Array slicing both syntax-wise and execution-wise. A start, end, and step have the same mechanism as the slice() constructor.

Below we will see string slicing in Python with examples.

Syntax

arr[start:stop]	# items start through stop-1
arr[start:]	# items start through the rest of the array
arr[:stop]	# items from the beginning through stop-1
arr[:]	# a copy of the whole array
arr[start:stop:step]	# start through not past stop, by step

Example 1:

In this example, we will see slicing in python list the index start from 0 indexes and ending with a 2 index(stops at 3-1=2).


```
# Python program to demonstrate
# string slicing

# String slicing
String = "Welcome to Python"

# Using indexing sequence
print(String[:3])
Output: Wel
```

Example 2:

In this example, we will see the example of starting from 1 index and ending with a 5 index(stops at 3-1=2), and the skipping step is 2. It is a good example of Python slicing string by character.

```
# Python program to demonstrate
# string slicing
# String slicing
String = "WELCOME TO PYTHON"

# Using indexing sequence
print(String[1:5:2])

Output: EC
```

Example 3:

In this example, we will see the example of starting from -1 indexes and ending with a -12 index(stops at 3-1=2)and the skipping step is -2.

```
# Python program to demonstrate
# string slicing

# String slicing
String = 'WELCOME TO PYTHON'

# Using indexing sequence
print(String[-1:-12:-2])

Output: NHY TE
```

Example 4:

In this example, the whole string is printed in reverse order.

```
# Python program to demonstrate
# string slicing
```

```
# String slicing
String = 'WELCOME TO PYTHON'
```

```
# Prints string in reverse
print(String[::-1])
```

Output: NOHTYP OT EMOCLEW

Using islice()

The islice() is a built-in function defined in itertools module. It is used to get an iterator which is an index-based slicing of any iterable. It works like a standard slice but returns an iterator.

Syntax:

itertools.islice(iterable, start, stop[, step])

Parameters:

iterable: Any iterable sequence like list, string, tuple etc.

start: The start index from where the slicing of iterable starts.

stop: The end index from where the slicing of iterable ends.

step: An optional argument. It specifies the gap between each index for slicing.

Return Type: Return an iterator from the given iterable sequence.

```
# Python program to demonstrate
# islice()
```

```
import itertools
```

```
# Using islice()
String = 'WELCOME TO PYTHON'
```

```
# prints characters from 3 to 7 skipping one character.
print(''.join(itertools.islice(String, 3, 7)))
#This code is contributed by Edula Vinay Kumar Reddy
```

Output: COME

5. Deleting/Updating from a String

In Python, the Updation or deletion of characters from a String is not allowed. This will cause an error because item assignment or item deletion from a String is

not supported. Although deletion of the entire String is possible with the use of a built-in del keyword. This is because Strings are immutable, hence elements of a String cannot be changed once it has been assigned. Only new strings can be reassigned to the same name.

Updation of a character:

```
# Python Program to Update  
# character of a String
```

```
String1 = "Hello, I'm a Python"  
print("Initial String: ")  
print(String1)
```

```
# Updating a character of the String  
## As python strings are immutable, they don't support item updation directly  
### there are following two ways
```

```
#1  
list1 = list(String1)  
list1[2] = 'p'  
String2 = ''.join(list1)  
print("\nUpdating character at 2nd Index: ")  
print(String2)
```

```
#2  
String3 = String1[0:2] + 'p' + String1[3:]  
print(String3)
```

Output: Initial String:
Hello, I'm a Python
Updating character at 2nd Index:
Heplo, I'm a Python
Heplo, I'm a Python

Updating Entire String:

```
# Python Program to Update  
# entire String
```

```
String1 = "Hello, I'm a Python"  
print("Initial String: ")  
print(String1)
```

```
# Updating a String  
String1 = "Welcome to the Python World"
```

```
print("\nUpdated String: ")
print(String1)
```

Output: Initial String:
Hello, I'm a Python
Updated String:
Welcome to the Python World

Deletion of a character:

```
# Python Program to Delete
```

```
# characters from a String
```

```
String1 = "Hello, I'm a Python"
print("Initial String: ")
print(String1)

# Deleting a character
# of the String
String2 = String1[0:2] + String1[3:]
print("\nDeleting character at 2nd Index: ")
print(String2)
```

Output: Initial String:
Hello, I'm a Python
Deleting character at 2nd Index:
Helo, I'm a Python

Deleting Entire String:

Deletion of the entire string is possible with the use of del keyword. Further, if we try to print the string, this will produce an error because String is deleted and is unavailable to be printed.

```
# Python Program to Delete
# entire String
```

```
String1 = "Hello, I'm a Python"
print("Initial String: ")
print(String1)
```

```
# Deleting a String
# with the use of del
```

```
del String1
print("\nDeleting entire String: ")
print(String1)
Output:
```

Initial String:

Hello, I'm a Python

Deleting entire String:

Traceback (most recent call last):

File "G:/Academic/Notes/Python/Programme/EntireStringDeletion.py",
line 12, in <module>

```
print(String1)
```

NameError: name 'String1' is not defined

6. Escape Sequencing in Python

While printing Strings with single and double quotes in it causes SyntaxError because String already contains Single and Double Quotes and hence cannot be printed with the use of either of these. Hence, to print such a String either Triple Quotes are used or Escape sequences are used to print such Strings.

Escape sequences start with a backslash and can be interpreted differently. If single quotes are used to represent a string, then all the single quotes present in the string must be escaped and same is done for Double Quotes.

Python Program for Escape Sequencing of String

Initial String

```
String1 = '''I'm a "Python"'''
```

```
print("Initial String with use of Triple Quotes: ")
```

```
print(String1)
```

Escaping Single Quote

```
String1 = 'I\'m a "Python"'
```

```
print("\nEscaping Single Quote: ")
```

```
print(String1)
```

Escaping Double Quotes

```
String1 = "I'm a \"Python\""
```

```
print("\nEscaping Double Quotes: ")
```

```
print(String1)
```

Printing Paths with the

use of Escape Sequences

```
String1 = "C:\\Python\\Python\\"
```

```
print("\nEscaping Backslashes: ")
```

```

print(String1)

# Printing Paths with the
# use of Tab
String1 = "Hi\tPython"
print("\nTab: ")
print(String1)

# Printing Paths with the
# use of New Line
String1 = "Python\nBooks"
print("\nNew Line: ")
print(String1)
Output:

```

Initial String with use of Triple Quotes:

I'm a "Python"

Escaping Single Quote:

I'm a "Python"

Escaping Double Quotes:

I'm a "Python"

Escaping Backslashes:

C:\Python\Python\

Tab:

Hi Python

New Line:

Python

Books

To ignore the escape sequences in a String, r or R is used, this implies that the string is a raw string and escape sequences inside it are to be ignored.

```

# Printing hello in octal
String1 = "\110\145\154\154\157"
print("\nPrinting in Octal with the use of Escape Sequences: ")
print(String1)

```

```

# Using raw String to
# ignore Escape Sequences
String1 = r"This is \110\145\154\154\157"
print("\nPrinting Raw String in Octal Format: ")
print(String1)

```

```

# Printing Geeks in HEX
String1 = "This is \x47\x65\x65\x6b\x73 in \x48\x45\x58"
print("\nPrinting in HEX with the use of Escape Sequences: ")
print(String1)

# Using raw String to
# ignore Escape Sequences
String1 = r"This is \x47\x65\x65\x6b\x73 in \x48\x45\x58"
print("\nPrinting Raw String in HEX Format: ")
print(String1)

```

Output:

Printing in Octal with the use of Escape Sequences:

Hello

Printing Raw String in Octal Format:

This is \110\145\154\154\157

Printing in HEX with the use of Escape Sequences:

This is Geeks in HEX

Printing Raw String in HEX Format:

This is \x47\x65\x65\x6b\x73 in \x48\x45\x58

7. Formatting of Strings:

Strings in Python can be formatted with the use of format() method which is a very versatile and powerful tool for formatting Strings. Format method in String contains curly braces {} as placeholders which can hold arguments according to position or keyword to specify the order.

```

# Python Program for Formatting of Strings Default order
String1 = "{} {} {}".format('Python', 'For', 'Life')
print("Print String in default order: ")
print(String1)

# Positional Formatting
String1 = "{1} {0} {2}".format('Python', 'For', 'Life')
print("\nPrint String in Positional order: ")
print(String1)

# Keyword Formatting
String1 = "{l} {f} {p}".format(p='Python', f='For', l='Life')
print("\nPrint String in order of Keywords: ")
print(String1)

```

Output: Print String in default order:
 Python For Life

 Print String in Positional order:
 For Python Life

 Print String in order of Keywords:
 Life For Python

Built-In Function	Description
string.ascii_letters	Concatenation of the ascii_lowercase and ascii_uppercase constants.
string.ascii_lowercase	Concatenation of lowercase letters
string.ascii_uppercase	Concatenation of uppercase letters
string.digits	Digit in strings
string.hexdigits	Hexadigit in strings
string.letters	concatenation of the strings lowercase and uppercase
string.lowercase	A string must contain lowercase letters.
string.octdigits	Octadigit in a string
string.punctuation	ASCII characters having punctuation characters.
string.printable	String of characters which are printable
String.endswith()	Returns True if a string ends with the given suffix otherwise returns False
String.startswith()	Returns True if a string starts with the given prefix otherwise returns False
String.isdigit()	Returns “True” if all characters in the string are digits, Otherwise, It returns “False”.
String.isalpha()	Returns “True” if all characters in the string are alphabets, Otherwise, It returns “False”.
string.isdecimal()	Returns true if all characters in a string are decimal.

<code>str.format()</code>	one of the string formatting methods in Python3, which allows multiple substitutions and value formatting.
<code>String.index</code>	Returns the position of the first occurrence of substring in a string
<code>string.uppercase</code>	A string must contain uppercase letters.
<code>string.whitespace</code>	A string containing all characters that are considered whitespace.
<code>string.swapcase()</code>	Method converts all uppercase characters to lowercase and vice versa of the given string, and returns it
<code>replace()</code>	returns a copy of the string where all occurrences of a substring is replaced with another substring.

Advantages of String in Python:

- Strings are used at a larger scale i.e. for a wide areas of operations such as storing and manipulating text data, representing names, addresses, and other types of data that can be represented as text.
- Python has a rich set of string methods that allow you to manipulate and work with strings in a variety of ways. These methods make it easy to perform common tasks such as converting strings to uppercase or lowercase, replacing substrings, and splitting strings into lists.
- Strings are immutable, meaning that once you have created a string, you cannot change it. This can be beneficial in certain situations because it means that you can be confident that the value of a string will not change unexpectedly.
- Python has built-in support for strings, which means that you do not need to import any additional libraries or modules to work with strings. This makes it easy to get started with strings and reduces the complexity of your code.
- Python has a concise syntax for creating and manipulating strings, which makes it easy to write and read code that works with strings.

Drawbacks of String in Python:

- When we are dealing with large text data, strings can be inefficient. For instance, if you need to perform a large number of operations on a string, such as replacing substrings or splitting the string into multiple substrings, it can be slow and consume a lot resources.

- Strings can be difficult to work with when you need to represent complex data structures, such as lists or dictionaries. In these cases, it may be more efficient to use a different data type, such as a list or a dictionary, to represent the data.

PYTHON LIST:

Python Lists are just like dynamically sized arrays, declared in other languages (vector in C++ and ArrayList in Java). In simple language, a list is a collection of things, enclosed in [] and separated by commas.

```
Var = ["Python", "for", "Python"]  
print(Var)
```

Output: ["Python", "for", "Python"]

Lists are the simplest containers that are an integral part of the Python language. Lists need not be homogeneous always which makes it the most powerful tool in Python. A single list may contain DataTypes like Integers, Strings, as well as Objects. Lists are mutable, and hence, they can be altered even after their creation.

1. Creating a List in Python

Lists in Python can be created by just placing the sequence inside the square brackets[].

Unlike Sets, a list doesn't need a built-in function for its creation of a list.

Example 1: Creating a list in Python

```
# Python program to demonstrate Creation of List
```

```
# Creating a List
```

```
List = []  
print("Blank List: ")  
print(List)
```

```
# Creating a List of numbers
```

```
List = [10, 20, 14]  
print("\nList of numbers: ")  
print(List)
```

```
# Creating a List of strings and accessing
```

```
# using index
```

```
List = ["Welcome ", "TO", "ATESTC"]  
print("\nList Items: ")
```

```
print(List[0])
```

```
print(List[2])
```

Output: Blank List:

```
[]
```

List of numbers:

```
[10, 20, 14]
```

List Items:

```
Welcome
```

```
ATESTC
```

Example 2: Creating a list with multiple distinct or duplicate elements

A list may contain duplicate values with their distinct positions and hence, multiple distinct or duplicate values can be passed as a sequence at the time of list creation.

Creating a List with the use of Numbers (Having duplicate values)

```
List = [1, 2, 4, 4, 3, 3, 3, 6, 5]
```

```
print("\nList with the use of Numbers: ")
```

```
print(List)
```

Creating a List with mixed type of values (Having numbers and strings)

```
List = [1, 2, 'Python', 4, 'For', 6, 'World']
```

```
print("\nList with the use of Mixed Values: ")
```

```
print(List)
```

Output: List with the use of Numbers:

```
[1, 2, 4, 4, 3, 3, 3, 6, 5]
```

List with the use of Mixed Values:

```
[1, 2, 'Python', 4, 'For', 6, 'World']
```

2. Accessing elements from the List

In order to access the list items refer to the index number. Use the index operator [] to access an item in a list. The index must be an integer. Nested lists are accessed using nested indexing.

Example 1: Accessing elements from list

Python program to demonstrate accessing of element from list

Creating a List with the use of multiple values

```
List = ["Python", "For", "World"]
```

accessing a element from the list using index number

```
print("Accessing a element from the list")
```

```
print(List[0])
```

```
print(List[2])
```

Output: Accessing a element from the list

 Python

 World

Example 2: Accessing elements from a multi-dimensional list

```
# Creating a Multi-Dimensional List (By Nesting a list inside a List)
```

```
List = [['Python', 'For'], ['World']]
```

```
# accessing an element from the Multi-Dimensional List using index number
```

```
print("Accessing a element from a Multi-Dimensional list")
```

```
print(List[0][1])
```

```
print(List[1][0])
```

Output: Accessing a element from a Multi-Dimensional list

 For

 World

Negative indexing

In Python, negative sequence indexes represent positions from the end of the array. Instead of having to compute the offset as in `List[len(List)-3]`, it is enough to just write `List[-3]`. Negative indexing means beginning from the end, -1 refers to the last item, -2 refers to the second-last item, etc.

```
List = [1, 2, 'Python', 4, 'For', 6, 'World']
```

```
# accessing an element using negative indexing
```

```
print("Accessing element using negative indexing")
```

```
# print the last element of list
```

```
print(List[-1])
```

```
# print the third last element of list
```

```
print(List[-3])
```

Output: Accessing element using negative indexing

 World

 For

3. Getting the size of Python list

Python **len()** is used to get the length of the list.

```
# Creating a List
```

```
List1 = []
```

```
print(len(List1))
```

```
# Creating a List of numbers
```

```
List2 = [10, 20, 14]
```

```
print(len(List2))
```

Output: 0
 3

4. Taking Input of a Python List

We can take the input of a list of elements as string, integer, float, etc. But the default one is a string.

Example 1:

```
# Python program to take space separated input as a string split and store it to a list and print the string list
```

```
# input the list as string
```

```
string = input("Enter elements (Space-Separated): ")
```

```
# split the strings and store it to a list
```

```
lst = string.split()
```

```
print('The list is:', lst) # printing the list
```

Output: Enter elements (Space-Separated): Python For World
 The list is: ['Python', 'For', 'World']

Example 2:

```
# input size of the list
```

```
n = int(input("Enter the size of list : "))
```

```
# store integers in a list using map, split and strip functions
```

```
lst = list(map(int, input("Enter the integer elements:").strip().split()))[:n]
```

```
# printing the list
```

```
print('The list is:', lst)
```

Output: Enter the size of list : 4
 Enter the integer elements:10 20 30 40
 The list is: [10, 20, 30, 40]

5. Adding Elements to a Python List

Method 1: Using `append()` method

Elements can be added to the List by using the built-in **`append()`** function. Only one element at a time can be added to the list by using the **`append()`** method, for the addition of multiple elements with the **`append()`** method, loops are used. Tuples can also be added to the list with the use of the `append` method because tuples are immutable. Unlike Sets, Lists can also be added to the existing list with the use of the **`append()`** method.

Python program to demonstrate Addition of elements in a List

Creating a List

```
List = []
```

```
print("Initial blank List: ")
```

```
print(List)
```

Addition of Elements in the List

```
List.append(1)
```

```
List.append(2)
```

```
List.append(4)
```

```
print("\nList after Addition of Three elements: ")
```

```
print(List)
```

Adding elements to the List using Iterator

```
for i in range(1, 4):
```

```
    List.append(i)
```

```
print("\nList after Addition of elements from 1-3: ")
```

```
print(List)
```

Adding Tuples to the List

```
List.append((5, 6))
```

```
print("\nList after Addition of a Tuple: ")
```

```
print(List)
```

Addition of List to a List

```
List2 = ['For', 'Python']
```

```
List.append(List2)
```

```
print("\nList after Addition of a List: ")
```

```
print(List)
```

Output: Initial blank List:

```
[]
```

List after Addition of Three elements:

```
[1, 2, 4]
```

List after Addition of elements from 1-3:

[1, 2, 4, 1, 2, 3]

List after Addition of a Tuple:

[1, 2, 4, 1, 2, 3, (5, 6)]

List after Addition of a List:

[1, 2, 4, 1, 2, 3, (5, 6), ['For', 'Python']]

Method 2: Using insert() method

append() method only works for the addition of elements at the end of the List, for the addition of elements at the desired position, **insert()** method is used. Unlike **append()** which takes only one argument, the **insert()** method requires two arguments (position, value).

```
# Python program to demonstrate Addition of elements in a List
# Creating a List
List = [1,2,3,4]
print("Initial List: ")
print(List)
# Addition of Element at specific Position (using Insert Method)
List.insert(3, 12)
List.insert(0, 'Python')
print("\nList after performing Insert Operation: ")
print(List)
```

Output: Initial List:

[1, 2, 3, 4]

List after performing Insert Operation:

['Python', 1, 2, 3, 12, 4]

Method 3: Using extend() method

Other than **append()** and **insert()** methods, there's one more method for the Addition of elements, **extend()**, this method is used to add multiple elements at the same time at the end of the list.

Note: **append()** and **extend()** methods can only add elements at the end.

```
# Python program to demonstrate Addition of elements in a List
# Creating a List
```

```

List = [1, 2, 3, 4]
print("Initial List: ")
print(List)
# Addition of multiple elements to the List at the end (using Extend Method)
List.extend([8, 'Python', 'Always'])
print("\nList after performing Extend Operation: ")
print(List)

```

Output: Initial List:

 [1, 2, 3, 4]

 List after performing Extend Operation:

 [1, 2, 3, 4, 8, 'Python', 'Always']

6. Reversing a List

A list can be reversed by using the **reverse()** method in Python.

```

# Reversing a list
mylist = [1, 2, 3, 4, 5, 'Welcome', 'Python']
mylist.reverse()
print(mylist)

```

Output: ['Python', 'Welcome', 5, 4, 3, 2, 1]

7. Removing Elements from the List

Method 1: Using **remove()** method

Elements can be removed from the List by using the built-in **remove()** function but an Error arises if the element doesn't exist in the list. **Remove()** method only removes one element at a time, to remove a range of elements, the iterator is used. The **remove()** method removes the specified item.

Note: Remove method in List will only remove the first occurrence of the searched element.

Example 1:

```

# Python program to demonstrate Removal of elements in a List
# Creating a List
List = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

```



```

print("Initial List: ")
print(List)
# Removing elements from List using Remove() method
List.remove(5)
List.remove(6)
print("\nList after Removal of two elements: ")
print(List)

```

Output: Initial List:

[1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]

 List after Removal of two elements:

[1, 2, 3, 4, 7, 8, 9, 10, 11, 12]

Example 2:

```

# Creating a List
List = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12]
# Removing elements from List using iterator method
for i in range(1, 5):
    List.remove(i)
print("\nList after Removing a range of elements: ")
print(List)

```

Output: List after Removing a range of elements:

[5, 6, 7, 8, 9, 10, 11, 12]

Method 2: Using pop() method

pop() function can also be used to remove and return an element from the list, but by default it removes only the last element of the list, to remove an element from a specific position of the List, the index of the element is passed as an argument to the **pop()** method.

```

List = [1, 2, 3, 4, 5]
# Removing element from the Set using the pop() method
List.pop()
print("\nList after popping an element: ")

```

```

print(List)
# Removing element at a specific location from the Set using the pop() method
List.pop(2)
print("\nList after popping a specific element: ")
print(List)

```

Output: List after popping an element:
 [1, 2, 3, 4]
 List after popping a specific element:
 [1, 2, 4]

8. Slicing of a List

We can get substrings and sublists using a slice. In Python List, there are multiple ways to print the whole list with all the elements, but to print a specific range of elements from the list, we use the Slice operation.

Slice operation is performed on Lists with the use of a colon(:).

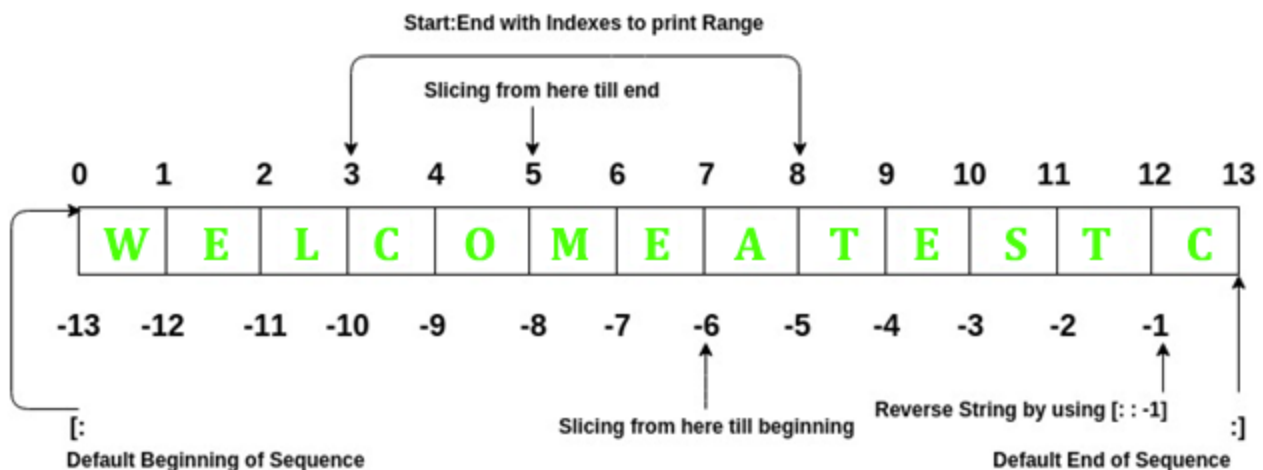
To print elements from beginning to a range use: **[: Index]**

To print elements from end-use: **[:-Index]**

To print elements from a specific Index till the end use: **[Index:]**

To print the whole list in reverse order, use: **[::-1]**

Note – To print elements of List from rear-end, use Negative Indexes.



UNDERSTANDING SLICING OF LISTS:

pr[0] accesses the first item, 2.

pr[-4] accesses the fourth item from the end, 5.

pr[2:] accesses [5, 7, 11, 13], a list of items from third to last.

pr[:4] accesses [2, 3, 5, 7], a list of items from first to fourth.

pr[2:4] accesses [5, 7], a list of items from third to fifth.

pr[1::2] accesses [3, 7, 13], alternate items, starting from the second item.

Python program to demonstrate Removal of elements in a List

Creating a List

```
List = ['W', 'E', 'L', 'C', 'O', 'M', 'E', 'A', 'T', 'E', 'S', 'T', 'C']
```

```
print("Initial List: ")
```

```
print(List)
```

Print elements of a range using Slice operation

```
Sliced_List = List[3:8]
```

```
print("\nSlicing elements in a range 3-8: ")
```

```
print(Sliced_List)
```

Print elements from a pre-defined point to end

```
Sliced_List = List[5:]
```

```
print("\nElements sliced from 5th element till the end: ")
```

```
print(Sliced_List)
```

Printing elements from beginning till end

```
Sliced_List = List[:]
```

```
print("\nPrinting all elements using slice operation: ")
```

```
print(Sliced_List)
```

Output: Initial List:

```
['W', 'E', 'L', 'C', 'O', 'M', 'E', 'A', 'T', 'E', 'S', 'T', 'C']
```

Slicing elements in a range 3-8:

```
['C', 'O', 'M', 'E', 'A']
```

Elements sliced from 5th element till the end:

```
['M', 'E', 'A', 'T', 'E', 'S', 'T', 'C']
```

Printing all elements using slice operation:

```
['W', 'E', 'L', 'C', 'O', 'M', 'E', 'A', 'T', 'E', 'S', 'T', 'C']
```

Negative index List slicing

Creating a List

```
List = ['W', 'E', 'L', 'C', 'O', 'M', 'E', 'A', 'T', 'E', 'S', 'T', 'C']
```

```
print("Initial List: ")
```

```
print(List)
```

Print elements from beginning to a pre-defined point using Slice

```
Sliced_List = List[:-6]
```

```
print("\nElements sliced till 6th element from last: ")
```

```
print(Sliced_List)
```

Print elements of a range using negative index List slicing

```
Sliced_List = List[-6:-1]
```

```
print("\nElements sliced from index -6 to -1")
```

```
print(Sliced_List)
```

Printing elements in reverse using Slice operation

```
Sliced_List = List[::-1]
```

```
print("\nPrinting List in reverse: ")
```

```
print(Sliced_List)
```

Output: Initial List:

```
['W', 'E', 'L', 'C', 'O', 'M', 'E', 'A', 'T', 'E', 'S', 'T', 'C']
```

Elements sliced till 6th element from last:

```
['W', 'E', 'L', 'C', 'O', 'M', 'E']
```

Elements sliced from index -6 to -1

```
['A', 'T', 'E', 'S', 'T']
```

Printing List in reverse:

```
['C', 'T', 'S', 'E', 'T', 'A', 'E', 'M', 'O', 'C', 'L', 'E', 'W']
```

9. List Comprehension

Python List comprehensions are used for creating new lists from other iterables like tuples, strings, arrays, lists, etc. A list comprehension consists of brackets containing the expression, which is executed for each element along with the for loop to iterate over each element.

Syntax:

newList = [expression(element) for element in oldList if condition]

Example:

Python program to demonstrate list comprehension in Python below list contains
#square of all odd numbers from range 1 to 10

```
odd_square = [x ** 2 for x in range(1, 11) if x % 2 == 1]
print(odd_square)
```

Output: [1, 9, 25, 49, 81]

For better understanding, the above code is similar to as follows:

```
# for understanding, above generation is same as,
odd_square = []
for x in range(1, 11):
    if x % 2 == 1:
        odd_square.append(x**2)
```

```
print(odd_square)
```

Output: [1, 9, 25, 49, 81]

List Methods

S.no	Method	Description
1	append()	Used for appending and adding elements to the end of the List.
2	copy()	It returns a shallow copy of a list
3	clear()	This method is used for removing all items from the list.
4	count()	These methods count the elements
5	extend()	Adds each element of the iterable to the end of the List
6	index()	Returns the lowest index where the element appears.

7	insert()	Inserts a given element at a given index in a list.
8	pop()	Removes and returns the last value from the List or the given index value.
9	remove()	Removes a given object from the List.
10	reverse()	Reverses objects of the List in place.
11	sort()	Sort a List in ascending, descending, or user-defined order
12	min()	Calculates the minimum of all the elements of the List
13	max()	Calculates the maximum of all the elements of the List

Built-in functions with List

Function	Description
reduce()	apply a particular function passed in its argument to all of the list elements stores the intermediate result and only returns the final summation value
sum()	Sums up the numbers in the list
ord()	Returns an integer representing the Unicode code point of the given Unicode character
cmp()	This function returns 1 if the first list is “greater” than the second list
max()	return maximum element of a given list
min()	return minimum element of a given list
all()	Returns true if all element is true or if the list is empty
any()	return true if any element of the list is true. if the list is empty, return false
len()	Returns length of the list or size of the list
enumerate()	Returns enumerate object of the list
accumulate()	apply a particular function passed in its argument to all of the list elements returns a list containing the intermediate results
filter()	tests if each element of a list is true or not
map()	returns a list of the results after applying the given function to each item of a given iterable
lambda()	This function can have any number of arguments but only one expression, which is evaluated and returned.

Python Tuples:

Tuple is a collection of Python objects much like a list. The sequence of values stored in a tuple can be of any type, and they are indexed by integers.

Values of a tuple are syntactically separated by 'commas'. Although it is not necessary, it is more common to define a tuple by closing the sequence of values in parentheses. This helps in understanding the Python tuples more easily.

1. Creating a Tuple

In Python, tuples are created by placing a sequence of values separated by 'comma' with or without the use of parentheses for grouping the data sequence.

Note: Creation of Python tuple without the use of parentheses is known as Tuple Packing.

Python program to demonstrate the addition of elements in a Tuple.

Creating an empty Tuple

```
Tuple1 = ()
```

```
print("Initial empty Tuple: ")
```

```
print(Tuple1)
```

Creating a Tuple with the use of string

```
Tuple1 = ('Python', 'For')
```

```
print("\nTuple with the use of String: ")
```

```
print(Tuple1)
```

Creating a Tuple with the use of list

```
list1 = [1, 2, 4, 5, 6]
```

```
print("\nTuple using List: ")
```

```
print(tuple(list1))
```

Creating a Tuple with the use of built-in function

```
Tuple1 = tuple('World')
```

```
print("\nTuple with the use of function: ")
```

```
print(Tuple1)
```

Output: Initial empty Tuple:

 ()

 Tuple with the use of String:

```
('Python', 'For')
```

Tuple using List:

```
(1, 2, 4, 5, 6)
```

Tuple with the use of function:

```
('W', 'o', 'r', 'l', 'd')
```

Creating a Tuple with Mixed Datatypes.

Tuples can contain any number of elements and of any datatype (like strings, integers, list, etc.). Tuples can also be created with a single element, but it is a bit tricky. Having one element in the parentheses is not sufficient, there must be a trailing 'comma' to make it a tuple.

```
# Creating a Tuple with Mixed Datatype
```

```
Tuple1 = (5, 'Welcome', 7, 'Python')
```

```
print("\nTuple with Mixed Datatypes: ")
```

```
print(Tuple1)
```

```
# Creating a Tuple with nested tuples
```

```
Tuple1 = (0, 1, 2, 3)
```

```
Tuple2 = ('Python', 'World')
```

```
Tuple3 = (Tuple1, Tuple2)
```

```
print("\nTuple with nested tuples: ")
```

```
print(Tuple3)
```

```
# Creating a Tuple with repetition
```

```
Tuple1 = ('World',) * 3
```

```
print("\nTuple with repetition: ")
```

```
print(Tuple1)
```

```
# Creating a Tuple with the use of loop
```

```
Tuple1 = ('World')
```

```
n = 5
```

```
print("\nTuple with a loop")
```

```
for i in range(int(n)):
```

```
    Tuple1 = (Tuple1,)
```

```
    print(Tuple1)
```

Output: Tuple with Mixed Datatypes:


```
(5, 'Welcome', 7, 'Python')
Tuple with nested tuples:
((0, 1, 2, 3), ('Python', 'World'))
Tuple with repetition:
('World', 'World', 'World')
Tuple with a loop
('World',)
(('World',),)
((( 'World',),),)
(((( 'World',),),),)
((((('World',),),),),)
```

2. Accessing of Tuples

Tuples are immutable, and usually, they contain a sequence of heterogeneous elements that are accessed via unpacking or indexing (or even by attribute in the case of named tuples). Lists are mutable, and their elements are usually homogeneous and are accessed by iterating over the list.

Note: In unpacking of tuple number of variables on the left-hand side should be equal to a number of values in given tuple a.

Accessing Tuple with Indexing

```
Tuple1 = tuple("Python")
print("\nFirst element of Tuple: ")
print(Tuple1[0])
# Tuple unpacking
Tuple1 = ("Python ", "For", "World")
# This line unpack values of Tuple1
a, b, c = Tuple1
print("\nValues after unpacking: ")
print(a)
print(b)
print(c)
```

Output: First element of Tuple:
P

Values after unpacking:

Python

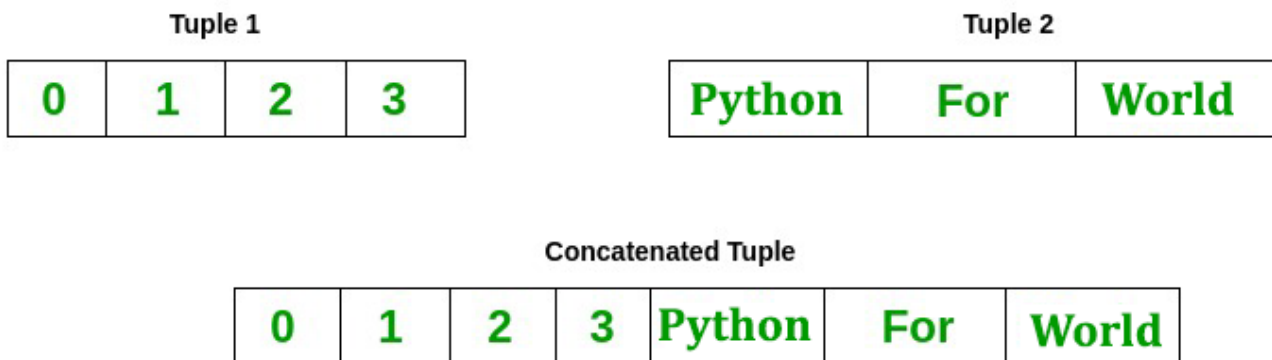
For

World

3. Concatenation of Tuples

Concatenation of tuple is the process of joining two or more Tuples. Concatenation is done by the use of '+' operator. Concatenation of tuples is done always from the end of the original tuple. Other arithmetic operations do not apply on Tuples.

Note- Only the same datatypes can be combined with concatenation, an error arises if a list and a tuple are combined.



```
# Concatenation of tuples
```

```
Tuple1 = (0, 1, 2, 3)
```

```
Tuple2 = ('Python', 'For', 'World')
```

```
Tuple3 = Tuple1 + Tuple2
```

```
# Printing first Tuple
```

```
print("Tuple 1: ")
```

```
print(Tuple1)
```

```
# Printing Second Tuple
```

```
print("\nTuple2: ")
```

```
print(Tuple2)
```

```
# Printing Final Tuple
```

```
print("\nTuples after Concatenation: ")
```

```
print(Tuple3)
```

Output: Tuple 1:

(0, 1, 2, 3)

Tuple2:

('Python', 'For', 'World')

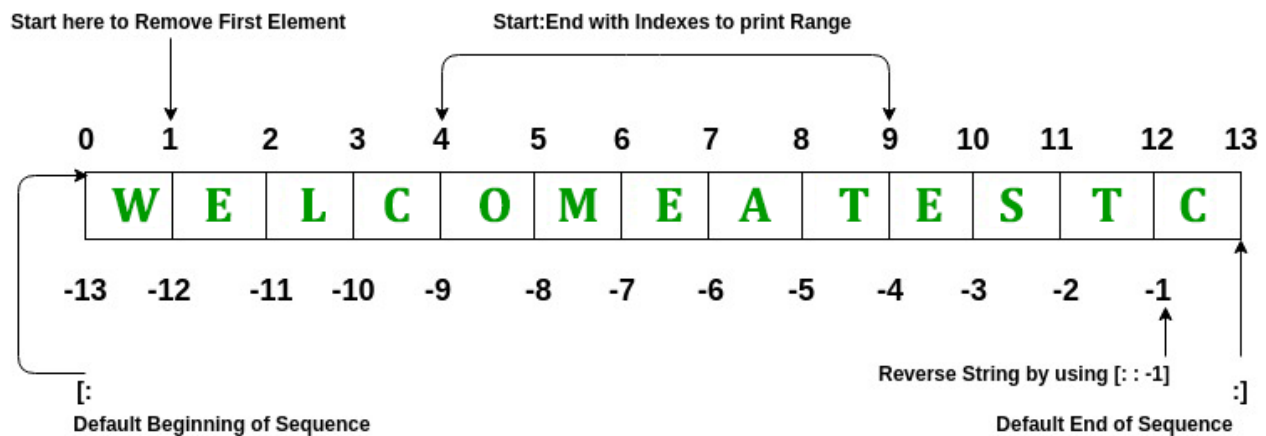
Tuples after Concatenation:

(0, 1, 2, 3, 'Python', 'For', 'World')

4. Slicing of Tuple:

Slicing of a Tuple is done to fetch a specific range or slice of sub-elements from a Tuple. Slicing can also be done to lists and arrays. Indexing in a list results to fetching a single element whereas Slicing allows to fetch a set of elements.

Note- Negative Increment values can also be used to reverse the sequence of Tuples.



Slicing of a Tuple with Numbers

```
Tuple1 = tuple('WELCOMEATESTC')
```

Removing First element

```
print("Removal of First Element: ")
```

```
print(Tuple1[1:])
```

Reversing the Tuple

```
print("\nTuple after sequence of Element is reversed: ")
```

```
print(Tuple1[::-1])
```

Printing elements of a Range

```
print("\nPrinting elements between Range 4-9: ")
```

```
print(Tuple1[4:9])
```

Output: Removal of First Element:

```
('E', 'L', 'C', 'O', 'M', 'E', 'A', 'T', 'E', 'S', 'T', 'C')
```

Tuple after sequence of Element is reversed:

```
('C', 'T', 'S', 'E', 'T', 'A', 'E', 'M', 'O', 'C', 'L', 'E', 'W')
```

Printing elements between Range 4-9:

```
('O', 'M', 'E', 'A', 'T')
```

5. Deleting a Tuple

Tuples are immutable and hence they do not allow deletion of a part of it. The entire tuple gets deleted by the use of del() method.

Note- Printing of Tuple after deletion results in an Error.

```
# Deleting a Tuple
```

```
Tuple1 = (0, 1, 2, 3, 4)
```

```
del Tuple1
```

```
print(Tuple1)
```

Output

Traceback (most recent call last):

File "G:/Academic/Notes/Python/Programme/TupleDemo6.py", line 6, in
<module>

```
print(Tuple1)
```

NameError: name 'Tuple1' is not defined. Did you mean: 'tuple'?

Built-In Methods

Built-in-Method	Description
<u>index()</u>	Find in the tuple and returns the index of the given value where it's available
<u>count()</u>	Returns the frequency of occurrence of a specified value

Built-In Functions

Built-in Function	Description
all()	Returns true if all element are true or if tuple is empty
any()	return true if any element of the tuple is true. if tuple is empty, return false
len()	Returns length of the tuple or size of the tuple
enumerate()	Returns enumerate object of tuple
max()	return maximum element of given tuple
min()	return minimum element of given tuple
sum()	Sums up the numbers in the tuple
sorted()	input elements in the tuple and return a new sorted list
tuple()	Convert an iterable to a tuple.

Tuples VS Lists:

Similarities	Differences
Functions that can be used for both lists and tuples: len(), max(), min(), sum(), any(), all(), sorted()	Methods that cannot be used for tuples: append(), insert(), remove(), pop(), clear(), sort(), reverse()
Methods that can be used for both lists and tuples: count(), Index()	we generally use 'tuples' for heterogeneous (different) data types and 'lists' for homogeneous (similar) data types.
Tuples can be stored in lists.	Iterating through a 'tuple' is faster than in a 'list'.
Lists can be stored in tuples.	'Lists' are mutable whereas 'tuples' are immutable.
Both 'tuples' and 'lists' can be nested.	Tuples that contain immutable elements can be used as a key for a dictionary.

PYTHON DICTIONARY

Dictionary in Python is a collection of keys values, used to store data values like a map, which, unlike other data types which hold only a single value as an element.

Dictionary holds **key:value** pair. Key-Value is provided in the dictionary to make it more optimized.

```
Dict = {1: 'Python', 2: 'For', 3: 'Python'}  
print(Dict)
```

Output: {1: 'Python', 2: 'For', 3: 'Python'}

1. Creating a Dictionary:

In Python, a dictionary can be created by placing a sequence of elements within **curly {} braces**, separated by '**comma**'. Dictionary holds **pairs of values**, one being the Key and the other corresponding pair element being its **Key:value**. Values in a dictionary can be of any data type and can be duplicated, whereas keys can't be repeated and must be immutable.

Note – Dictionary keys are case sensitive, the same name but different cases of Key will be treated distinctly.

Creating a Dictionary with Integer Keys

```
Dict = {1: 'Python', 2: 'For', 3: 'Python'}  
print("\nDictionary with the use of Integer Keys: ")  
print(Dict)
```

Creating a Dictionary with Mixed keys

```
Dict = {'Name': 'Python', 1: [1, 2, 3, 4]}  
print("\nDictionary with the use of Mixed Keys: ")  
print(Dict)
```

Output: Dictionary with the use of Integer Keys:
 {1: 'Python', 2: 'For', 3: 'Python'}
 Dictionary with the use of Mixed Keys:
 {'Name': 'Python', 1: [1, 2, 3, 4]}

Dictionary can also be created by the built-in function dict(). An empty dictionary can be created by just placing to curly braces{}

Creating an empty Dictionary

```
Dict = {}
```

```
print("Empty Dictionary: ")
```

```
print(Dict)
```

Creating a Dictionary with dict() method

```
Dict = dict({1: 'Python', 2: 'For', 3: 'Python'})
```

```
print("\nDictionary with the use of dict(): ")
```

```
print(Dict)
```

Creating a Dictionary with each item as a Pair

```
Dict = dict([(1, 'Python'), (2, 'For')])
```

```
print("\nDictionary with each item as a pair: ")
```

```
print(Dict)
```

Output: Empty Dictionary:

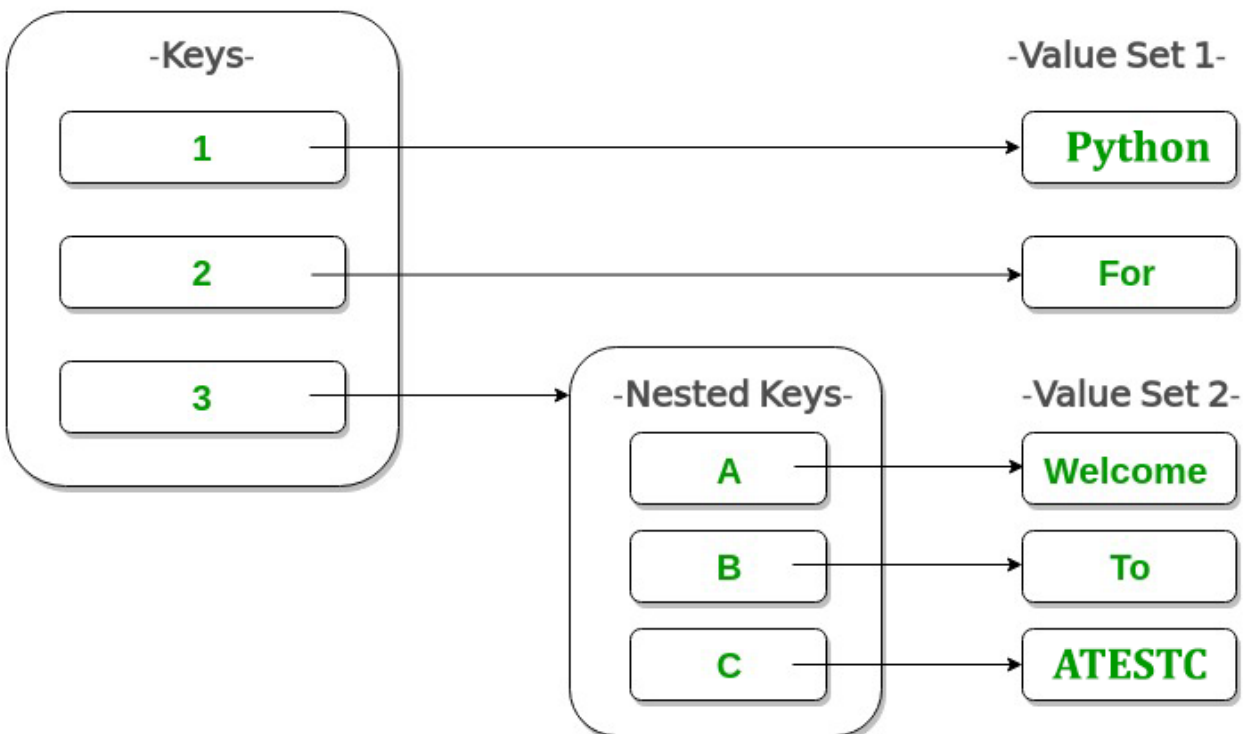
{}

Dictionary with the use of dict():

{1: 'Python', 2: 'For', 3: 'Python'}

Dictionary with each item as a pair:

{1: 'Python', 2: 'For'}



Creating a Nested Dictionary as shown in the above image

```
Dict = {1: 'Python', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'ATESTC'}}  
print(Dict)
```

Output: {1: 'Python', 2: 'For', 3: {'A': 'Welcome', 'B': 'To', 'C': 'ATESTC'}}

2. Adding elements to a Dictionary

Addition of elements can be done in multiple ways. One value at a time can be added to a Dictionary by defining value along with the key e.g. Dict[Key] = 'Value'. Updating an existing value in a Dictionary can be done by using the built-in update() method. Nested key values can also be added to an existing Dictionary.

Note- While adding a value, if the key-value already exists, the value gets updated otherwise a new Key with the value is added to the Dictionary.

Creating an empty Dictionary

```
Dict = {}  
print("Empty Dictionary: ")  
print(Dict)
```

Adding elements one at a time

```
Dict[0] = 'Python'  
Dict[2] = 'For'  
Dict[3] = 1  
print("\nDictionary after adding 3 elements: ")  
print(Dict)
```

Adding set of values to a single Key

```
Dict['Value_set'] = 2, 3, 4  
print("\nDictionary after adding 3 elements: ")  
print(Dict)
```

Updating existing Key's Value

```
Dict[2] = 'Welcome'  
print("\nUpdated key value: ")  
print(Dict)
```



```
# Adding Nested Key value to Dictionary
```

```
Dict[5] = {'Nested': {'1': 'Life', '2': 'Python'}}
```

```
print("\nAdding a Nested Key: ")
```

```
print(Dict)
```

Output: Empty Dictionary:

```
{}
```

Dictionary after adding 3 elements:

```
{0: 'Python', 2: 'For', 3: 1}
```

Dictionary after adding 3 elements:

```
{0: 'Python', 2: 'For', 3: 1, 'Value_set': (2, 3, 4)}
```

Updated key value:

```
{0: 'Python', 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4)}
```

Adding a Nested Key:

```
{0: 'Python', 2: 'Welcome', 3: 1, 'Value_set': (2, 3, 4), 5: {'Nested': {'1': 'Life',  
'2': 'Python'}}}
```

3. Accessing elements of a Dictionary

In order to access the items of a dictionary refer to its key name. Key can be used inside square brackets.

```
# Python program to demonstrate accessing a element from a Dictionary
```

```
# Creating a Dictionary
```

```
Dict = {1: 'Python', 'name': 'For', 3: 'Python'}
```

```
# accessing a element using key
```

```
print("Accessing a element using key:")
```

```
print(Dict['name'])
```

```
# accessing a element using key
```

```
print("Accessing a element using key:")
```

```
print(Dict[1])
```

Output: Accessing a element using key:

For

Accessing a element using key:

Python

There is also a method called **get()** that will also help in accessing the element from a dictionary. This method accepts key as argument and returns the value.

Creating a Dictionary

```
Dict = {1: 'Python', 'name': 'For', 3: 'Python'}
```

accessing a element using get() method

```
print("Accessing a element using get:")
```

```
print(Dict.get(3))
```

Output: Accessing a element using get:

Python

4. Accessing an element of a nested dictionary

In order to access the value of any key in the nested dictionary, use indexing []

Creating a Dictionary

```
Dict = {'Dict1': {1: 'Python'}, 'Dict2': {'Name': 'For'}}
```

Accessing element using key

```
print(Dict['Dict1'])
```

```
print(Dict['Dict1'][1])
```

```
print(Dict['Dict2']['Name'])
```

Output: {1: 'Python'}

Python

For

5. Deleting Elements using del Keyword

The items of the dictionary can be deleted by using the del keyword as given below.

Python program to demonstrate Deleting Elements using del Keyword

Creating a Dictionary

```
Dict = {1: 'Python', 'name': 'For', 3: 'Python'}
```

```
print("Dictionary =")
```

```
print(Dict)
```

#Deleting some of the Dictionar data

```
del(Dict[1])
```

```
print("Data after deletion Dictionary=")
```

```
print(Dict)
```

Output: Dictionary =

{1: 'Python', 'name': 'For', 3: 'Python'}

Data after deletion Dictionary=

{'name': 'For', 3: 'Python'}

6. Dictionary methods

Method	Description
dic.clear()	Remove all the elements from the dictionary
dict.copy()	Returns a copy of the dictionary
dict.get(key, default = "None")	Returns the value of specified key
dict.items()	Returns a list containing a tuple for each key value pair
dict.keys()	Returns a list containing dictionary's keys
dict.update(dict2)	Updates dictionary with specified key-value pairs
dict.values()	Returns a list of all the values of dictionary
pop()	Remove the element with specified key
popItem()	Removes the last inserted key-value pair
dict.setdefault(key,default= "None")	set the key to the default value if the key is not specified in the dictionary
dict.has_key(key)	returns true if the dictionary contains the specified key.
dict.get(key, default = "None")	used to get the value specified for the passed key.

demo for all dictionary methods

```
dict1 = {1: "Python", 2: "Java", 3: "Ruby", 4: "Scala"}
```

copy() method

```
dict2 = dict1.copy()
```

```
print(dict2)
```

clear() method

```

dict1.clear()
print(dict1)
# get() method
print(dict2.get(1))
# items() method
print(dict2.items())
# keys() method
print(dict2.keys())
# pop() method
dict2.pop(4)
print(dict2)
# popitem() method
dict2.popitem()
print(dict2)
# update() method
dict2.update({3: "Scala"})
print(dict2)
# values() method
print(dict2.values())

```

Output: {1: 'Python', 2: 'Java', 3: 'Ruby', 4: 'Scala'}

{}

Python

dict_items([(1, 'Python'), (2, 'Java'), (3, 'Ruby'), (4, 'Scala')])

dict_keys([1, 2, 3, 4])

{1: 'Python', 2: 'Java', 3: 'Ruby'}

{1: 'Python', 2: 'Java'}

{1: 'Python', 2: 'Java', 3: 'Scala'}

dict_values(['Python', 'Java', 'Scala'])

PYTHON SETS:

In Python, a **Set** is an **unordered collection** of data types that is **iterable, mutable** and **has no duplicate** elements. The order of elements in a set is undefined though it may consist of various elements. The major advantage of using a set, as opposed to a list, is that it has a highly optimized method for checking whether a specific element is contained in the set.

1. Creating a Set:

Sets can be created by using the built-in **set()** function with an iterable object or a sequence by placing the sequence inside curly braces, separated by a 'comma'.

Note: A set cannot have mutable elements like a list or dictionary, as it is mutable.

Python program to demonstrate Creation of Set in Python

Creating a Set

```
set1 = set()
```

```
print("Initial blank Set: ")
```

```
print(set1)
```

Creating a Set with the use of a String

```
set1 = set("PythonForPython")
```

```
print("\nSet with the use of String: ")
```

```
print(set1)
```

Creating a Set with the use of Constructor (Using object to Store String)

```
String = ' PythonForPython'
```

```
set1 = set(String)
```

```
print("\nSet with the use of an Object: ")
```

```
print(set1)
```

Creating a Set with the use of a List

```
set1 = set(["Python", "For", "Python"])
```

```
print("\nSet with the use of List: ")
```

```
print(set1)
```

Output: Initial blank Set:

```
set()
```

Set with the use of String:

```
{'P', 'y', 'h', 'o', 't', 'r', 'n', 'F'}
```

Set with the use of an Object:

```
{'P', 'y', 'h', 'o', ' ', 't', 'r', 'n', 'F'}
```

Set with the use of List:

```
{'Python', 'For'}
```

A set contains only unique elements but at the time of set creation, multiple duplicate values can also be passed. Order of elements in a set is undefined and is unchangeable. Type of elements in a set need not be the same, various mixed-up data type values can also be passed to the set.

Creating a Set with a List of Numbers (Having duplicate values)

```
set1 = set([1, 2, 4, 4, 3, 3, 3, 6, 5])  
print("\nSet with the use of Numbers: ")  
print(set1)
```

Creating a Set with a mixed type of values (Having numbers and strings)

```
set1 = set([1, 2, 'Python', 4, 'For', 6, 'Python'])  
print("\nSet with the use of Mixed Values")  
print(set1)
```

Output: Set with the use of Numbers:

```
{1, 2, 3, 4, 5, 6}
```

Set with the use of Mixed Values

```
{1, 2, 4, 6, 'Python', 'For'}
```

Creating a set with another method

Another Method to create sets in Python3

Set containing numbers

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

Output: {1, 2, 3}

2. Adding Elements to a Set

Using add() method

Elements can be added to the Set by using the built-in add() function. Only one element at a time can be added to the set by using add() method, loops are used to add multiple elements at a time with the use of add() method.

Note: Lists cannot be added to a set as elements because Lists are not hashable whereas Tuples can be added because tuples are immutable and hence Hashable.

Python program to demonstrate Addition of elements in a Set

Creating a Set

```
set1 = set()
```

```
print("Initial blank Set: ")
```

```
print(set1)
```

Adding element and tuple to the Set

```
set1.add(8)
```

```
set1.add(9)
```

```
set1.add((6, 7))
```

```
print("\nSet after Addition of Three elements: ")
```

```
print(set1)
```

Adding elements to the Set using Iterator

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

```
    set1.add(i)
```

```
print("\nSet after Addition of elements from 1-5: ")
```

```
print(set1)
```

Output: Initial blank Set:

```
set()
```

Set after Addition of Three elements:

```
{8, 9, (6, 7)}
```

Set after Addition of elements from 1-5:

```
{1, 2, 3, 4, 5, 8, 9, (6, 7)}
```

Using update() method

For the addition of two or more elements **Update()** method is used. The **update()** method accepts lists, strings, tuples as well as other sets as its arguments. In all of these cases, duplicate elements are avoided.

Python program to demonstrate Addition of elements in a Set Addition of
#elements to the Set using Update function

```
set1 = set([4, 5, (6, 7)])
```

```
set1.update([10, 11])
```

```
print("\nSet after Addition of elements using Update: ")
```

```
print(set1)
```

Output: Set after Addition of elements using Update:
{4, 5, 10, 11, (6, 7)}

3. Accessing a Set:

Set items cannot be accessed by referring to an index, since sets are unordered the items has no index. 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.

Python program to demonstrate Accessing of elements in a set

Creating a set

```
set1 = set(["Python", "For", "Python"])
```

```
print("\nInitial set")
```

```
print(set1)
```

Accessing element using for loop

```
print("\nElements of set: ")
```

```
for i in set1:
```

```
    print(i, end=" ")
```

Checking the element using in keyword

```
print("Python " in set1)
```

Output: Initial set

{'Python', ' Python ', 'For'}

Elements of set:

Python Python For False

4. Removing elements from the Set

Using remove() method or discard() method:

Elements can be removed from the Set by using the built-in **remove()** function but a **KeyError** arises if the element doesn't exist in the set. To remove elements from a set without **KeyError**, use **discard()**, if the element doesn't exist in the set, it remains unchanged.

Python program to demonstrate Deletion of elements in a Set

Creating a Set

```
set1 = set([1, 2, 3, 4, 5, 6,  
           7, 8, 9, 10, 11, 12])
```



```

print("Initial Set: ")
print(set1)
# Removing elements from Set using Remove() method
set1.remove(5)
set1.remove(6)
print("\nSet after Removal of two elements: ")
print(set1)
# Removing elements from Set using Discard() method
set1.discard(8)
set1.discard(9)
print("\nSet after Discarding two elements: ")
print(set1)
# Removing elements from Set using iterator method
for i in range(1, 5):
    set1.remove(i)
print("\nSet after Removing a range of elements: ")
print(set1)

```

Output: Initial Set:

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

Set after Removal of two elements:

{1, 2, 3, 4, 7, 8, 9, 10, 11, 12}

Set after Discarding two elements:

{1, 2, 3, 4, 7, 10, 11, 12}

Set after Removing a range of elements:

{7, 10, 11, 12}

Using pop() method:

pop() function can also be used to remove and return an element from the set, but it removes only the last element of the set.

Note: If the set is unordered then there's no such way to determine which element is popped by using the **pop()** function.

Python program to demonstrate Deletion of elements in a Set

Creating a Set

```
set1 = set([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])
```

```

print("Initial Set: ")
print(set1)
# Removing element from the Set using the pop() method
set1.pop()
print("\nSet after popping an element: ")
print(set1)

```

Output: Initial Set:

{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

Set after popping an element:

{2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}

Using clear() method:

To remove all the elements from the set, clear() function is used.

```

#Creating a set
set1 = set([1,2,3,4,5])
print("\n Initial set: ")
print(set1)
# Removing all the elements from Set using clear() method
set1.clear()
print("\nSet after clearing all the elements: ")
print(set1)

```

Output: Initial set:

{1, 2, 3, 4, 5}

Set after clearing all the elements:

set()

Frozen sets in Python are immutable objects that only support methods and operators that produce a result without affecting the frozen set or sets to which they are applied. While elements of a set can be modified at any time, elements of the frozen set remain the same after creation.

If no parameters are passed, it returns an empty frozenset.

Python program to demonstrate working of a FrozenSet

Creating a Set

```
String = ('W', 'e', 'l', 'c', 'o', 'm', 'e')
```

```
Fset1 = frozenset(String)
print("The FrozenSet is: ")
print(Fset1)
# To print Empty Frozen Set No parameter is passed
print("\nEmpty FrozenSet: ")
print(frozenset())
```

Output: The FrozenSet is:
 frozenset({'m', 'W', 'c', 'o', 'e', 'l'})
 Empty FrozenSet:
 frozenset()

5. Typecasting Objects into sets

Typecasting Objects in Python3 into sets

Typecasting list into set

```
my_list = [1, 2, 3, 3, 4, 5, 5, 6, 2]
```

```
my_set = set(my_list)
```

```
print("my_list as a set: ", my_set)
```

Typecasting string into set

```
my_str = "PythonForPython"
```

```
my_set1 = set(my_str)
```

```
print("my_str as a set: ", my_set1)
```

Typecasting dictionary into set

```
my_dict = {1: "One", 2: "Two", 3: "Three"}
```

```
my_set2 = set(my_dict)
```

```
print("my_dict as a set: ", my_set2)
```

Output: my_list as a set: {1, 2, 3, 4, 5, 6}
 my_str as a set: {'P', 'n', 'y', 'h', 't', 'r', 'F', 'o'}
 my_dict as a set: {1, 2, 3}

Example: Implementing all functions:

```
def create_set():
```

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

```
    print(my_set)
```

```
def add_element():
```

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

```
    my_set.add(6)
```

```
    print(my_set)
```

```
def remove_element():
```

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

```
    my_set.remove(3)
```

```
    print(my_set)
```

```
def clear_set():
```

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

```
    my_set.clear()
```

```
    print(my_set)
```

```
def set_union():
```

```
    set1 = {1, 2, 3}
```

```
    set2 = {4, 5, 6}
```

```
    my_set = set1.union(set2)
```

```
    print(my_set)
```

```
def set_intersection():
```

```
    set1 = {1, 2, 3, 4, 5}
```

```
    set2 = {4, 5, 6, 7, 8}
```

```
    my_set = set1.intersection(set2)
```

```
    print(my_set)
```

```

def set_difference():
    set1 = {1, 2, 3, 4, 5}
    set2 = {4, 5, 6, 7, 8}
    my_set = set1.difference(set2)
    print(my_set)

def set_symmetric_difference():
    set1 = {1, 2, 3, 4, 5}
    set2 = {4, 5, 6, 7, 8}
    my_set = set1.symmetric_difference(set2)
    print(my_set)

def set_subset():
    set1 = {1, 2, 3, 4, 5}
    set2 = {2, 3, 4}
    subset = set2.issubset(set1)
    print(subset)

def set_superset():
    set1 = {1, 2, 3, 4, 5}
    set2 = {2, 3, 4}
    superset = set1.issuperset(set2)
    print(superset)

if __name__ == '__main__':
    create_set()
    add_element()
    remove_element()
    clear_set()
    set_union()
    set_intersection()
    set_difference()
    set_symmetric_difference()

```

```
set_subset()
```

```
set_superset()
```

Output: {1, 2, 3, 4, 5}
 {1, 2, 3, 4, 5, 6}
 {1, 2, 4, 5}
 set()
 {1, 2, 3, 4, 5, 6}
 {4, 5}
 {1, 2, 3}
 {1, 2, 3, 6, 7, 8}
 True
 True

Advantages:

- **Unique Elements:** Sets can only contain unique elements, so they can be useful for removing duplicates from a collection of data.
- **Fast Membership Testing:** Sets are optimized for fast membership testing, so they can be useful for determining whether a value is in a collection or not.
- **Mathematical Set Operations:** Sets support mathematical set operations like union, intersection, and difference, which can be useful for working with sets of data.
- **Mutable:** Sets are mutable, which means that you can add or remove elements from a set after it has been created.

Disadvantages:

- **Unordered:** Sets are unordered, which means that you cannot rely on the order of the data in the set. This can make it difficult to access or process data in a specific order.
- **Limited Functionality:** Sets have limited functionality compared to lists, as they do not support methods like `append()` or `pop()`. This can make it more difficult to modify or manipulate data stored in a set.

- **Memory Usage:** Sets can consume more memory than lists, especially for small datasets. This is because each element in a set requires additional memory to store a hash value.
- **Less Commonly Used:** Sets are less commonly used than lists and dictionaries in Python, which means that there may be fewer resources or libraries available for working with them. This can make it more difficult to find solutions to problems or to get help with debugging.

Set Methods

Function	Description
add()	Adds an element to a set
remove()	Removes an element from a set. If the element is not present in the set, raise a <code>KeyError</code>
clear()	Removes all elements from a set
copy()	Returns a shallow copy of a set
pop()	Removes and returns an arbitrary set element. Raise <code>KeyError</code> if the set is empty
update()	Updates a set with the union of itself and others
union()	Returns the union of sets in a new set
difference()	Returns the difference of two or more sets as a new set
difference_update()	Removes all elements of another set from this set
discard()	Removes an element from set if it is a member. (Do nothing if the element is not in set)
intersection()	Returns the intersection of two sets as a new set
intersection_update()	Updates the set with the intersection of itself and another
isdisjoint()	Returns <code>True</code> if two sets have a null intersection
issubset()	Returns <code>True</code> if another set contains this set
issuperset()	Returns <code>True</code> if this set contains another set
symmetric_difference()	Returns the symmetric difference of two sets as a new set

BOOLEAN DATA TYPE IN PYTHON:

Data type with one of the two built-in values, True or False. Boolean objects that are equal to True are truthy (true), and those equal to False are falsy (false). But non-Boolean objects can be evaluated in a Boolean context as well and determined to be true or false. It is denoted by the class bool.

Note – True and False with capital ‘T’ and ‘F’ are valid booleans otherwise python will throw an error.

Python program to demonstrate boolean type

```
print(type(True))
```

```
print(type(False))
```

```
print(type(true))
```

Output: <class 'bool'>

<class 'bool'>

Traceback (most recent call last):

File "G:/Academic/Notes/Python/Programme/BoolTypeDemo.py", line 7, in <module>

```
print(type(true))
```

NameError: name 'true' is not defined. Did you mean: 'True'?

1.7. Understanding python blocks

Whitespace is used for indentation in Python. Unlike many other programming languages which only serve to make the code easier to read, Python indentation is mandatory. One can understand it better by looking at an example of indentation in Python.

Role of Indentation in Python

A block is a combination of all these statements. Block can be regarded as the grouping of statements for a specific purpose. Most programming languages like C, C++, and Java use braces { } to define a block of code for indentation. One of the distinctive roles of Python is its use of indentation to highlight the blocks of code. All statements with the

same distance to the right belong to the same block of code. If a block has to be more deeply nested, it is simply indented further to the right.

Example 1:

```
# Python indentation
site = 'atestc'
if site == 'atestc':
    print('Logging on to 'atestc'...')
else:
    print('retype the URL.')
print('All set !')
```

Output: Logging on to atestc...
 All set !

Example 2:

```
j = 1
while(j <= 5):
    print(j)
    j = j + 1
```

Output: 1
 2
 3
 4
 5

1.8. Python Program Flow Control

There comes situations in real life when we need to make some decisions and based on these decisions, we decide what should we do next. Similar situations arise in programming also where we need to make some decisions and based on these decisions we will execute the next block of code. Decision-making statements in programming languages decide the direction(Control Flow) of the flow of program execution.

Types of Control Flow in Python

In Python programming language, the type of control flow statements are as follows:

1. The if statement
2. The if-else statement

3. The nested-if statement

4. The if-elif-else ladder

if Statement:

If the simple code of block is to be performed if the condition holds true then the if statement is used. Here the condition mentioned holds true then the code of the block runs otherwise not.

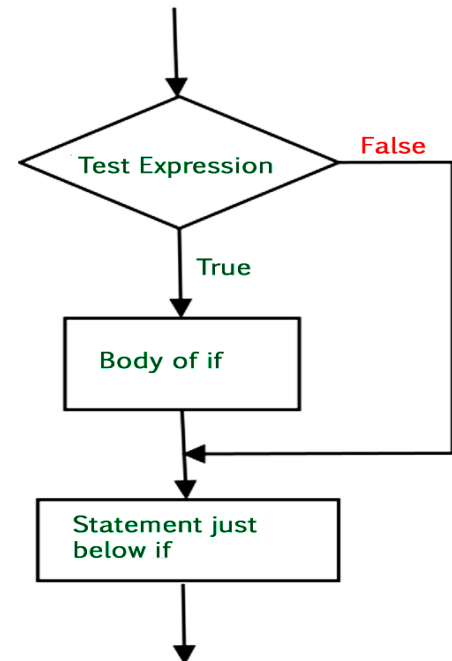
Syntax:

if condition:

Statements to execute if

condition is true

Flowchart:



Example:

```
# if statement example
if 10 > 5:
    print("10 greater than 5")
print("Program ended")
```

Output: 10 greater than 5
 Program ended

Indentation (White space) is used to delimit the block of code. As shown in the above example it is mandatory to use indentation in Python3 coding.

if..else Statement:

In conditional if Statement the additional block of code is merged as else statement which is performed when if condition is false.

Syntax:

if (condition):

Executes this block if

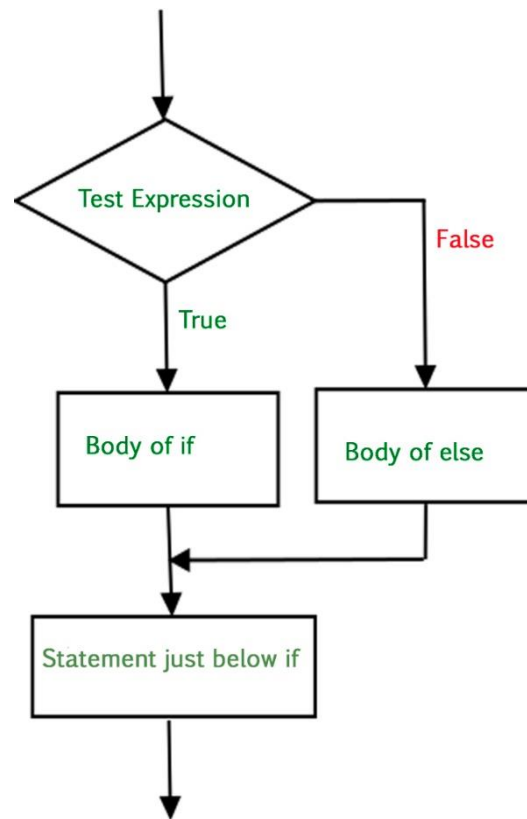
condition is true

else:

Executes this block if

condition is false

Flow Chart:



Example:

if..else statement example

x = 3

if x == 4:

print("Yes")

else:

print("No")

Output: No

Example: You can also chain if..else statement with more than one condition.

if..else chain statement

letter = "A"

if letter == "B":

print("letter is B")

else:

```

if letter == "C":
    print("letter is C")
else:
    if letter == "A":
        print("letter is A")
    else:
        print("letter isn't A, B and C")

```

Output: letter is A

Example of Python if else statement in a list comprehension

Explicit function

```

def digitSum(n):
    dsum = 0
    for ele in str(n):
        dsum += int(ele)
    return dsum
# Initializing list
List = [367, 111, 562, 945, 6726, 873]
# Using the function on odd elements of the list
newList = [digitSum(i) for i in List if i & 1]
# Displaying new list
print(newList)

```

Output: [16, 3, 18, 18]

Nested if Statement:

if statement can also be checked inside other if statement. This conditional statement is called a nested if statement. This means that inner if condition will be checked only if outer if condition is true and by this, we can see multiple conditions to be satisfied.

Syntax:

```

if (condition1):
    # Executes when condition1 is true
    if (condition2):

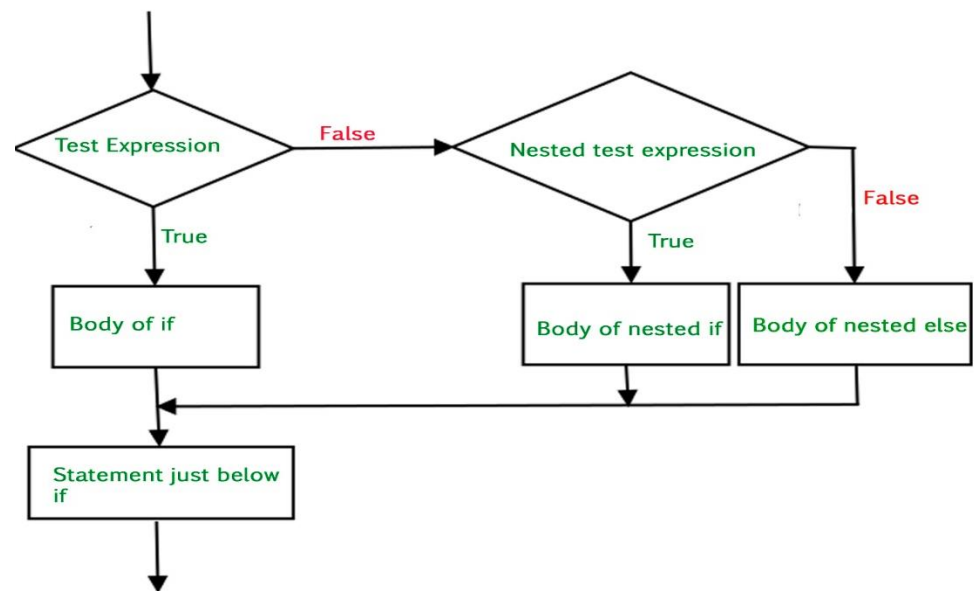
```

Executes when condition2 is true

if Block is end here

if Block is end here

Flow chart:



Example:

Nested if statement example

num = 10

if num > 5:

print("Bigger than 5")

if num <= 15:

print("Between 5 and 15")

Output: Bigger than 5
Between 5 and 15

Example:

python program to illustrate nested If statement

i = 10

if (i == 10):

First if statement

if (i < 15):

print("i is smaller than 15")

Nested - if statement Will only be executed if statement above it is true

```

if (i < 12):
    print("i is smaller than 12 too")
else:
    print("i is greater than 15")

```

Output: i is smaller than 15
 i is smaller than 12 too

Example:

Python program to illustrate short hand if-else

```

i = 10
print(True) if i < 15 else print(False)

```

Output: True

if-elif Statement:

The if-elif statement is shortcut of if..else chain. While using if-elif statement at the end else block is added which is performed if none of the above if-elif statement is true.

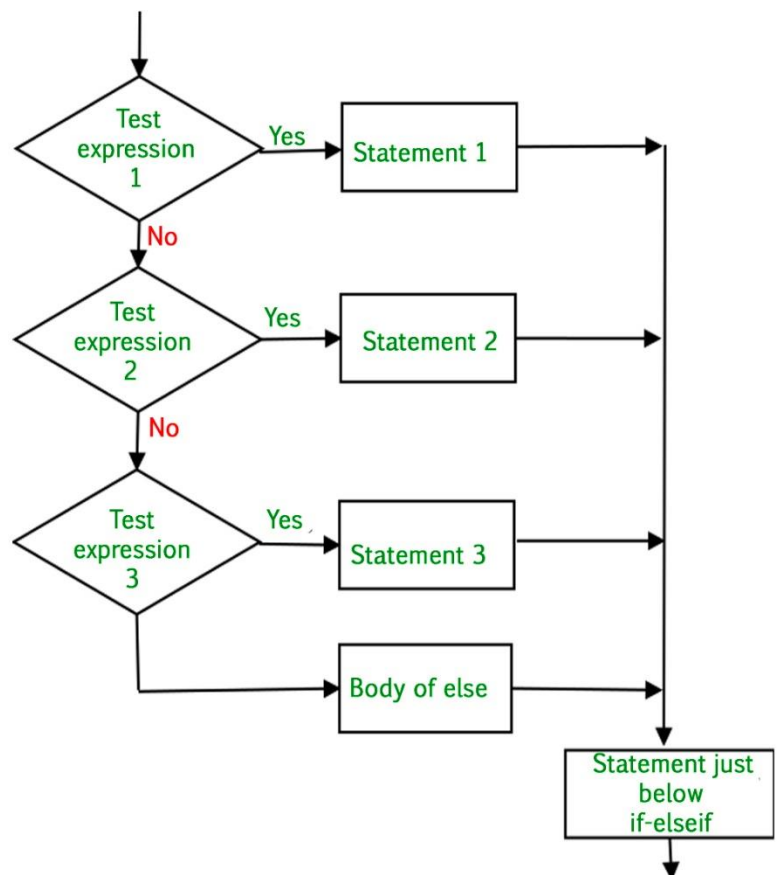
Syntax:

```

if (condition):
    statement
elif (condition):
    statement
.
.
else:
    statement

```

Flow Chart:



Example:

if-elif statement example

```
letter = "A"
if letter == "B":
    print("letter is B")
elif letter == "C":
    print("letter is C")
elif letter == "A":
    print("letter is A")
else:
    print("letter isn't A, B or C")
```

Output: letter is A

1.9. Loops in python

Python programming language provides the following types of loops to handle looping requirements. Python provides three ways for executing the loops. While all the ways provide similar basic functionality, they differ in their syntax and condition-checking time.

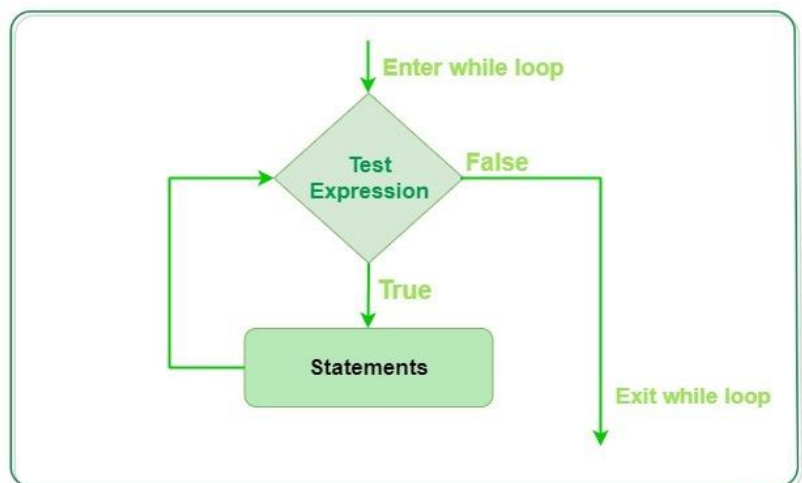
Python While Loop:

Python While Loop is used to execute a block of statements repeatedly until a given condition is satisfied. And when the condition becomes false, the line immediately after the loop in the program is executed.

Syntax:

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

Flowchart:



While loop falls under the category of indefinite iteration. Indefinite iteration means that the number of times the loop is executed isn't specified explicitly in advance.

Statements represent all the statements indented by the same number of character spaces after a programming construct are considered to be part of a single block of code. Python uses indentation as its method of grouping statements. When a while loop is executed, expr is first evaluated in a Boolean context and if it is true, the loop body is executed. Then the expr is checked again, if it is still true then the body is executed again and this continues until the expression becomes false.

Example: Python While Loop

```
# Python program to illustrate while loop
```

```
count = 0
```

```
while (count < 3):
```

```
    count = count + 1
```

```
    print("Hello Python")
```

```
Output:      Hello Python
```

```
            Hello Python
```

```
            Hello Python
```

In the above example, the condition for while will be True as long as the counter variable (count) is less than 3.

Example : Python while loop with list

```
# checks if list still contains any element
```

```
a = [1, 2, 3, 4]
```

```
while a:
```

```
    print(a.pop())
```

```
Output:      4
```

```
            3
```

```
            2
```

```
            1
```

In the above example, we have run a while loop over a list that will run until there is an element present in the list.

Example: Single statement while block

Just like the if block, if the while block consists of a single statement we can declare the entire loop in a single line. If there are multiple statements in the block that makes up the loop body, they can be separated by semicolons (;).

Python program to illustrate Single statement while block

```
count = 0
```

```
while (count < 5): count += 1; print("Hello Python")
```

Output: Hello Python
 Hello Python
 Hello Python
 Hello Python
 Hello Python

Example 4: Loop Control Statements

Loop control statements change execution from its normal sequence. When execution leaves a scope, all automatic objects that were created in that scope are destroyed. Python supports the following control statements.

Continue Statement:

Python Continue Statement returns the control to the beginning of the loop.

Example: Python while loop with continue statement

Prints all letters except 'e' and 's'

```
i = 0
```

```
a = 'pythonforpython'
```

```
while i < len(a):
```

```
    if a[i] == 'e' or a[i] == 's':
```

```
        i += 1
```

```
        continue
```

```
    print('Current Letter :', a[i])
```

```
    i += 1
```

Output: Current Letter : p
 Current Letter : y
 Current Letter : t
 Current Letter : h

Current Letter : o
Current Letter : n
Current Letter: f
Current Letter: o
Current Letter: r
Current Letter: p
Current Letter: y
Current Letter: t
Current Letter: h
Current Letter: o
Current Letter: n

Break Statement:

Python Break Statement brings control out of the loop.

Example: Python while loop with a break statement

break the loop as soon it sees 'e' or 's'

```
i = 0
a = 'pythonforpython'
while i < len(a):
    if a[i] == 'e' or a[i] == 's':
        i += 1
        break
    print('Current Letter :', a[i])
    i += 1
```

Output: Current Letter: p

Example:

```
# Initialize a counter
count = 0
# Loop infinitely
while True:
```

```

# Increment the counter
count += 1
print(f"Count is {count}")
# Check if the counter has reached a certain value
if count == 10:
    # If so, exit the loop
    break
# This will be executed after the loop exits
print("The loop has ended.")

```

Output:

```

Count is 1
Count is 2
Count is 3
Count is 4
Count is 5
Count is 6
Count is 7
Count is 8
Count is 9
Count is 10
The loop has ended.

```

Pass Statement:

The Python pass statement to write empty loops. Pass is also used for empty control statements, functions, and classes.

Example: Python while loop with a pass statement

```

# An empty loop
a = 'pythonforpython'
i = 0
while i < len(a):
    i += 1
    pass
print('Value of i:', i)

```

Output: Value of i: 13

Python For Loops :

Python For loop is used for sequential traversal i.e. it is used for iterating over an iterable like String, Tuple, List, Set or Dictionary.

In Python, there is no C style for loop, i.e., for (i=0; i<n; i++). There is “for” loop which is similar to each loop in other languages. Let us learn how to use for in loop for sequential traversals.

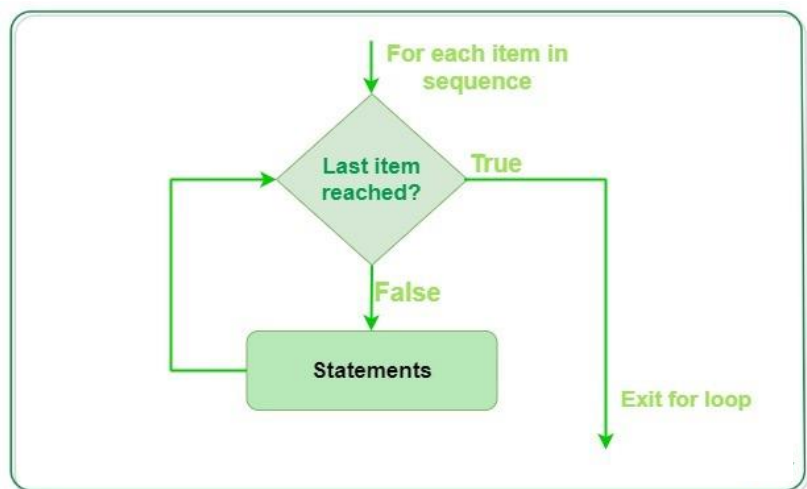
Note: In Python, for loops only implements the collection-based iteration.

For Loops Syntax:

for var in iterable:

 # statements

Flowchart of for loop



Here the iterable is a collection of objects like lists, tuples. The indented statements inside the for loops are executed once for each item in an iterable. The variable var takes the value of the next item of the iterable each time through the loop.

Example 1: Using For Loops in Python List

Python program to illustrate Iterating over a list

```
l = ["Python", "For", "Python"]
```

```
for i in l:
```

```
    print(i)
```

Output: Python
 For
 Python

Example 2: Using For Loops in Python Dictionary

```
# Iterating over dictionary
print("Dictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d:
    print("% s % d" % (i, d[i]))
```

Output: Dictionary Iteration
 xyz 123
 abc 345

Example 3: Using For Loops in Python String

```
# Iterating over a String
print("String Iteration")
s = "Python"
for i in s:
    print(i)
```

Output: String Iteration
 P
 y
 t
 h
 o
 n

Example with List, Tuple, string, and dictionary iteration using For Loops in Python

```
# Python program to illustrate Iterating over a list
print("List Iteration")
l = ["Python", "for", "Python"]
for i in l:
    print(i)
# Iterating over a tuple (immutable)
```

```

print("\nTuple Iteration")
t = ("Python", "for", "Python")
for i in t:
    print(i)
# Iterating over a String
print("\nString Iteration")
s = "Python"
for i in s:
    print(i)
# Iterating over dictionary
print("\nDictionary Iteration")
d = dict()
d['xyz'] = 123
d['abc'] = 345
for i in d:
    print("%s %d" % (i, d[i]))
# Iterating over a set
print("\nSet Iteration")
set1 = {1, 2, 3, 4, 5, 6}
for i in set1:
    print(i),

```

Output: List Iteration
 Python
 for
 Python
 Tuple Iteration
 Python
 for
 Python
 String Iteration
 P
 y
 t

```
h
o
n
Dictionary Iteration
xyz 123
abc 345
Set Iteration
1
2
3
4
5
6
```

Python range() function

The Python range() function returns a sequence of numbers, in a given range. The most common use of it is to iterate sequence on a sequence of numbers using Python loops.

Syntax:

range(start, stop, step)

Parameter:

start: [optional] start value of the sequence

stop: next value after the end value of the sequence

step: [optional] integer value, denoting the difference between any two numbers in the sequence.

Return: Returns a range type object.

Example of Python range() function

```
# print first 5 integers
# using python range() function
for i in range(5):
    print(i, end=" ")
print()
```

Output: 0 1 2 3 4

What is the use of the range function in Python

In simple terms, range() allows the user to generate a series of numbers within a given range. Depending on how many arguments the user is passing to the function, the user can decide where that series of numbers will begin and end, as well as how big the difference will be between one number and the next. Python range() function takes can be initialized in 3 ways.

range (stop) takes one argument.

range (start, stop) takes two arguments.

range (start, stop, step) takes three arguments.

Example: Demonstration of Python range (stop)

```
# printing first 6 whole number
for i in range(6):
    print(i, end=" ")
print()
```

Output: 0 1 2 3 4 5

Example: Demonstration of Python range (start, stop)

```
# printing a natural number from 5 to 20
for i in range(5, 20):
    print(i, end=" ")
print()
```

Output: 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19

Example: Demonstration of Python range (start, stop, step)

```
for i in range(0, 10, 2):
    print(i, end=" ")
print()
```

Output: 0 2 4 6 8

```
# incremented by 4
for i in range(0, 30, 4):
    print(i, end=" ")
print()
```

Output: 0 4 8 12 16 20 24 28

Python range() using negative step

```
# incremented by -2
for i in range(25, 2, -2):
    print(i, end=" ")
print()
```

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

Some Important points to remember about the Python range() function:

- range() function only works with the integers, i.e. whole numbers.
- All arguments must be integers. Users can not pass a string or float number or any other type in a start, stop and step argument of a range().
- All three arguments can be positive or negative.
- The step value must not be zero. If a step is zero, python raises a ValueError exception.
- range() is a type in Python
- Users can access items in a range() by index, just as users do with a list:

1.10. Loop manipulation using pass, continue, break and else

Python Continue Statement:

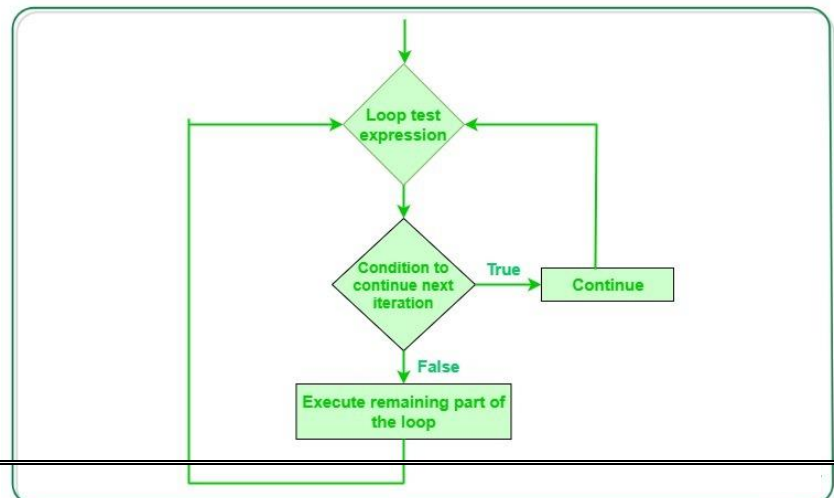
Python Continue Statement skips the execution of the program block from after the continue statement and forces the control to start the next iteration.

Python Continue statement is a loop control statement that forces to execute the next iteration of the loop while skipping the rest of the code inside the loop for the current iteration only, i.e. when the continue statement is executed in the loop, the code inside the loop following the continue statement will be skipped for the current iteration and the next iteration of the loop will begin.

Syntax:

```
while True:
    ...
    if x == 10:
        continue
    print(x)
```

Flowchart:



Example 1: Demonstration of Continue statement in Python

for var **in** "Welcome to Python":

if var == "e":

continue

print(var)

Output: W

 l

 c

 o

 m

 t

 o

 P

 y

 t

 h

 o

 n

Example 2: Printing range with Python Continue Statement

loop from 1 to 10

for i **in** **range**(1, 11):

 # If i is equals to 6, continue to next iteration without printing

if i == 6:

continue

else:

 # otherwise print the value

 # of i

print(i, end=" ")

Output: 1 2 3 4 5 7 8 9 10

Python pass Statement:

The Python pass statement is a null statement. But the difference between pass and comment is that comment is ignored by the interpreter whereas pass is not ignored.

Syntax:

pass

What is pass statement in Python?

When the user does not know what code to write, So user simply places a pass at that line. Sometimes, the pass is used when the user doesn't want any code to execute. So users can simply place a pass where empty code is not allowed, like in loops, function definitions, class definitions, or in if statements. So using a pass statement user avoids this error.

Why Python Needs “pass” Statement?

If we do not use pass or simply enter a comment or a blank here, we will receive an IndentationError error message.

```
n = 26
if n > 26:
    # write code your here
print('Python')
```

Output: IndentationError: expected an indented block after 'if' statement

Use of pass keyword in Function

Python Pass keyword can be used in empty functions.

```
def function:
    pass
```

Use of pass keyword in Python Loop

The pass keyword can also be used in an empty class in Python.

```
class pythonClass:
    pass
```

Use of pass keyword in Python Loop

The pass keyword can be used in Python for loop, when a user doesn't know what to code inside the loop in Python.

```
n = 10
for i in range(n):
    # pass can be used as placeholder when code is to added later
    pass
```

Use of pass keyword in Conditional statement

Python pass keyword can be used with conditional statements.

```
a = 10
b = 20
if(a<b):
    pass
else:
    print("b<a")
```

the pass statement gets executed when the condition is true.

```
li=['a', 'b', 'c', 'd']

for i in li:
    if(i=='a'):
        pass
    else:
        print(i)
```

Output: b
 c
 d

Python break statement:

Python break is used to terminate the execution of the loop.

Syntax:

```
Loop{
```

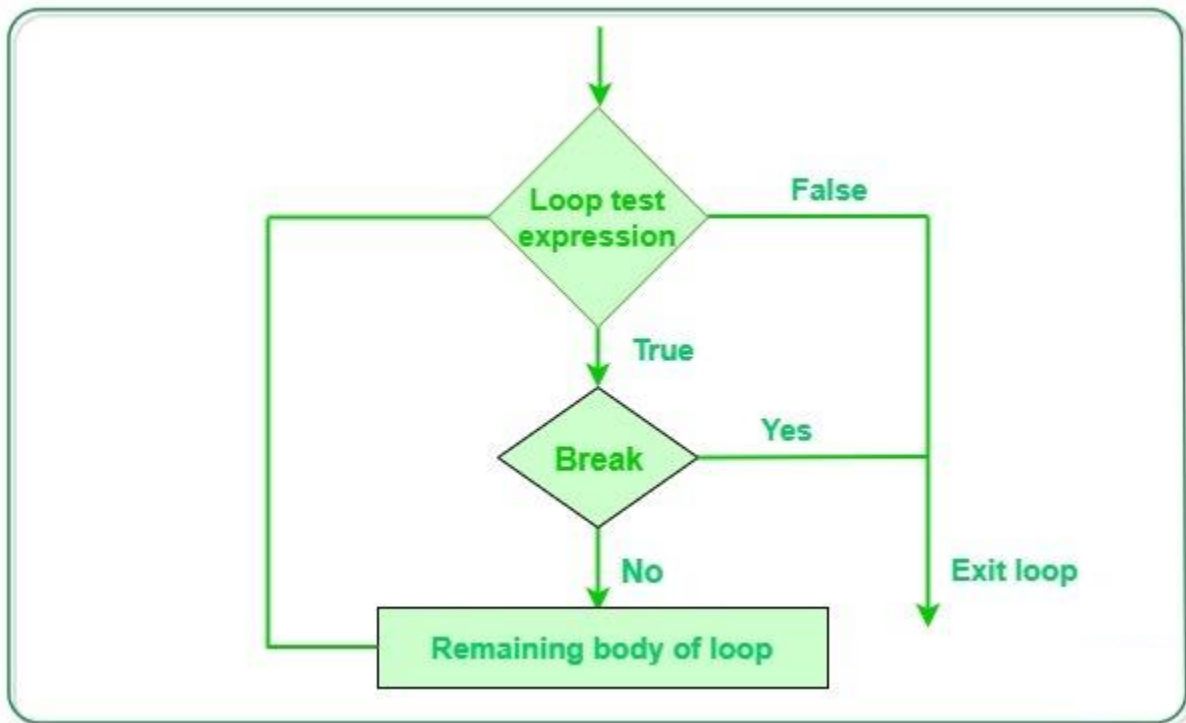
Condition:

break

}

Python break statement

break statement in Python is used to bring the control out of the loop when some external condition is triggered. break statement is put inside the loop body (generally after if condition). It terminates the current loop, i.e., the loop in which it appears, and resumes execution at the next statement immediately after the end of that loop. If the break statement is inside a nested loop, the break will terminate the innermost loop.



Example 1:

```
for i in range(10):  
    print(i)  
    if i == 2:  
        break
```

Output: 0
 1
 2

Example 2:

```
# Python program to demonstrate break statement
```

```
s = 'Welcome Python'
```

```
# Using for loop
```

```
for letter in s:
```

```
    print(letter)
```

```
    # break the loop as soon it sees 'e'
```

```
    # or 's'
```

```
    if letter == 'e' or letter == 's':
```

```
        break
```

```
print("Out of for loop" )
```

```
print()
```

```
i = 0
```

```
# Using while loop
```

```
while True:
```

```
    print(s[i])
```

```
    # break the loop as soon it sees 'e' or 's'
```

```
    if s[i] == 'e' or s[i] == 's':
```

```
        break
```

```
    i += 1
```

```
print("Out of while loop ")
```

Output: W
 e
 Out of for loop

 W
 e
 Out of while loop

Example 3:

```
num = 0
```

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

```
    num += 1
```

```
    if num == 8:
```

```
break
```

```
print("The num has value:", num)
```

```
print("Out of loop")
```

Output: The num has value: 1
 The num has value: 2
 The num has value: 3
 The num has value: 4
 The num has value: 5
 The num has value: 6
 The num has value: 7
 Out of loop

Extra Reading: Python installation with windows, Linux and MAC OS, creating virtual environment, configuring python on EC2 instance, understanding python IDE –[VSCode, PyCharm, Spyder], Installing Anaconda and setting up environment for python