



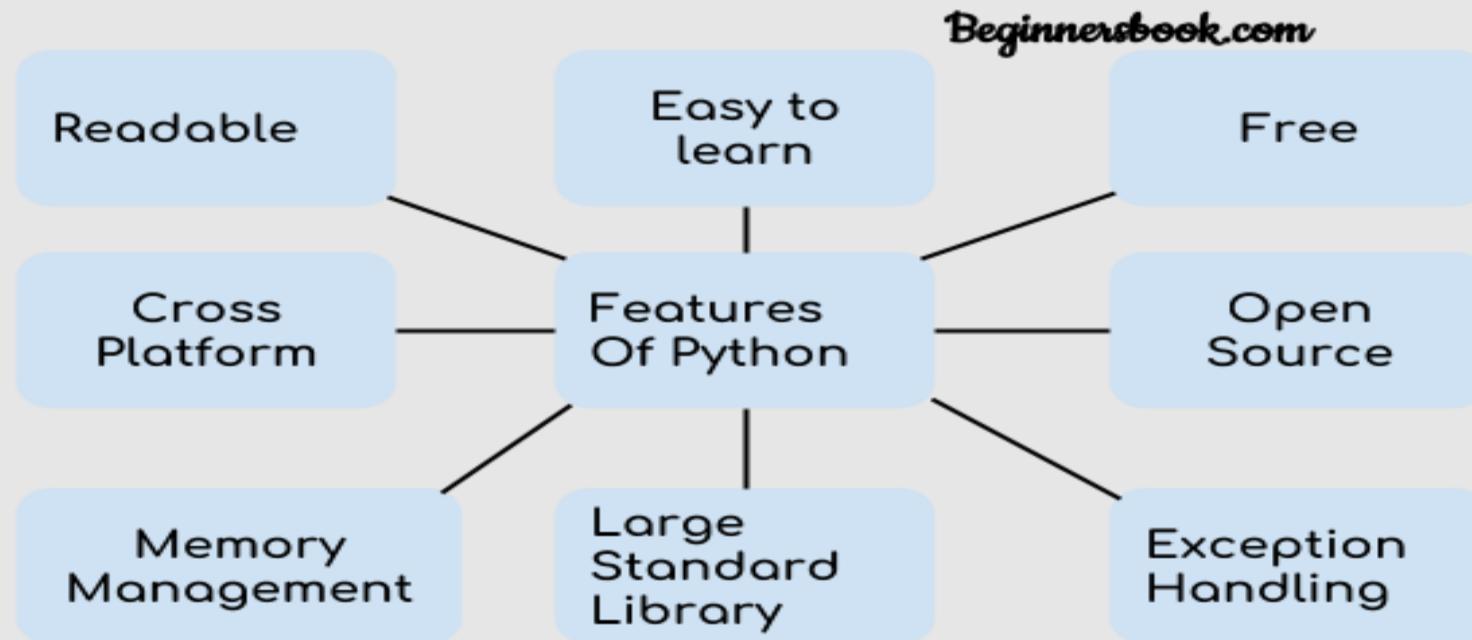
# INTRODUCTION TO PYTHON

Unit I

# Introduction

- **Python** is a high-level, interpreted, interactive and object-oriented scripting language.
- Python is designed to be highly readable.
- Python is a popular programming language.
- It was created by Guido van Rossum and released in 1991.
- It is used for:
  - web development (server-side),
  - software development,
  - mathematics,
  - system scripting.

# Introduction



Reference : <https://beginnersbook.com/2018/01/introduction-to-python-programming/>

# Features of Python

- 1. **Readable:** Python is a very readable language.
- 2. **Easy to Learn:** Learning python is easy as this is a expressive and high level programming language, which means it is easy to understand the language and thus easy to learn.
- 3. **Cross platform:** Python is available and can run on various operating systems such as Mac, Windows, Linux, Unix etc. This makes it a cross platform and portable language.
- 4. **Open Source:** Python is a open source programming language.
- 5. **Large standard library:** Python comes with a large standard library that has some handy codes and functions which we can use while writing code in Python.
- 6. **Free:** Python is free to download and use. This means you can download it for free and use it in your application. See: [Open Source Python License](#). Python is an example of a FLOSS (Free/Libre Open Source Software), which means you can freely distribute copies of this software, read its source code and modify it.

# Features of Python

- 7. **Supports exception handling:** If you are new, you may wonder what is an exception? An exception is an event that can occur during program execution and can disrupt the normal flow of program. Python supports exception handling which means we can write less error prone code and can test various scenarios that can cause an exception later on.
- 8. **Advanced features:** Supports generators and list comprehensions. We will cover these features later.
- 9. **Automatic memory management:** Python supports automatic memory management which means the memory is cleared and freed automatically. You do not have to bother clearing the memory.

# **Applications of Python**

- 1. Web development – Web framework like Django and Flask are based on Python. They help you write server-side code which helps you manage database, write backend programming logic, mapping urls etc.
- 2. Machine learning – There are various machine learning applications written in Python. For example, products recommendation in websites like Amazon, Flipkart, eBay etc. is a machine learning algorithm that recognizes user's interest.
- Face recognition and Voice recognition in the phone is another example of machine learning.
- 3. Data Analysis – Data analysis and data visualization in form of charts can also be developed using Python.

# Applications of Python

- 4. Scripting – Scripting is writing small programs to automate simple tasks such as sending automated response emails etc.
- Such type of applications can also be written in Python programming language.
  
- 5. Game development using Python.
- 6. With Python one can develop various **Embedded applications**.
- 7. Desktop applications – You can develop desktop application in Python using library like TKinter or QT.

# What Are IDEs and Code Editors?

- An IDE (or Integrated Development Environment) is a program dedicated to software development. As the name implies, IDEs integrate several tools specifically designed for software development.
- These tools usually include:
  - An editor designed to handle code (with, for example, syntax highlighting and auto-completion)
  - Build, execution, and debugging tools
  - Some form of source control

# What Are IDEs and Code Editors?

- Most IDEs support many different programming languages and contain many more features. They can, therefore, be large and take time to download and install.
- In contrast, a dedicated code editor can be as simple as a text editor with syntax highlighting and code formatting capabilities.
- The very best ones interact with source control systems as well.
- Compared to an IDE, a good dedicated code editor is usually smaller and quicker, but often less feature rich.

# General Editors and IDEs with Python Support

- **Eclipse + PyDev**
- Category: IDE
- Website: [www.eclipse.org](http://www.eclipse.org)
- Python tools: PyDev, [www.pydev.org](http://www.pydev.org)
- If you've spent any amount of time in the open-source community, you've heard about Eclipse. Available for Linux, Windows, and OS X at, Eclipse is the de-facto open-source IDE for Java development. It has a rich marketplace of extensions and add-ons, which makes Eclipse useful for a wide range of development activities.
- One such extension is PyDev, which enables Python debugging, code completion, and an interactive Python console. Installing PyDev into Eclipse is easy: from Eclipse, select Help, Eclipse Marketplace, then search for PyDev. Click Install and restart Eclipse if necessary.

# General Editors and IDEs with Python Support

- **Sublime Text**
- Category: Code Editor
- Website: <http://www.sublimetext.com>
- Written by a Google engineer with a dream for a better text editor, Sublime Text is an extremely popular code editor. Supported on all platforms, Sublime Text has built-in support for Python code editing and a rich set of extensions (called packages) that extend the syntax and editing features.
- Installing additional Python packages can be tricky: all Sublime Text packages are written in Python itself, and installing community packages often requires you to execute Python scripts directly in Sublime Text.

# General Editors and IDEs with Python Support

- **Atom**
- Category: Code Editor
- Website: <https://atom.io/>
- Available on all platforms, Atom is billed as the “hackable text editor for the 21st Century.” With a sleek interface, file system browser, and marketplace for extensions, open-source Atom is built using Electron, a framework for creating desktop applications using JavaScript, HTML, and CSS. Python language support is provided by an extension that can be installed when Atom is running.

# General Editors and IDEs with Python Support

- **GNU Emacs**
- Category: Code Editor
- Website: <https://www.gnu.org/software/emacs/>
- Back before the iPhone vs Android war, before the Linux vs Windows war, even before the PC vs Mac war, there was the Editor War, with GNU Emacs as one of the combatants. Billed as “the extensible, customizable, self-documenting, real-time display editor,” GNU Emacs has been around almost as long as UNIX and has a fervent following.
- Always free and available on every platform (in one form or another), GNU Emacs uses a form of the powerful Lisp programming language for customization, and various customization scripts exist for Python development.

# General Editors and IDEs with Python Support

- **Vi / Vim**
- Category: Code Editor
- Website: <https://www.vim.org/>
- On the other side of the Text Editor War stands VI (aka VIM). Included by default on almost every UNIX system and Mac OS X, VI has an equally fervent following.
- VI and VIM are modal editors, separating the viewing of a file from the editing of a file. VIM includes many improvements on the original VI, including an extensibility model and in-place code building. VIMScripts are available for various Python development tasks.

# General Editors and IDEs with Python Support

- **Visual Studio**
- Category: IDE
- Website: <https://www.visualstudio.com/vs/>
- Python tools: Python Tools for Visual Studio, aka PTVS
- Built by Microsoft, Visual Studio is a full-featured IDE, in many ways comparable to Eclipse. Built for Windows and Mac OS only, VS comes in both free (Community) and paid (Professional and Enterprise) versions. Visual Studio enables development for a variety of platforms and comes with its own marketplace for extensions.
- Python Tools for Visual Studio (aka PTVS) enables Python coding in Visual Studio, as well as Intellisense for Python, debugging, and other tools.

# General Editors and IDEs with Python Support

- **Visual Studio Code**
- Category: Code Editor
- Website: <https://code.visualstudio.com/>
- Python tools: <https://marketplace.visualstudio.com/items?itemName=ms-python.python>
- Not to be confused with full Visual Studio, Visual Studio Code (aka VS Code) is a full-featured code editor available for Linux, Mac OS X, and Windows platforms. Small and light-weight, but full-featured, VS Code is open-source, extensible, and configurable for almost any task. Like Atom, VS Code is built on Electron, so it has the same advantages and disadvantages that brings.
- Installing Python support in VS Code is very accessible: the Marketplace is a quick button click away. Search for Python, click Install, and restart if necessary. VS Code will recognize your Python installation and libraries automatically.

# Python-Specific Editors and IDEs

- **PyCharm**
- Category: IDE
- Website: <https://www.jetbrains.com/pycharm/>
- One of the best (and only) full-featured, dedicated IDEs for Python is PyCharm. Available in both paid (Professional) and free open-source (Community) editions, PyCharm installs quickly and easily on Windows, Mac OS X, and Linux platforms.
- Out of the box, PyCharm supports Python development directly. You can just open a new file and start writing code. You can run and debug Python directly inside PyCharm, and it has support for source control and projects.

# Python-Specific Editors and IDEs

- **Spyder**
- Category: IDE
- Website: <https://github.com/spyder-ide/spyder>
- Spyder is an open-source Python IDE that's optimized for data science workflows. Spyder comes included with the Anaconda package manager distribution, so depending on your setup you may already have it installed on your machine.
- What's interesting about Spyder is that its target audience is data scientists using Python. You'll notice this throughout. For example, Spyder integrates well with common Python data science libraries like SciPy, NumPy, and Matplotlib.
- Spyder features most of the "common IDE features" you might expect, such as a code editor with robust syntax highlighting, Python code completion, and even an integrated documentation browser.

# Python-Specific Editors and IDEs

- **Spyder**
- Category: IDE
- Website: <https://github.com/spyder-ide/spyder>
- A special feature in Spyder is “variable explorer” that allows you to display data using a table-based layout right inside your IDE.
- If you regularly do data science work using Python, this is a unique feature. The IPython/Jupyter integration is nice as well.
- Overall, Spyder feels more basic than other IDEs but for more experienced programmers.

# Python-Specific Editors and IDEs

- **Thonny**
- Category: IDE
- Website: <http://thonny.org/>
- A recent addition to the Python IDE family, Thonny is billed as an IDE for beginners. Written and maintained by the Institute of Computer Science at the University of Tartu in Estonia, Thonny is available for all major platforms, with installation instructions on the site.
- By default, Thonny installs with its own bundled version of Python, so you don't need to install anything else new. More experienced users may need to tweak this setting so already installed libraries are found and used.

# Python-Specific Editors and IDEs

- **Kite**
  - Kite is IDE for Python that automatically completes multiple line codes. This editor supports more than 16 languages. It helps you to code faster with no hassle.
  - Features:
    - It offers Python documentation.
    - This editor provides a function signature as you type.
    - You will get a tooltip on mouse hover.
    - Provides support in email.
    - Uses machine learning models for Python language.

# Python-Specific Editors and IDEs

- **IDLE**
- IDLE (Integrated Development and Learning Environment) is a default editor that comes with Python. It is one of the best Python IDE software which helps a beginner to learn Python easily. IDLE software package is optional for many Linux distributions. The tool can be used on Windows, macOS, and Unix.
- Features:
  - Search multiple files
  - It has an interactive interpreter with colorizing of input, output, and error messages.
  - Supports smart indent, undo, call tips, and auto-completion.
  - Enable you to search and replace within any window.
- Download Link: <https://docs.python.org/3/library/idle.html>

# The Python programming language

- Python is a high-level language.
- Python uses both processes of compiling and interpreting.
- There are two ways to use the Python interpreter: shell mode and script mode.  
In shell mode, you type Python expressions into the Python shell, and the interpreter immediately shows the result.
- One can write a program in a file and use the interpreter to execute the contents of the file. Such a file is called a script.

# Reserved Words

- The following list shows the Python keywords. These are reserved words, and you cannot use them as constant or variable or any other identifier names. All the Python keywords contain lowercase letters only.

and	assert	break	class	continue	def	if	return
exec	finally	for	from	global	in	except	yield
not	or	pass	print	raise	while	lambda	
del	import	try	elif	else	is	with	

# The Python programming language

- Comments :
- Comments increase the readability of the code.
- In python, comments are given by preceding the statement with #.
- Python does not really have a syntax for multi line comments, but one can insert a # for each line.

# The Python programming language

- Indentation :
  - Indentation refers to the spaces at the beginning of a code line.
  - Python uses indentation to indicate a block of code.
  - Python will give you an error if you skip the indentation.

# The Python programming language

- Multi-Line Statements:
- Statements in Python typically end with a new line. Python does, however, allow the use of the line continuation character (\) to denote that the line should continue. For example –
- `total = item_one + \`
  - `item_two + \`
  - `item_three`
- Statements contained within the [], {}, or () brackets do not need to use the line continuation character. For example –
- `days = ['Monday', 'Tuesday', 'Wednesday',`
  - `'Thursday', 'Friday']`

# The Python programming language

- Quotation in Python:
- Python accepts single ('), double ("") and triple (" " or """") quotes to denote string literals, as long as the same type of quote starts and ends the string.
- The triple quotes are used to span the string across multiple lines. For example, all the following are legal –
  - word = 'word'
  - sentence = "This is a sentence."
  - paragraph = """This is a paragraph. It is  
made up of multiple lines and sentences."""

# The Python programming language

- **Variables :**
- Variables are the containers to store the data values.
- Python has no command to declare a variable, it is created when one assigns a value to it for the first time.
- The type of the variable can change even after it is set.
- E.g. `x = 3 // x is of type int here`
- `x= "Ant" // x is of type string here`
  
- Variables are not declared with a specific type.
- If you want to specify the data type of the variable, you can use type cast.
  - For e.g. `X = str(5)` so x will be '5'
  - `y = int(5)` so y will be 5
  - `z = float(5)` so z will be 5.0

# The Python programming language

- **Variables :**
- Python allows you to assign a single value to several variables simultaneously. For example –
  - `a = b = c = 1`
  - Here, an integer object is created with the value 1, and all three variables are assigned to the same memory location. You can also assign multiple objects to multiple variables. For example –
    - `a,b,c = 4,5,"XYZ"`

# Python Data Types

- Variables can hold values, and every value has a data-type.
- Python is a dynamically typed language; hence we do not need to define the type of the variable while declaring it.
- The interpreter implicitly binds the value with its type.
  - **a = 5**
- The variable **a** holds integer value five and we did not define its type.
- Python interpreter will automatically interpret variable **a** as an integer type.
- Python enables us to check the type of the variable used in the program.
- Python provides us the **type()** function, which returns the type of the variable passed.

# Python Data Types

- a=10
- b="Hi Python"
- c = 10.5
- print(type(a))
- print(type(b))
- print(type(c))
- del(c)

- **Output:**

- <type 'int'>
- <type 'str'>
- <type 'float'>

# Python Data Types

- Variable names are case-sensitive.
- String variables can be declared either by using single or double quotes.
- **Rules for Python variables:**
  - A variable name must start with a letter or the underscore character
  - A variable name cannot start with a number
  - A variable name can only contain alpha-numeric characters and underscores (A-z, 0-9, and \_ )
  - Variable names are case-sensitive (age, Age and AGE are three different variables)

## Summary :

- Variables are referred to "envelop" or "buckets" where information can be maintained and referenced. Like any other programming language Python also uses a variable to store the information.
- Variables can be declared by any name or even alphabets like a, aa, abc, etc.
- Variables can be re-declared even after you have declared them for once
- In Python you cannot concatenate string with number directly, you need to declare them as a separate variable, and after that, you can concatenate number with string
- Python constants can be understood as types of variables that hold the value which can not be changed. Usually, Python constants are referenced from other files. Python defined constant is declared in a new or separate file which contains functions, modules, etc.
- Types of variables in Python or Python variable types : Local & Global
- Declare local variable when you want to use it for current function
- Declare Global variable when you want to use the same variable for rest of the program
- To delete a variable, it uses keyword "del".

## Standard data types

- A variable can hold different types of values. For example, a person's name must be stored as a string whereas it's id must be stored as an integer.
- Python has five standard data types –
  - Numbers
  - String
  - List
  - Tuple
  - Dictionary

## Standard data types

- Number data types store numeric values. Number objects are created when you assign a value to them. For example –
  - `var1 = 1`
  - `var2 = 10`

## Numbers

- Number stores numeric values.
- The integer, long, float, and complex values belong to a Python Numbers data-type.
- Python provides the **type()** function to know the data-type of the variable.
- Similarly, the **isinstance()** function is used to check an object belongs to a particular class.
- Python creates Number objects when a number is assigned to a variable. For example;

# Numbers

- `a = 8`
- `print("The type of a", type(a))`
- 
- `b = 23.2`
- `print("The type of b", type(b))`
- 
- `c = 1+3j`
- `print("The type of c", type(c))`
- `print(" c is a complex number", isinstance(1+3j,complex))`
- **Output:**
  - The type of a <class 'int'>
  - The type of b <class 'float'>
  - The type of c <class 'complex'>
  - c is complex number: True

## Numbers

- Python supports three types of numeric data.
- **Int** - Integer value can be any length such as integers 3, 5, 21, -30, -450 etc. Python has no restriction on the length of an integer. Its value belongs to int
- **Float** - Float is used to store floating-point numbers like 4.2, 6.802, 11.2, etc. It is accurate upto 15 decimal points.
- **complex** - A complex number contains an ordered pair, i.e.,  $x + iy$  where  $x$  and  $y$  denote the real and imaginary parts, respectively. The complex numbers like 1.14j, 5.0 + 7.3j, etc.

# Sequence Type

- **String**
- The string can be defined as the sequence of characters represented in the quotation marks. In Python, we can use single, double, or triple quotes to define a string.
- String handling in Python is a straightforward task since Python provides built-in functions and operators to perform operations in the string.
- In the case of string handling, the operator + is used to concatenate two strings as the operation "hello"+ " python" returns "hello python".
- The operator \* is known as a repetition operator as the operation "Python" \*2 returns 'Python Python'.

## Sequence Type

- str = "string using double quotes"

- print(str)

- s = "A multiline"

- print(s)

- Output:

- string using double quotes

- A multiline

## Sequence Type

- Strings in Python are identified as a contiguous set of characters represented in the quotation marks. Python allows for either pairs of single or double quotes. Subsets of strings can be taken using the slice operator ([ ] and [:] ) with indexes starting at 0 in the beginning of the string and working their way from -1 at the end.
- The plus (+) sign is the string concatenation operator and the asterisk (\*) is the repetition operator.

## Sequence Type

- For example –
- str = 'Hello World!'
- print str # Prints complete string
- print str[0] # Prints first character of the string
- print str[2:5] # Prints characters starting from 3rd to 5th
- print str[2:] # Prints string starting from 3rd character
- print str \* 2 # Prints string two times
- print str + "Concat" # Prints concatenated string

## Python Lists

- Lists are the most versatile of Python's compound data types. A list contains items separated by commas and enclosed within square brackets ([]).
  - To some extent, lists are similar to arrays in C.
  - One difference between them is that all the items belonging to a list can be of different data type.
- 
- The values stored in a list can be accessed using the slice operator ([ ] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.
  - The plus (+) sign is the list concatenation operator, and the asterisk (\*) is the repetition operator.

## Python Lists

- `list = [ 'xyz', 456 , 2.23, 'rama', 50.2 ]`
- `otherlist = [123, 'rama']`
  
- `print list` # Prints complete list
- `print list[0]` # Prints first element of the list
- `print list[1:3]` # Prints elements starting from 2nd till 3rd
- `print list[2:]` # Prints elements starting from 3rd element
- `print otherlist * 2` # Prints list two times
- `print list + otherlist` # Prints concatenated lists

## Python Lists

- Gives this output :
- ['xyz', 456, 2.23, 'rama', 50.2]
- xyz
- [456, 2.23]
- [2.23, 'rama', 50.2]
- [123, 'rama', 123, 'rama']
- ['xyz', 456, 2.23, 'rama', 50.2, 123, 'rama']

## Python Tuples

- A tuple is another sequence data type that is like the list. A tuple consists of several values separated by commas. Unlike lists, however, tuples are enclosed within parentheses.
- The main differences between lists and tuples are:
- Lists are enclosed in brackets ( [ ] ) and their elements and size can be changed, while tuples are enclosed in parentheses ( ( ) ) and cannot be updated.
- Tuples can be thought of as read-only lists.
- For example –

## Python Tuples

- `tuple = ( 'xyz', 456 , 2.23, 'rama', 50.2 )`
- `othertuple = (123, 'rama')`
  
- `print tuple` # Prints the complete tuple
- `print tuple[0]` # Prints first element of the tuple
- `print tuple[1:3]` # Prints elements of the tuple starting from 2nd till 3rd
- `print tuple[2:]` # Prints elements of the tuple starting from 3rd element
- `print othertuple * 2` # Prints the contents of the tuple twice
- `print tuple + othertuple` # Prints concatenated tuples

## Python Dictionary

- Python's dictionaries are kind of hash table type.
  - They work like associative arrays or hashes found in Perl and consist of key-value pairs.
  - A dictionary key can be almost any Python type but are usually numbers or strings.
  - Values, on the other hand, can be any arbitrary Python object.
- 
- Dictionaries are enclosed by curly braces ({ }) and values can be assigned and accessed using square braces ([]).

# Python Dictionary

- `dict = {}`
- `dict['one'] = "This is one"`
- `dict[2] = "This is two"`
  
- `otherdict = {'name': 'rama','code':6000, 'dept': 'support'}`
  
- `print dict['one'] # Prints value for 'one' key`
- `print dict[2] # Prints value for 2 key`
- `print otherdict # Prints complete dictionary`
- `print otherdict.keys() # Prints all the keys`
- `print otherdict.values() # Prints all the values`

## Python - Basic Operators

- Python language supports the following types of operators.
- Arithmetic Operators
- Comparison (Relational) Operators
- Assignment Operators
- Logical Operators
- Bitwise Operators
- Membership Operators
- Identity Operators

# Python - Arithmetic Operators

Operator	Description	Example
+ Addition	Adds values on either side of the operator.	$a + b = 30$
- Subtraction	Subtracts right hand operand from left hand operand.	$a - b = -10$
* Multiplication	Multiplies values on either side of the operator	$a * b = 200$
/ Division	Divides left hand operand by right hand operand	$b / a = 2$
% Modulus	Divides left hand operand by right hand operand and returns remainder	$b \% a = 0$
** Exponent	Performs exponential (power) calculation on operators	$a^{**}b = 10 \text{ to the power } 20$
//	Floor Division - The division of operands where the result is the quotient in which the digits after the decimal point are removed. But if one of the operands is negative, the result is floored, i.e., rounded away from zero (towards negative infinity)	$9//2 = 4 \text{ and } 9.0//2.0 = 4.0, -11//3 = -4, -11.0//3 = -4.0$

# Python - Comparison Operators ( also called Relational Operators)

Operator	Description	Example
<code>==</code>	If the values of two operands are equal, then the condition becomes true.	$(a == b)$ is not true.
<code>!=</code>	If values of two operands are not equal, then condition becomes true.	$(a != b)$ is true.
<code>&lt;&gt;</code>	If values of two operands are not equal, then condition becomes true.	$(a <> b)$ is true. This is similar to <code>!=</code> operator.
<code>&gt;</code>	If the value of left operand is greater than the value of right operand, then condition becomes true.	$(a > b)$ is not true.
<code>&lt;</code>	If the value of left operand is less than the value of right operand, then condition becomes true.	$(a < b)$ is true.
<code>&gt;=</code>	If the value of left operand is greater than or equal to the value of right operand, then condition becomes true.	$(a >= b)$ is not true.
<code>&lt;=</code>	If the value of left operand is less than or equal to the value of right operand, then condition becomes true.	$(a <= b)$ is true.

# Python - Assignment Operators

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

# Python - Bitwise Operators

Operator	Description	Example
=	Assigns values from right side operands to left side operand	c = a + b assigns value of a + b into c
+= Add AND	It adds right operand to the left operand and assign the result to left operand	c += a is equivalent to c = c + a
-= Subtract AND	It subtracts right operand from the left operand and assign the result to left operand	c -= a is equivalent to c = c - a
*= Multiply AND	It multiplies right operand with the left operand and assign the result to left operand	c *= a is equivalent to c = c * a
/= Divide AND	It divides left operand with the right operand and assign the result to left operand	c /= a is equivalent to c = c / a
%= Modulus AND	It takes modulus using two operands and assign the result to left operand	c %= a is equivalent to c = c % a
**= Exponent AND	Performs exponential (power) calculation on operators and assign value to the left operand	c **= a is equivalent to c = c ** a
//= Floor Division	It performs floor division on operators and assign value to the left operand	c //= a is equivalent to c = c // a

# Python - Membership Operators

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

Operator	Description	Example
in	Evaluates to true if it finds a variable in the specified sequence and false otherwise.	x in y, here in results in a 1 if x is a member of sequence y.
not in	Evaluates to true if it does not finds a variable in the specified sequence and false otherwise.	x not in y, here not in results in a 1 if x is not a member of sequence y.

# Python - Identity Operators

Identity operators compare the memory locations of two objects.

Operator	Description	Example
is	Evaluates to true if the variables on either side of the operator point to the same object and false otherwise.	x is y, here <b>is</b> results in 1 if id(x) equals id(y).
is not	Evaluates to false if the variables on either side of the operator point to the same object and true otherwise.	x is not y, here <b>is not</b> results in 1 if id(x) is not equal to id(y).

## Assignments :

- 1. Write a python program to demonstrate the relational operators.
- 2. Write a python program to demonstrate the bitwise operators.
- 3. Write a python program to print the type of the variables used in the program.
- 4. Write a python program to read values from the user and apply the arithmetic operations on those values.[Do not use exception handling now.]

## Eval function:

- sum = eval(input("enter the expression:"))
- # This is an in-built function available in python, which takes the strings as an input.
- # The strings which we pass to it should, generally, be expressions.
- # The eval() function takes the expression in the form of string and evaluates it and returns the result.
- print(sum)

# Memory Management in Python:

- Memory management is the process of efficiently allocating, de-allocating, and coordinating memory so that all the different processes run smoothly and can optimally access different system resources. Memory management also involves cleaning memory of objects that are no longer being accessed.
- In Python, the memory manager is responsible for these kinds of tasks by periodically running to clean up, allocate, and manage the memory. Unlike C, Java, and other programming languages, Python manages objects by using reference counting. This means that the memory manager keeps track of the number of references to each object in the program. When an object's reference count drops to zero, which means the object is no longer being used, the garbage collector (part of the memory manager) automatically frees the memory from that particular object.
- The user need not to worry about memory management as the process of allocation and de-allocation of memory is fully automatic. The reclaimed memory can be used by other objects..]

## Garbage collector in Python:

- Python removes those objects that are no longer in use or can say that it frees up the memory space. This process of vanish the unnecessary object's memory space is called the Garbage Collector. The Python garbage collector initiates its execution with the program and is activated if the reference count falls to zero.
- When we assign the new name or placed it in containers such as a dictionary or tuple, the reference count increases its value. If we reassign the reference to an object, the reference count decreases its value. It also decreases its value when the object's reference goes out of scope or an object is deleted.
- As we know, Python uses the dynamic memory allocation which is managed by the Heap data structure. Memory Heap holds the objects and other data structures that will be used in the program. Python memory manager manages the allocation or de-allocation of the heap memory space through the API functions.

## Python Objects in Memory:

- Each variable in Python acts as an object. Objects can either be simple (containing numbers, strings, etc.) or containers (dictionaries, lists, or user defined classes).
- Furthermore, Python is a dynamically typed language which means that we do not need to declare the variables or their types before using them in a program.

## References :

- <https://realpython.com/>
- <https://beginnersbook.com>
- <https://www.openbookproject.net/>
- <https://docs.python.org/>
- <https://www.javatpoint.com/>
- <https://www.w3schools.com/>
- <https://www.tutorialspoint.com/>
- <https://www.javatpoint.com/>
- <https://stackabuse.com/>
- <https://dotnettutorials.net/>



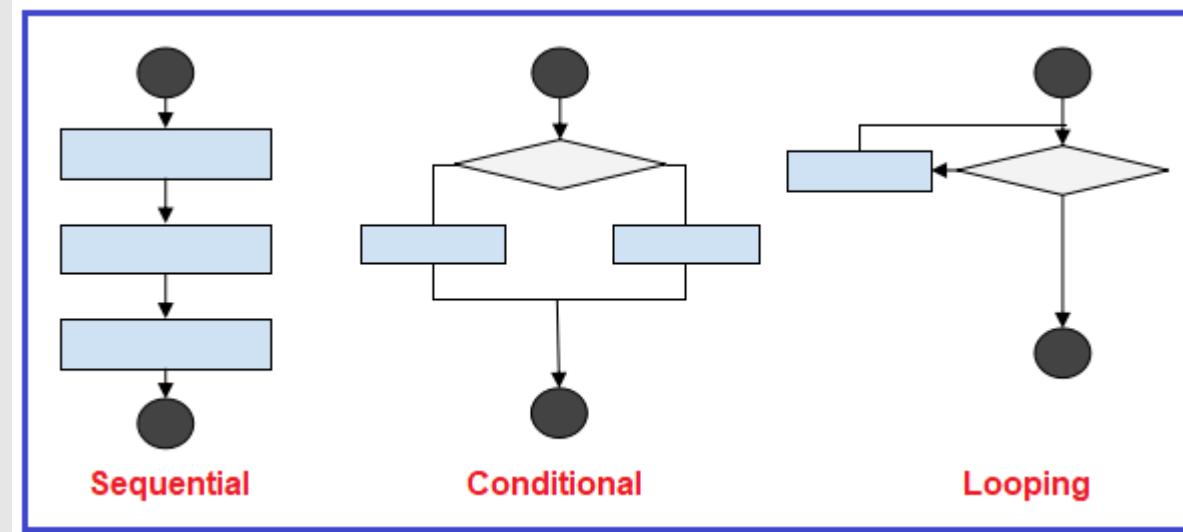
# CONTROL FLOW STATEMENTS IN PYTHON

Unit III

# Introduction

- In programming languages, flow control means the order in which the statements or instructions, that we write, get executed. In order to understand a program, we should be aware of what statements are being executed and in which order.
  - So, understanding the flow of the program is very important.
  - There are, generally, three ways in which the statements will be executed. They are,
- 
- Sequential
  - Conditional
  - Looping

# Introduction



<https://dotnettutorials.net/>

# Sequential Statements in Python

- The statements are executed sequentially; one by one.
- E.g. `print("once")`
- `print("there")`
- `print("was")`
- `print("a kingdom")`
- Output :
- once
- there
- was
- a kingdom

# Conditional Statements in Python

- Conditional statements are also called decision making statements.
- There are three types of conditional statements in python. They are as follows:
  - if statement
  - if else statement
  - if elif else

# Conditional Statements in Python

- if statement

**Syntax:**

```
if condition:  
    if block statements
```

```
out of if block statements
```

- The if statement contains an expression/condition. As per the syntax colon (:) is mandatory otherwise it throws syntax error.
- Condition gives the result as a bool type, either True or False.
- If the condition result is True, then if block statements will be executed.
- If the condition result is False, then if block statements won't execute.

# Conditional Statements in Python

- if i < 4:
  - print("i less than four")
- elif i > 4 and i < 10 :
  - print("value of i is :",i)
- else:
  - print("i not in range")

# Conditional Statements in Python

- There can be zero or more elif parts, and the else part is optional.
- The keyword ‘elif’ is short for ‘else if’, and is useful to avoid excessive indentation.
- An if ... elif ... elif ... sequence is a substitute for the switch or case statements found in other languages.

# Conditional Statements in Python

- There can be zero or more elif parts, and the else part is optional.
- The keyword ‘elif’ is short for ‘else if’, and is useful to avoid excessive indentation.
- An if ... elif ... elif ... sequence is a substitute for the switch or case statements found in other languages.

# Looping Statements in Python

- While statement.
- # while statement
- num1 = 1
- num2 = 2
- # print the mathematical power of num2 raise to num1
- while num1 <= 10:
  - print(num2, "^", num1, " = ", num2\*\*num1)
  - num1 += 1

# Looping Statements in Python

- **For statements:**
- Basically, a for loop is used to iterate elements one by one from sequences like string, list, tuple, etc.
- While iterating elements from sequence we can perform operations on every element.
- Syntax :
- `for variable in sequence:`
  - statements

# Looping Statements in Python

- # using the range function
- for i in range(1,10,2):
  - print("i = ", i)
- # printing elements of a list
- list1 = [10,20,'abc',123.34]
- for x in list1:
  - print(x)
- # printing characters in a string
- string1 = "This is the example"
- for ch in string1:
  - print(ch)

# Looping Statements in Python

- # adding the elements of a tuple
- tuple1 = (10,20,30,40)
- sum=0
- for t in tuple1:
- sum = sum + t
- print("summation of the tuple is : ",sum)

# Looping Statements in Python

- Nested loops in Python:
- rows = range(1, 10)
- for x in rows:
  - for star in range(1, x+1):
  - print('\*', end=' ')
  - print()

# Looping Statements in Python

- Loops with else block in Python:
- In python, loops can also have an else block. These else blocks will get executed irrespective of the loop i.e even if the loop gets executed or not.

for with else block	while with else block
<pre>for variable in sequence:     statements else:     statements</pre>	<pre>while condition:     statements else:     statements</pre>

# Looping Statements in Python

- The continue statement continues with the next iteration of the loop:
  - for num in range(2, 10):
  - ... if num % 2 == 0:
  - ... print("Found an even number", num)
  - ... continue
  - ... print("Found an odd number", num)
- The break statement breaks out of the innermost enclosing for or while loop.

# Looping Statements in Python

- PASS statement in Python:
- The pass statement is used when you have to include some statement syntactically but doesn't want to perform any operation.
- num = [10, 20, 30, 400, 500, 600]
- for i in num:
  - if i<100:
  - print("pass statement executed")
  - pass
  - else:
  - print("Give these numbers to the students: ",i)

# Looping Statements in Python

- RETURN statement in Python:
- Return statement is one which returns something.
- This statement is used with functions.

# Strings in Python

- A group of characters enclosed within single or double or triple quotes is called as string.
- We can say string is a sequential collection of characters.

## Syntax:

```
variable_name = 'String1'  
variable_name = "string2"  
variable_name = ""string3""  
variable_name = """string4"""
```

```
#single quotes  
#double quotes  
#triple single quotes  
#triple double quotes
```

# Strings in Python

- Like many other popular programming languages, strings in Python are arrays of bytes representing unicode characters.
- However, Python does not have a character data type, a single character is simply a string with a length of 1.
- Square brackets can be used to access elements of the string.

# Strings in Python

- Generally, to create a string mostly used syntax is double quotes syntax.
- If you want to create multiple lines of string, then triple single or triple double quotes are the best to use.
- Inside string we can use single and double quotes.
  - `s1 = "Welcome to 'python' learning"`
  - `s2 = 'Welcome to "python" learning'`
  - `s3 = """Welcome to 'python' learning"""`
  - `print(s1)`
  - `print(s2)`
  - `print(s3)`
- Output:

```
Welcome to 'python' learning
Welcome to "python" learning
Welcome to 'python' learning
```

# Strings in Python

- Accessing string characters in python:
- We can access string characters in python by using,
  
- Indexing
- Slicing
  
- If we are trying to access characters of a string with out of range index, then we will get error as IndexError.

# Strings in Python

- Indexing:
- Indexing means a position of string's characters where it stores. We need to use square brackets [] to access the string index. String indexing result is string type. String indices should be integer otherwise we will get an error. We can access the index within the index range otherwise we will get an error.
- Positive indexing: The position of string characters can be a positive index from left to right direction (we can say forward direction). In this way, the starting position is 0 (zero).
- Negative indexing: The position of string characters can be negative indexes from right to left direction (we can say backward direction). In this way, the starting position is -1 (minus one).

# Strings in Python

- Slicing:
- A substring of a string is called a slice.
- A slice can represent a part of string or a piece of string.
- String slicing result is string type. We need to use square brackets [] in slicing.
- E.g. `string_name[start:stop:step]`

# Strings in Python

- The **len()** function returns the length of a string:
- `a = "Hello, World!"`
- `print(len(a))`
- To check if a certain phrase or character is present in a string, we can use the keyword **in**.
- Example
- Check if "free" is present in the following text:
- `txt = "The best things in life are free!"`
- `print("free" in txt)`

# Strings in Python

- To check if a certain phrase or character is NOT present in a string, we can use the keyword **not in**.

◦ <b>Function</b>	<b>Description</b>
◦ <code>chr()</code>	Converts an integer to a character
◦ <code>ord()</code>	Converts a character to an integer
◦ <code>len()</code>	Returns the length of a string
◦ <code>str()</code>	Returns a string representation of an object

# Strings in Python

- Unlike Java, the '+' does not automatically convert numbers or other types to string form.
- The str() function converts values to a string form so they can be combined with other strings.
- `pi = 3.14`
- `##text = 'The value of pi is ' + pi ## NO, does not work`
- `text = 'The value of pi is ' + str(pi) ## yes`

# Strings in Python

- Python has a set of built-in methods that you can use on strings.
- The **upper()** method returns the string in upper case:
  - `a = "Hello, World!"`
  - `print(a.upper())`
- The **lower()** method returns the string in lower case:
  - `a = "Hello, World!"`
  - `print(a.lower())`

# Strings in Python

- The **strip()** method removes any whitespace from the beginning or the end:

- `a = " Hello, World! "`

- `print(a.strip())` # returns "Hello, World!"

- The **replace()** method replaces a string with another string:

- `a = "Hello, World!"`

- `print(a.replace("H", "J"))`

# Strings in Python

- The **split()** method returns a list where the text between the specified separator becomes the list items.
- Example
- The **split()** method splits the string into substrings if it finds instances of the separator:
  - `a = "Hello, World!"`
  - `print(a.split(",")) # returns ['Hello', ' World!']`

# Strings in Python

- **ord(c)**
  - Returns an integer value for the given character.
    - `ord('a')`
    - 97
- **chr(n)**
  - Returns a character value for the given integer.
    - `chr(97)`
    - ‘a’
- The **replace()** method replaces a specified phrase with another specified phrase.
- `string.replace(oldvalue, newvalue, count)` # count is optional

# Strings in Python

- **str.format()** is one of the string formatting methods in Python3, which allows multiple substitutions and value formatting. This method lets us concatenate elements within a string through positional formatting.
- **Using a Single Formatter :**
- Formatters work by putting in one or more replacement fields and placeholders defined by a pair of curly braces {} into a string and calling the str.format(). The value we wish to put into the placeholders and concatenate with the string passed as parameters into the format function.
- Syntax : { } .**format(value)**
- Parameters :
- (value) : Can be an integer, floating point numeric constant, string, characters or even variables.
- Return type : Returns a formatted string with the value passed as parameter in the placeholder position.

# Strings in Python

- we can combine strings and numbers by using the **format()** method!
- The **format()** method takes the passed arguments, formats them, and places them in the string where the placeholders `{}` are:
- Example
- Use the `format()` method to insert numbers into strings:
- `age = 36`
- `txt = "My name is John, and I am {}"`
- `print(txt.format(age))`

# Strings in Python

- The `format()` method takes unlimited number of arguments, and are placed into the respective placeholders:
- **Example :**
  - `quantity = 3`
  - `itemno = 567`
  - `price = 49.95`
  - `myorder = "I want {} pieces of item {} for {} dollars."`
  - `print(myorder.format(quantity, itemno, price))`

# Strings in Python

- You can use index numbers {0} to be sure the arguments are placed in the correct placeholders:
- Example
  - quantity = 3
  - itemno = 567
  - price = 49.95
  - myorder = "I want to pay {2} dollars for {0} pieces of item {1}."
  - print(myorder.format(quantity, itemno, price))

# Strings in Python

Method	Description
capitalize()	Converts the first character to upper case
casefold()	Converts string into lower case
center()	Returns a centered string
count()	Returns the number of times a specified value occurs in a string
encode()	Returns an encoded version of the string
endswith()	Returns true if the string ends with the specified value
expandtabs()	Sets the tab size of the string
find()	Searches the string for a specified value and returns the position of where it was found
format()	Formats specified values in a string
format_map()	Formats specified values in a string
index()	Searches the string for a specified value and returns the position of where it was found

# Strings in Python

Method	Description
isalnum()	Returns True if all characters in the string are alphanumeric
isalpha()	Returns True if all characters in the string are in the alphabet
isdecimal()	Returns True if all characters in the string are decimals
isdigit()	Returns True if all characters in the string are digits
isidentifier()	Returns True if the string is an identifier
islower()	Returns True if all characters in the string are lower case
isnumeric()	Returns True if all characters in the string are numeric
isprintable()	Returns True if all characters in the string are printable
isspace()	Returns True if all characters in the string are whitespaces
istitle()	Returns True if the string follows the rules of a title
isupper()	Returns True if all characters in the string are upper case

# Strings in Python

Method	Description
join()	Joins the elements of an iterable to the end of the string
ljust()	Returns a left justified version of the string
lower()	Converts a string into lower case
lstrip()	Returns a left trim version of the string
maketrans()	Returns a translation table to be used in translations
partition()	Returns a tuple where the string is parted into three parts
replace()	Returns a string where a specified value is replaced with a specified value
rfind()	Searches the string for a specified value and returns the last position of where it was found
rindex()	Searches the string for a specified value and returns the last position of where it was found
rjust()	Returns a right justified version of the string
rpartition()	Returns a tuple where the string is parted into three parts

# Strings in Python

Method	Description
rsplit()	Splits the string at the specified separator, and returns a list
rstrip()	Returns a right trim version of the string
split()	Splits the string at the specified separator, and returns a list
splitlines()	Splits the string at line breaks and returns a list
startswith()	Returns true if the string starts with the specified value
strip()	Returns a trimmed version of the string
swapcase()	Swaps cases, lower case becomes upper case and vice versa
title()	Converts the first character of each word to upper case
translate()	Returns a translated string
upper()	Converts a string into upper case
zfill()	Fills the string with a specified number of 0 values at the beginning

## Assignments Set 2.1

- Write a program to read three numbers from user and compare the greatest and smallest of those numbers and print the appropriate message.
- Write a program to print the addition of first 30 natural numbers.(Use while statement)
- Write a program to print even numbers in between 10 to 20 by using while loop.
- Write a program to search a number or a string in a list and print it. Read the number /string to be searched, from the user. Make use of for and if statements.
- Write a program to print the prime and non-prime numbers.(Can also decide a range like in between 20 to 50 numbers).

## Assignments Set 2.2

- Write a program to read a string from the user. Also, get a substring from the user and search it in the given string.
- Write a program to read a string and read the substring from a specific position and print it.
- Write a program to count the occurrences of a character in the given string. (for e.g. in string “Hello World” count the occurrences of the letter ‘l’).
- Write a program to capitalize the first letter of a given string.(e.g. this is a example)
- Write a program to replace a substring in a string with any other substring or special characters. ( e.g. In string s =“ This has to be the beginning of the chapter”, where replace the substring ‘ the beginning’ with ‘the end’)
- Count all lower case, upper case, digits, and special symbols from a given string.  
For e.g. string = ‘Py#@th12)ON’

## Assignments Set 2.2

- Given a string of odd length greater 7, return a string made of the middle three chars of a given String. For e.g. str1 = ‘Pythonexample’ and str2 = ‘trialprograms’ , the resulting string = ‘exarog’
- Find all occurrences of “sample” in given string ignoring the case. For e.g. string= ‘This is a SAMPLE program. Solve the sample program’.
- Given a string, calculate the occurrences of each character in the string. For e.g. ‘Hello’ H:1, e:1,l :2, and o:1

# Functions

- A function is a block of code which only runs when it is called.
- You can pass data, known as parameters, into a function.
- A function can return data as a result.
- Creating a Function
- In Python a function is defined using the def keyword:
- Example
- ```
def my_function():
```
- ```
    print("Hello from a function")
```

# Functions

- Calling a Function
- To call a function, use the function name followed by parenthesis:
  
- Example
- ```
def my_function():  
    print("Hello from a function")
```
  
- `my_function()`

# Fixed/Positional Arguments

- Here we have defined a function demo() with two parameters name and age, which means this function can accept two arguments while we are calling it.
- def demo(name,age):
  - "This function displays the age of a person"
  - print(name + " is " + age + " years old")
- # calling the function
- demo("Sachin","47")
- Output:
- Sachin is 47 years old

# Fixed/Positional Arguments

- Let's see what happens we call this function with variable number of arguments:  
Here we are calling the function demo() with a single argument, as you can see in the output that we got an error.
- def demo(name,age):
  - print(name + " is " + age + " years old")
- # calling the function
- demo("Sachin")
- Output:
- TypeError: demo() missing 1 required positional argument: 'age'

# Variable Function Arguments

- There are three ways in which we can use the variable function arguments.
- Python Default arguments
- Python Keyword arguments
- Python Arbitrary arguments
- **Python Default Arguments**
- We can provide **default values** to the function arguments. Once we provide a default value to a function argument, the argument becomes **optional during the function call**, which means it is not mandatory to pass that argument during function call

# Variable Function Arguments

- **Note:**
  1. If you provide a value to the default argument during a function call, it **overrides the default value**.
  2. A function can have any number of default arguments, however if an argument is set to be a default, all the arguments to its **right** must always be default.
- For example, `def demo(name = "Sachin", age):` would throw an error (**SyntaxError: non-default argument follows default argument**) because **name** is a default argument so all the arguments that are following it, must always be default.

# Variable Function Arguments

- `def demo(name, age = "30"):`
- `""" This function displays the name and age of a person If age is not provided, it's default value 30 would be displayed. """`
- `print(name + " is " + age + " years old")`
  
- `demo("Rohit")`
- `demo("Rahul", "46")`
- `demo("Saurav", "48")`

# Variable Function Arguments

- **Python Keyword Arguments**
- When we pass the values during function call, they are assigned to the respective **arguments according to their position**.
- For example if a function is defined like this:
- `def demo(name, age):`
- and we are calling the function like this:
- `demo("Rohit", "30")` then the value "Rohit" is assigned to the argument name and the value "30" is assigned to the argument age.
- Such arguments are called **positional arguments**.

# Variable Function Arguments

- Python allows us to pass the arguments in non-positional manner using **keyword arguments**.
  
- ```
def demo(name, age):  
    print(name + " is " + age + " years old")
```
  
- # 2 keyword arguments (In order)
- `demo(name = "Rohit", age = "30")`
- # 2 keyword arguments (Not in order)
- `demo(age = "46", name = "Rahul")`
- # 1 positional and 1 keyword argument
- `demo("Srinath", age = "49")`

# Variable Function Arguments

- When we are **mixing** the positional and keyword arguments, the **keyword argument must always be after the non-keyword argument** (positional argument).
- **Arbitrary Function Arguments**
- Functions in Python can accept arbitrary number of arguments.
- `def find_sum(*args):`
  - “Function returns the sum of all values”
  - `sum = 0`
  - `for i in args:`
    - `sum += i`
  - `return sum`

# Variable Function Arguments

- We use the \* operator to indicate that the function accepts arbitrary number of arguments.
- The `find_sum()` function returns the sum of all arguments.
- When an argument in a function call is preceded by an asterisk (\*), it indicates that the argument is a tuple that should be **unpacked** and passed to the function as separate values.
- For e.g. `find_sum(10,20,30,40)`
  
- Although this type of unpacking is called **tuple** unpacking, it doesn't only work with tuples.
- The asterisk (\*) operator can be applied to any iterable in a Python function call.

# Variable Function Arguments

- We can also use the `**` construct in our functions.
  - In such a case, the function will accept a dictionary.
  - The dictionary has arbitrary length. We can then normally parse the dictionary, as usual.
  - Preceding a parameter in a Python function definition by a double asterisk (`**`) indicates that the corresponding arguments, which are expected to be key=value pairs, should be packed into a dictionary:
- 
- ```
def f(**kwargs):
```
  - `print(kwargs)`
  - `print(type(kwargs))`
  - `for key, val in kwargs.items():`
  - `print(key, '->', val)`
  - `f('a'=1, 'b'=2, 'c'=3)`

# Variable Function Arguments

- **Argument dictionary unpacking** is analogous to argument tuple unpacking.
- When the double asterisk (\*\*) precedes an argument in a Python function call, it specifies that the argument is a dictionary that should be unpacked, with the resulting items passed to the function as keyword arguments.
- The items in the dictionary d are unpacked and passed to f() as keyword arguments.
- So, `f(**d)` is equivalent to `f('a':1, 'b': 2, 'c':3)`

# Variable length Function parameter

- def inventory(category, \*items):
  - print("%s [items=%d]: "% (category, len(items)), items)
  - for item in items:
    - print("-", item)
- inventory('Electronics', 'tv', 'Computer', 'Geyser', 'refrigerator', 'heater')  
inventory('Books', 'python', 'java', 'SPM', 'c++')
- The output of the above code goes like this.
- Electronics [items=5]: ('tv', 'Computer', 'Geyser', 'refrigerator', 'heater ')
  - - tv - Computer - Geyser - refrigerator - heater
- Books [items=4]: ('python', 'java', 'SPM', 'c++)
  - - python - java - SPM - c++

# Local Variables inside a Function

- A local variable has visibility only inside a code block such as the function def.
- They are available only while the function is executing.
- `def fn(a, b) :`
- `temp = 1`
- `for iter in range(b) :`
- `temp = temp*a`
- `return temp`
- `print(fn(2, 4))`
- `print(temp)` # error : can not access 'temp' out of scope of function 'fn'
- `print(iter)` # error : can not access 'iter' out of scope of function 'fn'

# Local Variables inside a Function

- Function's local variables don't retain values between calls.
- The names used inside a def do not conflict with variables outside the def, even if you've used the same names elsewhere.
- In Python, the variables assignment can occur at three different places.
  - Inside a def – it is local to that function
  - In an enclosing def – it is nonlocal to the nested functions
  - Outside all def(s) – it is global to the entire file

# Global Variables in a Function

- The global keyword is a statement in Python. It enables variable (names) to retain changes that live outside a def, at the top level of a module file.
- In a single global statement, you can specify one or more names separated by commas.
- All the listed names attach to the enclosing module's scope when assigned or referenced within the function body.

# Global Variables in a Function

- `x = 5` `y = 55`
- `def fn():`
- `global x`
- `x = [3, 7]`
- `y = [1, 33, 55]` # a local 'y' is assigned and created here
- # whereas, 'x' refers to the global name
- `fn()`
- `print(x, y)`
  
- In the above code, 'x' is a global variable which will retain any change in its value made in the function. Another variable 'y' has local scope and won't carry forward the change.

# Assignments- Functions

- Write a Python function to find the Max of three numbers.
- Write a Python function to sum all the numbers in a list.
- Write a Python function to multiply all the numbers in a list.
- Write a Python program to reverse a string.
- Write a Python function to calculate the factorial of a number (a non-negative integer). The function accepts the number as an argument.
- Write a Python function that checks whether a passed string is palindrome or not.

# Assignments : Functions

- Write a Python function to check whether a number is perfect or not.
- According to Wikipedia : In number theory, a perfect number is a positive integer that is equal to the sum of its proper positive divisors, that is, the sum of its positive divisors excluding the number itself (also known as its aliquot sum). Equivalently, a perfect number is a number that is half the sum of all of its positive divisors (including itself).

Example : The first perfect number is 6, because 1, 2, and 3 are its proper positive divisors, and  $1 + 2 + 3 = 6$ . Equivalently, the number 6 is equal to half the sum of all its positive divisors:  $(1 + 2 + 3 + 6) / 2 = 6$ . The next perfect number is  $28 = 1 + 2 + 4 + 7 + 14$ . This is followed by the perfect numbers 496 and 8128.

- Write a Python function to get a string from a given string where all occurrences of its first char have been changed to '@', except the first char itself.

# Functions

- **Python abs()** is a built-in function available with the standard library of python. It returns the absolute value for the given number. Absolute value of a number is the value without considering its sign. The number can be integer, floating point number or complex number. If the given number is complex, then it will return its magnitude.
  
- Syntax:
- `abs(value)`
- Parameters: `(value)`
- The input value to be given to `abs()` to get the absolute value. It can be an integer, a float, or a complex number.
  
- Return Value:
- It will return the absolute value for the given number.

# Functions

- **# testing abs() for an integer and float**
- **int\_num = -50**
- **float\_num = -15.70**
- **print("The absolute value of an integer number is:", abs(int\_num))**
- **print("The absolute value of a float number is:", abs(float\_num))**
- **Output:**
- **The absolute value of an integer number is: 50**
- **The absolute value of a float number is: 15.7**

# Functions

- **Abs() is a built-in function available with python, and it will return you the absolute value for the input given.**
- **Value is an input value to be given to abs() to get the absolute value. It can be an integer, a float, or a complex number.**
- **The abs() method takes one argument, i.e. the value you want to get the absolute.**
- **The abs function returns the absolute value for the given number.**

# Functions

- **Round()**
- Round() is a built-in function available with python. It will return you a float number that will be rounded to the decimal places which are given as input.
- If the decimal places to be rounded are not specified, it is considered as 0, and it will round to the nearest integer.
- Syntax:
- **round(float\_num, num\_of\_decimals)**
- Parameters
- float\_num: the float number to be rounded.
- num\_of\_decimals: (optional) The number of decimals to be considered while rounding. It is optional, and if not specified, it defaults to 0, and the rounding is done to the nearest integer.

# Functions

- Description
- The round() method takes two argument.
  - the number to be rounded and
  - the decimal places it should consider while rounding.
- The second argument is optional and defaults to 0 when not specified, and in such case, it will round to the nearest integer, and the return type will also be an integer.
- When the decimal places, i.e. the second argument, is present, it will round to the number of places given. The return type will be a float.
- If the number after the decimal place given
  - $\geq 5$  than + 1 will be added to the final value
  - $< 5$  than the final value will return as it is up to the decimal places mentioned.

# Functions

- Python range() is a built-in function available with Python from Python(3.x), and it gives a sequence of numbers based on the start and stop index given.
- In case the start index is not given, the index is considered as 0, and it will increment the value by 1 till the stop index.
- For example range(5) will output you values 0,1,2,3,4 .
- The Python range()is a very useful command and mostly used when you have to iterate using for loop.

# Functions

- **Syntax**
- `range(start, stop, step)`
- **Parameters**
  - start: (optional) The start index is an integer, and if not given, the default value is 0.
  - stop: The stop index decides the value at which the range function has to stop. It is a mandatory input to range function. The last value will be always 1 less than the stop value.
  - step: (optional).The step value is the number by which the next number is range has to be incremented, by default, it is 1.
- **Return value:**
  - The return value is a sequence of numbers from the given start to stop index.

# Functions

- Using for-loop with Python range()
- Example:
- arr\_list = ['Mysql', 'Mongodb', 'PostgreSQL', 'Firebase']
- ```
for i in range(len(arr_list)):  
    print(arr_list[i], end = " ")
```
- Output:
- MysqlMongodb PostgreSQL Firebase

# Functions

- Using Python range() as a list
- In this example will see how to make use of the output from range as a list.
- Example:
- `print(list(range(10)))`
- Output:
- `[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]`

# Functions

- Using index
- The index is used with range to get the value available at that position. If the range value is 5, to get the startvalue, you can use range(5)[0] and the next value range(5)[1] and so on.
- Example:
- startvalue = range(5)[0]
- print("The first element in range is = ", startvalue)
- secondvalue = range(5)[1]
- print("The second element in range is = ", secondvalue)
- lastvalue = range(5)[-1]
- print("The first element in range is = ", lastvalue)

# Functions

- Python map() applies a function on all the items of an iterator given as input.
  - An iterator, for example, can be a list, a tuple, a set, a dictionary, a string, and it returns an iterable map object.
  - Python map() is a built-in function.
- 
- Syntax:
  - `map(function, iterator1, iterator2 ... iteratorN)`
- 
- function: A mandatory function to be given to map, that will be applied to all the items available in the iterator.
  - iterator: An iterable compulsory object. It can be a list, a tuple, etc. You can pass multiple iterator objects to map() function.
- Return Value
  - The map() function is going to apply the given function on all the items inside the iterator and return an iterable map object i.e a tuple, a list, etc.

# Functions

- The map() function takes two inputs as a function and an iterable object. The function that is given to map() is a normal function, and it will iterate over all the values present in the iterable object given.
- For example, consider you have a list of numbers, and you want to find the square of each of the numbers.
- To get the output, we need the function that will return the square of the given number. The function will be as follows:
- ```
def square(n):  
    return n*n
```
- The list of items that we want to find the square is as follows:
- `my_list = [2,3,4,5,6,7,8,9]`

# Functions

- use map() python built-in function to get the square of each of the items in my\_list.
- ```
def square(n):  
    return n*n
```
- ```
my_list = [2,3,4,5,6,7,8,9]  
updated_list = map(square, my_list)  
print(updated_list)  
print(list(updated_list))
```
- Output:
  - <map object at 0x0000002C59601748>
  - [4, 9, 16, 25, 36, 49, 64, 81]

# Functions

- Python map() function is a built-in function and can also be used with other built-in functions available in Python. In the example, we are going to make use of Python round() built-in function that rounds the values given.
- Example:
- my\_list = [2.6743,3.63526,4.2325,5.9687967,6.3265,7.6988,8.232,9.6907] .
- Let us print the rounded values for each item present in the list. We will make use of round() as the function to map().

# Functions

- my\_list = [2.6743,3.63526,4.2325,5.9687967,6.3265,7.6988,8.232,9.6907]
- updated\_list = map(round, my\_list)
- print(updated\_list)
- print(list(updated\_list))
- Output:
  - <map object at 0x000000E65F901748>
  - [3, 4, 4, 6, 6, 8, 8, 10]

# Lambda functions

- A lambda function is a small anonymous function.
- A lambda function can take any number of arguments, but can only have one expression.
- **Syntax**
- **lambda arguments : expression**
- The expression is executed, and the result is returned:
- Example:
- Multiply 10 with the argument a, and return the result:
- `x = lambda a : a * 10`
- `print(x(5))`

# Lambda functions

- You can apply the function above to an argument by surrounding the function and its argument with parentheses:
  - `(lambda a: a * 10)(2)`
  - 20
- Lambda functions can take any number of arguments:
  - Example
  - Multiply argument a with argument b and return the result:
    - `x = lambda a, b : a * b`
    - `print(x(5, 6))`

# Why Use Lambda Functions?

- #What does a lambda function return
  - string='example for lambda function'
  - print(lambda string : print(string))
- Output:
- <function <lambda> at 0x00000165C3BF41E0>

# Why Use Lambda Functions?

- This is because the lambda itself returns a function object.
  - In this example, the lambda is not being called by the print function but simply returning the function object and the memory location where it is stored.
  - That's what gets printed at the console.
- 
- #What does a lambda function returns #2
    - x='example for lambda function'
    - (lambda x : print(x))(x)
- 
- Output:
- 
- example for lambda function

## Why Use Lambda Functions?

- This happens because we are defining and immediately passing argument to that lambda function.
- This is sometimes called as immediately invoked function execution.

## lambdas in filter()

- The filter function is used to select some particular elements from a sequence of elements. The sequence can be any iterator like lists, sets, tuples, etc.
- The elements which will be selected is based on some pre-defined constraint.  
It takes 2 parameters:
- A function that defines the filtering constraint
- A sequence (any iterator like lists, tuples, etc.)
- For example,

## lambdas in filter()

- sequences = [10,2,8,7,5,4,3,11,0, 1]
- filtered\_result = filter (lambda x: x > 4, sequences)
- print(list(filtered\_result))
- Here's the output:
- [10, 8, 7, 5, 11]

## lambdas in map()

- the map function is used to apply a particular operation to every element in a sequence. Like filter(), it also takes 2 parameters:
- A function that defines the op to perform on the elements
- One or more sequences
- For example, here is a program that prints the squares of numbers in a given list:
- sequences = [10,2,8,7,5,4,3,11,0, 1]
- filtered\_result = map (lambda x: x\*x, sequences)
- print(list(filtered\_result))
- Output:
- [100, 4, 64, 49, 25, 16, 121, 0, 1]

# Why (and why not) use lambda functions?

- One of the most common use cases for lambdas is in functional programming as Python supports a paradigm (or style) of programming known as functional programming.
- It allows you to provide a function as a parameter to another function (for example, in map, filter, etc.). In such cases, using lambdas offer an elegant way to create a one-time function and pass it as the parameter.

## Summary :

- Lambdas, also known as anonymous functions, are small, restricted functions which do not need a name (i.e., an identifier).
- Every lambda function in Python has 3 essential parts:
- The lambda keyword.
- The parameters (or bound variables), and
- The function body.
- The syntax for writing a lambda is: `lambda parameter: expression`
- Lambdas can have any number of parameters, but they are not enclosed in braces
- A lambda can have only 1 expression in its function body, which is returned by default.
- At the bytecode level, there is not much difference between how lambdas and regular functions are handled by the interpreter.

# Python Documentation string i.e docstring

- Difference between python comments and docstring.
- Python Comments are the useful information that the developers provide to make the reader understand the source code.
- It explains the logic or a part of it used in the code. It is written by using # symbol.
- Whereas Python Docstrings provides a convenient way of associating documentation with Python modules, functions, classes, and methods.

# Function Annotations

- Basic Terminology
- **PEP:** PEP stands for Python Enhancement Proposal. It is a design document that describes new features for Python or its processes or environment. It also provides information to the python community.
- PEP is a primary mechanism for proposing major new features, for example – Python Web Server Gateway Interface, collecting the inputs of the community on the issues and documenting design decisions that have been implemented in Python.
- Function Annotations – PEP 3107 : PEP-3107 introduced the concept and syntax for adding arbitrary metadata annotations to Python. It was introduced in Python3 which was previously done using external libraries in python 2.x

# Function Annotations

- Function annotations are arbitrary python expressions that are associated with various part of functions.
- These expressions are evaluated at compile time and have no life in python's runtime environment.
- Python does not attach any meaning to these annotations.
- They take life when interpreted by third party libraries, for example, mypy.

# Function Annotations

- Purpose of function annotations:
- The benefits from function annotations can only be reaped via third party libraries. The type of benefits depends upon the type of the library, for example
- Python supports dynamic typing and hence no module is provided for type checking.
- Annotations like
- [def foo(a:"int", b:"float"=5.0) -> "int"]
- can be used to collect information about the type of the parameters and the return type of the function to keep track of the type change occurring in the function.
- ‘mypy’ is one such library.

# Function Annotations

- String based annotations can be used by the libraries to provide better help messages at compile time regarding the functionalities of various methods, classes and modules.
- Syntax of function annotations
- They are like the optional parameters that follow the parameter name.
- Note: The word ‘expression’ mentioned can be the type of the parameters that should be passed or comment or any arbitrary string that can be used by external libraries in a meaningful way.

# Function Annotations

- **Annotations for simple parameters :**
- The argument name is followed by ‘:’ which is then followed by the expression.  
Annotation syntax is shown below.
- `def foobar(a: expression, b: expression = 5):`
  
- **Annotations for excess parameters :** Excess parameters for e.g. `*args` and `**kwargs`, allow arbitrary number of arguments to be passed in a function call.  
Annotation syntax of such parameters is shown below.
- `def foobar(*args: expression, *kwargs: expression):`

# Function Annotations

- **Annotations for return type :**
- Annotating return type is slightly different from annotating function arguments. The ‘->’ is followed by expression which is further followed by ‘:’. Annotation syntax of return type is shown below.
- `def foobar(a: expression)->expression:`

# Python Modules

- A module is a file containing Python definitions and statements.
  - A module can define functions, classes, and variables.
  - A module can also include runnable code.
  - Grouping related code into a module makes the code easier to understand and use.
  - It also makes the code logically organized.
- 
- Consider a module to be the same as a code library.
  - A file containing a set of functions you want to include in your application.

# Python Modules

- To create a module just save the code you want in a file with the file extension .py:
- Example
- Save this code in a file named exmodule.py
  
- ```
def greeting(name):  
    print("Hello, " + name)
```

# Python Modules

- Now we can use the module we just created, by using the import statement:
- Example
- Import the module named exmodule, and call the greeting function:
- `import exmodule`
- `exmodule.greeting("Vinayak")`

# Python Modules

- # A simple module, calc.py
- 
- def add(x, y):
- return (x+y)
- 
- def subtract(x, y):
- return (x-y)

# Python Modules

- Variables in Module
- The module can contain functions, as already described, but also variables of all types (arrays, dictionaries, objects etc):
- Example
- Save this code in the file exmodule.py
  - person1 = {
  - "name": "Sachin",
  - "age": 47,
  - "country": "India"
  - }

# Python Modules

- Import the module named exmodule, and access the person1 dictionary:
- `import exmodule`
- `a = exmodule.person1["age"]`
- `print(a)`

# Python Modules

- You can create an alias when you import a module, by using the **as** keyword:
- Example
- Create an alias for exmodule called mx:
- `import exmodule as mx`
- `a = mx.person1["age"]`
- `print(a)`

# Python Modules

- The from import Statement
- Python's from statement lets you import specific attributes from a module.
- The from .. import .. has the following syntax :
- # importing sqrt() and factorial from the module math
- **from math import sqrt, factorial**
- # if we simply do "import math", then # math.sqrt(16) and math.factorial()
- # are required.
- **print(sqrt(16))**
- **print(factorial(6))**

# Python Modules

- The from import \* Statement
- The \* symbol used with the from import the statement is used to import all the names from a module to a current namespace.
- Syntax:
- **from module\_name import \***

# Python Modules

- # Import built-in module random
- from random import \*
- print(dir(random))
- Output:
  - ['\_\_call\_\_', '\_\_class\_\_', '\_\_delattr\_\_', '\_\_dir\_\_', '\_\_doc\_\_', '\_\_eq\_\_', '\_\_format\_\_', '\_\_ge\_\_', '\_\_getattribute\_\_', '\_\_gt\_\_', '\_\_hash\_\_', '\_\_init\_\_', '\_\_init\_subclass\_\_', '\_\_le\_\_', '\_\_lt\_\_', '\_\_module\_\_', '\_\_name\_\_', '\_\_ne\_\_', '\_\_new\_\_', '\_\_qualname\_\_', '\_\_reduce\_\_', '\_\_reduce\_ex\_\_', '\_\_repr\_\_', '\_\_self\_\_', '\_\_setattr\_\_', '\_\_sizeof\_\_', '\_\_str\_\_', '\_\_subclasshook\_\_', '\_\_text\_signature\_\_']

# Python Modules

- The `dir()` function
  - The `dir()` built-in function returns a sorted list of strings containing the names defined by a module.
  - The list contains the names of all the modules, variables, and functions that are defined in a module.
- 
- `# Import built-in module random`
  - `import random`
  - `print(dir(random))`

# Python Modules

- Output:
  - `['BPF', 'LOG4', 'NV_MAGICCONST', 'RECIP_BPF', 'Random', 'SG_MAGICCONST', 'SystemRandom', 'TWOPI', '_BuiltinMethodType', '_MethodType', '_Sequence', '_Set', '__all__', '__builtins__', '__cached__', '__doc__', '__file__', '__loader__', '__name__', '__package__', '__spec__', '__acos', '__bisect', '__ceil', '__cos', '__e', '__exp', '__inst', '__itertools', '__log', '__pi', '__random', '__sha512', '__sin', '__sqrt', '__test', '__test_generator', '__urandom', '__warn', 'betavariate', 'choice', 'choices', 'expovariate', 'gammavariate', 'gauss', 'getrandbits', 'getstate', 'lognormvariate', 'normalvariate', 'paretovariate', 'randint', 'random', 'randrange', 'sample', 'seed', 'setstate', 'shuffle', 'triangular', 'uniform', 'vonmisesvariate', 'weibullvariate']`

# Python Modules

- # importing built-in module math
  - import math
- 
- # using square root(sqrt) function contained # in math module
  - print(math.sqrt(25))
- 
- # using pi function contained in math module
  - print(math.pi)
- 
- # 2 radians = 114.59 degrees
  - print(math.degrees(2))

# Python Modules

- # 60 degrees = 1.04 radians

- print(math.radians(60))

- # Sine of 2 radians

- print(math.sin(2))

- # Cosine of 0.5 radians

- print(math.cos(0.5))

- # Tangent of 0.23 radians

- print(math.tan(0.23))

# Python Modules

- #  $1 * 2 * 3 * 4 = 24$
- `print(math.factorial(4))`
  
- # importing built in module random
- `import random`
  
- # printing random integer between 0 and 5
- `print(random.randint(0, 5))`
  
- # print random floating point number between 0 and 1
- `print(random.random())`

# Python Modules

- # random number between 0 and 100
- `print(random.random() * 100)`
  
- `List = [1, 4, True, 800, "python", 27, "hello"]`
- # using choice function in random module for choosing a random element from a set such as a list
- `print(random.choice(List))`
  
- # importing built in module datetime
- **import** datetime
- **from** datetime **import** date
- **import** time

# Python Modules

- # Returns the number of seconds since the # Unix Epoch, January 1st 1970
  - print(time.time())
- 
- # Converts a number of seconds to a date object
  - print(date.fromtimestamp(454554))

# Python Packages

- Packages are a way of structuring many packages and modules which helps in a well-organized hierarchy of data set, making the directories and modules easy to access.
- Just like there are different drives and folders in an OS to help us store files, similarly packages help us in storing other sub-packages and modules, so that it can be used by the user when necessary.
- **Creating and Exploring Packages**
- To tell Python that a particular directory is a package, we create a file named `_init__.py` inside it and then it is considered as a package and we may create other modules and sub-packages within it.

# Python Packages

- This `__init__.py` file can be left blank or can be coded with the initialization code for the package.
- To create a package in Python, we need to follow these three simple steps:
  - First, we create a directory and give it a package name, preferably related to its operation.
  - Then we put the classes and the required functions in it.
  - Finally we create an `__init__.py` file inside the directory, to let Python know that the directory is a package.

# Python Packages

- Let's create a package named Cars and build three modules in it namely, Honda, Audi and Bugatti.
- First we create a directory and name it Cars.
- Then we need to create modules. To do this we need to create a file with the name Honda.py and create its content by putting this code into it.

# Python Packages

- # Python code to illustrate the Modules
- class Honda:
- # First we create a constructor for this class
- # and add members to it, here models
- def \_\_init\_\_(self):  
        self.models = ['City', 'Accord', 'Civic', 'Amaze']
- # A normal print function
- def outModels(self):  
        print('These are the available models for Honda')  
        for model in self.models:  
            print('\t%s' % model)

# Python Packages

- # Python code to illustrate the Module
- class Audi:
- # First we create a constructor for this class
- # and add members to it, here models
- def \_\_init\_\_(self):
- self.models = ['q7', 'a6', 'a8L', 'q8']
  
- # A normal print function
- def outModels(self):
- print('These are the available models for Audi')
- for model in self.models:
- print('\t%s' % model)

# Python Packages

- # Python code to illustrate the Module
- class Bugatti:
  - # First we create a constructor for this class
  - # and add members to it, here models
  - def \_\_init\_\_(self):
    - self.models = ['Chiron', 'Veyron', 'Divo', 'Centodieci']
  - # A normal print function
  - def outModels(self):
    - print('These are the available models for Bugatti')
    - for model in self.models:
      - print('\t%s' % model)

# Python Packages

- Finally we create the `__init__.py` file.
- This file will be placed inside Cars directory and can be left blank or we can put this initialisation code into it.
  
- `from Honda import Honda`
- `from Audi import Audi`
- `from Bugatti import Bugatti`
  
- Now, let's use the package that we created.
- To do this make a `sample.py` file in the same directory where Cars package is located and add the following code to it:

# Python Packages

- # Import classes from your brand new package
- from Cars import Honda
- from Cars import Audi
- from Cars import Bugatti
  
- # Create an object of Honda class & call its method
- ModHonda = Honda()
- ModHonda.outModels()
  
- # Create an object of Audi class & call its method
- ModAudi = Audi()
- ModAudi.outModels()
  
- # Create an object of Bugatti class & call its method
- ModBugatti = Bugatti()
- ModBugatti.outModels()

# Python Exercises

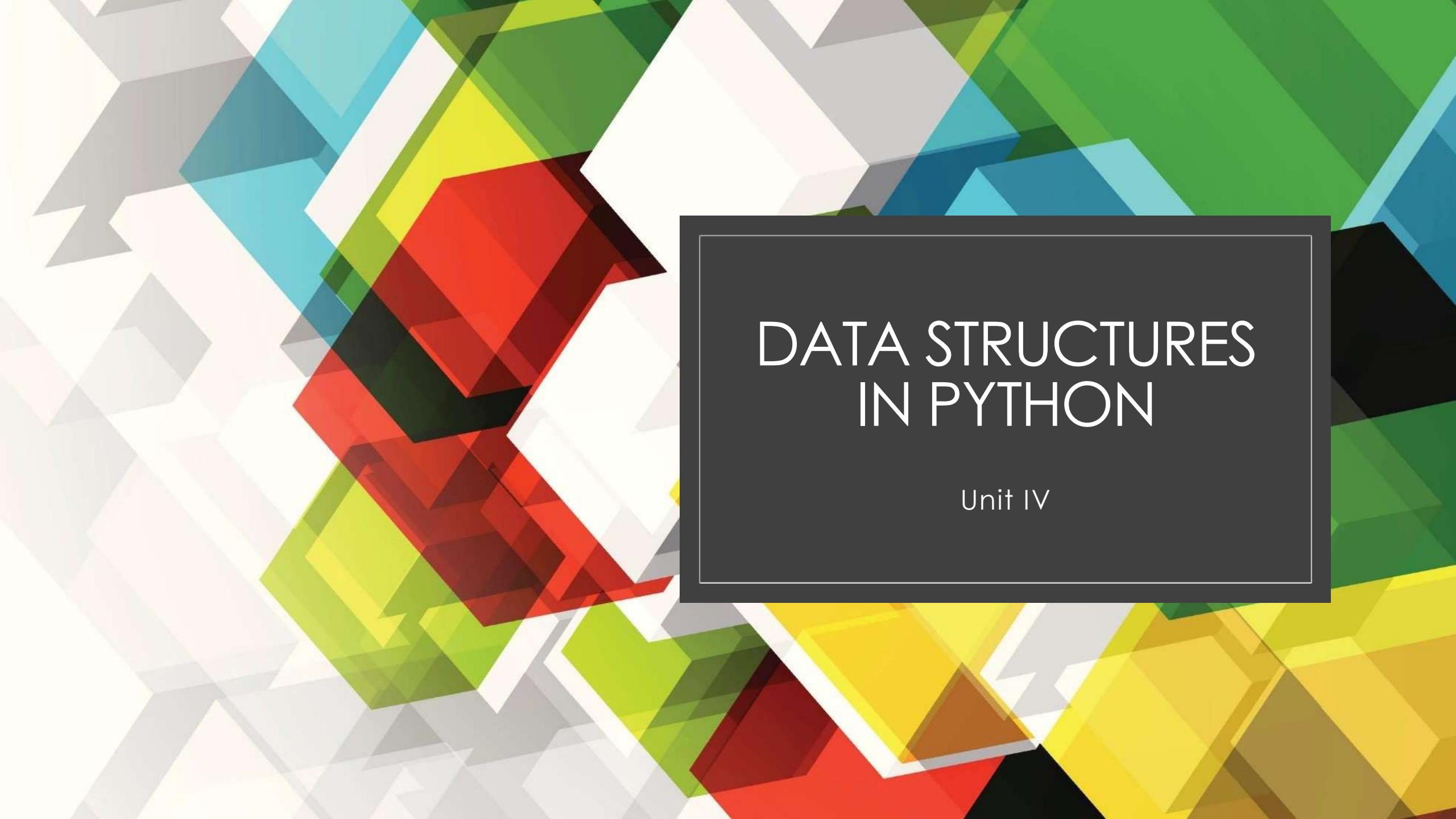
- Write a Python script to display the various Date Time formats.
  - a) Current date and time
  - b) Current year
  - c) Month of year
  - d) Week number of the year
  - e) Weekday of the week
  - f) Day of year
  - g) Day of the month
  - h) Day of week

# Python Exercises

- Write a Python program to determine whether a given year is a leap year.
- Write a Python program to sort a list of dictionaries using Lambda.
- Write a Python program to square and cube every number in a given list of integers using Lambda.
- Write a Python program to create Fibonacci series upto n using Lambda.
- Write a Python program to convert radian to degree.
- Note: The radian is the standard unit of angular measure, used in many areas of mathematics. An angle's measurement in radians is numerically equal to the length of a corresponding arc of a unit circle; one radian is just under 57.3 degrees (when the arc length is equal to the radius)

## References :

- <https://realpython.com/>
- <https://beginnersbook.com>
- <https://www.openbookproject.net/>
- <https://docs.python.org/>
- <https://www.javatpoint.com/>
- <https://www.w3schools.com/>
- <https://www.tutorialspoint.com/>
- <https://stackabuse.com/>
- <https://dotnettutorials.net/>
- <https://developers.google.com/>
- <https://www.techbeamers.com/>
- <https://www.guru99.com/>
- <https://www.geeksforgeeks.org/>



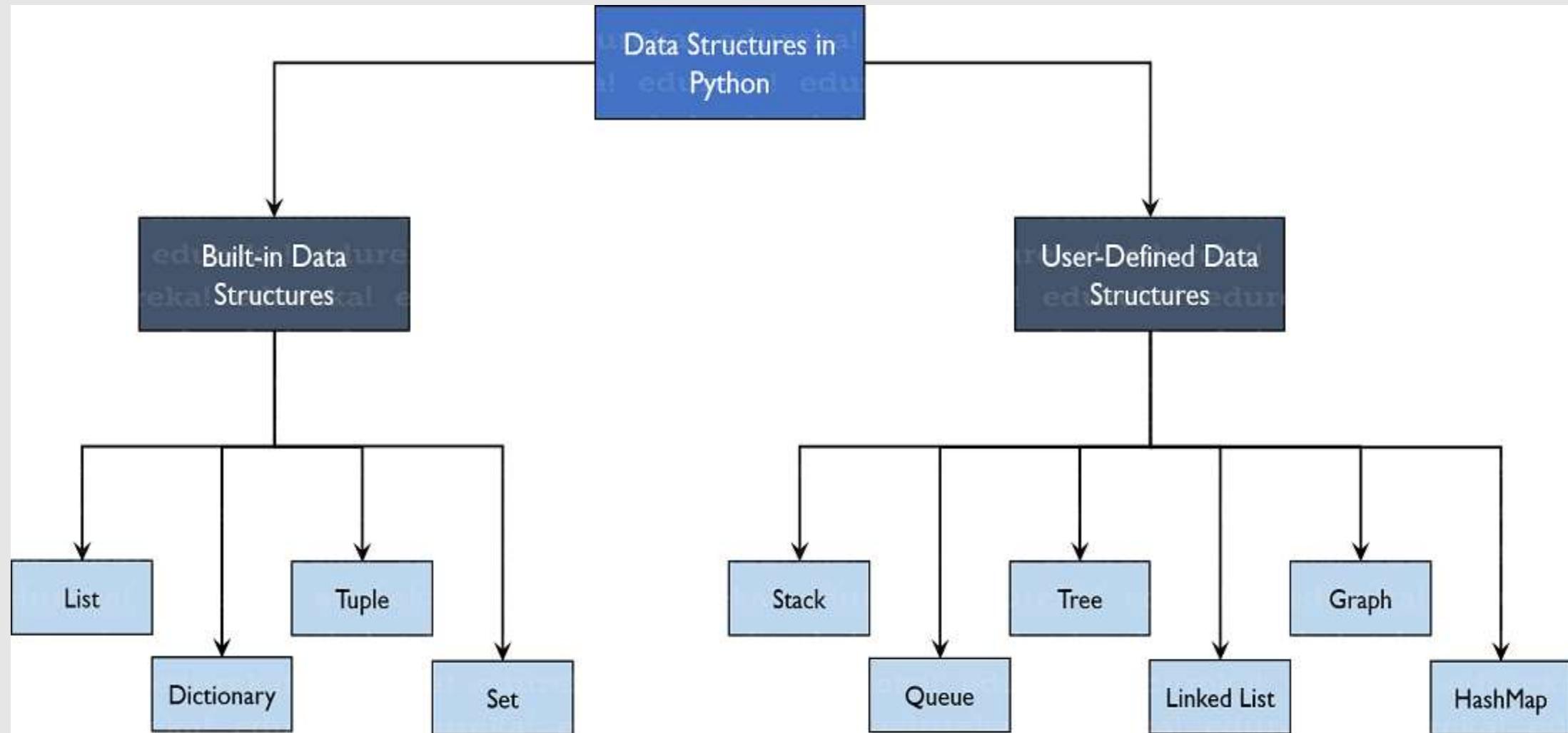
# DATA STRUCTURES IN PYTHON

Unit IV

# Introduction

- What is a Data Structure?
  - Organizing, managing and storing data is important as it enables easier access and efficient modifications.
  - Data Structures allows you to organize your data in such a way that enables you to store collections of data, relate them and perform operations on them accordingly.
- 
- **Types of Data Structures in Python**
  - Python has implicit support for Data Structures which enable you to store and access data.
  - These structures are called List, Dictionary, Tuple and Set.

# Introduction



# Lists

- Lists
- Lists are used to store data of different data types in a sequential manner. There are addresses assigned to every element of the list, which is called as Index.
- The index value starts from 0 and goes on until the last element called the positive index.
- There is also negative indexing which starts from -1 enabling you to access elements from the last to first.

# Lists

- Creating a list
- To create a list, you use the square brackets and add elements into it accordingly. If you do not pass any elements inside the square brackets, you get an empty list as the output.
- `my_list = [] #create empty list`
- `print(my_list)`
- `my_list = [1, 2, 3, 'example', 3.132] #creating list with data`
- `print(my_list)`
- **Output:**
- `[]`
- `[1, 2, 3, 'example', 3.132]`

# Lists

- Adding Elements
- Adding the elements in the list can be achieved using the append(), extend() and insert() functions.
- The append() function adds all the elements passed to it as a single element.
- The extend() function adds the elements one-by-one into the list.
- The insert() function adds the element passed to the index value and increase the size of the list too.
- **my\_list = [1, 2, 3]**
- **print(my\_list)**
- **my\_list.append([555, 12]) #add as a single element**
- **print(my\_list)**
- **my\_list.extend([234, 'more\_example']) #add as different elements**
- **print(my\_list)**
- **my\_list.insert(1, 'insert\_example') #add element i**
- **print(my\_list)**

# Lists

- Accessing Elements
- Accessing elements is the same as accessing Strings in Python. You pass the index values and hence can obtain the values as needed.
  
- `my_list = [1, 2, 3, 'example', 3.132, 10, 30]`
- `for element in my_list: #access elements one by one`
- `print(element)`
- `print(my_list) #access all elements`
- `print(my_list[3]) #access index 3 element`
- `print(my_list[0:2]) #access elements from 0 to 1 and exclude 2`
- `print(my_list[::-1]) #access elements in reverse`

# Lists

- Other Functions
- You have several other functions that can be used when working with lists.
- The **len()** function returns to us the length of the list.
- The **index()** function finds the index value of value passed where it has been encountered the first time.
- The **count()** function finds the count of the value passed to it.
- The **sorted()** and **sort()** functions do the same thing, that is to sort the values of the list. The sorted() has a return type whereas the sort() modifies the original list.

# Lists

- `my_list = [1, 2, 3, 10, 30, 10]`
- `print(len(my_list))` #find length of list
- `print(my_list.index(10))` #find index of element that occurs first
- `print(my_list.count(10))` #find count of the element
- `print(sorted(my_list))` #print sorted list but not change original
- `my_list.sort(reverse=True)` #sort original list
- `print(my_list)`
- Output:
  - 6
  - 3
  - 2
  - [1, 2, 3, 10, 10, 30]
  - [30, 10, 10, 3, 2, 1]

# Dictionary

- Dictionaries are used to store key-value pairs. To understand better, think of a phone directory where hundreds and thousands of names and their corresponding numbers have been added.
- Now the constant values here are Name and the Phone Numbers which are called as the keys.
- And the various names and phone numbers are the values that have been fed to the keys.
- If you access the values of the keys, you will obtain all the names and phone numbers.
- So that is what a key-value pair is.
- And in Python, this structure is stored using Dictionaries.

# Dictionary

- Creating a Dictionary
- Dictionaries can be created using the curly braces or using the dict() function..
- **my\_dict = {} #empty dictionary**
- **print(my\_dict)**
- **my\_dict = {1: 'Python', 2: 'Java'} #dictionary with elements**
- **print(my\_dict)**
- Output:
  - {}
  - {1: 'Python', 2: 'Java'}

# Dictionary

- Changing and Adding key, value pairs
  - To change the values of the dictionary, you need to do that using the keys. So, you firstly access the key and then change the value accordingly. To add values, you simply just add another key-value pair as shown below.
- 
- `my_dict = {'First': 'Python', 'Second': 'Java'}`
  - `print(my_dict)`
  - `my_dict['Second'] = 'C++' #changing element`
  - `print(my_dict)`
  - `my_dict['Third'] = 'Ruby' #adding key-value pair`
  - `print(my_dict)`

# Dictionary

- Deleting key, value pairs
- To delete the values, you use the `pop()` function which returns the value that has been deleted.
- To retrieve the key-value pair, you use the `popitem()` function which returns a tuple of the key and value.
- To clear the entire dictionary, you use the `clear()` function.
- **`my_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}`**
- **`a = my_dict.pop('Third') #pop element`**
- **`print('Value:', a)`**
- **`print('Dictionary:', my_dict)`**
- **`b = my_dict.popitem() #pop the key-value pair`**
- **`print('Key, value pair:', b)`**
- **`print('Dictionary', my_dict)`**
- **`my_dict.clear() #empty dictionary`**
- **`print('n', my_dict)`**

# Dictionary

- Accessing Elements
- You can access elements using the keys only. You can use either the get() function or just pass the key values and you will be retrieving the values.
  
- `my_dict = {'First': 'Python', 'Second': 'Java'}`
- `print(my_dict['First']) #access elements using keys`
- `print(my_dict.get('Second'))`
  
- **Output:**
- Python
- Java

# Dictionary

- **Other Functions**
- You have different functions which return to us the keys or the values of the key-value pair accordingly to the keys(), values(), items() functions accordingly.
- **my\_dict = {'First': 'Python', 'Second': 'Java', 'Third': 'Ruby'}**
- **print(my\_dict.keys()) #get keys**
- **print(my\_dict.values()) #get values**
- **print(my\_dict.items()) #get key-value pairs**
- **print(my\_dict.get('First'))**
- Output:
  - dict\_keys(['First', 'Second', 'Third'])
  - dict\_values(['Python', 'Java', 'Ruby'])
  - dict\_items([('First', 'Python'), ('Second', 'Java'), ('Third', 'Ruby')])
- Python

# Tuple

- Tuples are the same as lists are with the exception that the data once entered into the tuple cannot be changed no matter what.
- The only exception is when the data inside the tuple is mutable, only then the tuple data can be changed.
- Once a tuple is created, you cannot change its values. Tuples are unchangeable, or immutable as it also is called.
- But there is a workaround. You can convert the tuple into a list, change the list, and convert the list back into a tuple.

# Tuple

- Creating a Tuple
- You create a tuple using parenthesis or using the tuple() function.
- `my_tuple = (1, 2, 3) #create tuple`
- `print(my_tuple)`
- Output:
- `(1, 2, 3)`

# Tuple

- Accessing Elements
- Accessing elements is the same as it is for accessing values in lists.
  
- `my_tuple2 = (1, 2, 3, 'edureka') #access elements`
- `for x in my_tuple2:`
- `print(x)`
- `print(my_tuple2)`
- `print(my_tuple2[0])`
- `print(my_tuple2[:])`
- `print(my_tuple2[3][4])`

# Tuple

- Appending Elements
- To append the values, you use the ‘+’ operator which will take another tuple to be appended to it.
  
- `my_tuple = (1, 2, 3)`
- `my_tuple = my_tuple + (4, 5, 6) #add elements`
- `print(my_tuple)`
  
- **Output:**
- `(1, 2, 3, 4, 5, 6)`

# Tuple

- Other Functions
- These functions are the same as they are for lists.

- `my_tuple = (1, 2, 3, ['hindi', 'python'])`
- `my_tuple[3][0] = 'english'`
- `print(my_tuple)`
- `print(my_tuple.count(2))`
- `print(my_tuple.index(['english', 'python']))`

- **Output:**

- `(1, 2, 3, ['english', 'python'])`
- 1
- 3

# Tuple

- In Python, we are also allowed to extract the values back into variables. This is called "unpacking":
- Example
- Unpacking a tuple:
- fruits = ("apple", "banana", "cherry")
- (green, yellow, red) = fruits
- print(green)
- print(yellow)
- print(red)

# Tuple

- The number of variables must match the number of values in the tuple, if not, you must use an asterix to collect the remaining values as a list.
- Assign the rest of the values as a list called "red":
- fruits = ("apple", "banana", "cherry", "strawberry", "raspberry")
- (green, yellow, \*red) = fruits
  - print(green)
  - print(yellow)
  - print(red)

# Sets

- Sets
- Sets are a collection of unordered elements that are unique.
- Meaning that even if the data is repeated more than one time, it would be entered into the set only once. It resembles the sets that you have learnt in arithmetic.
- The operations also are the same as is with the arithmetic sets.
- Unordered means that the items in a set do not have a defined order.
- Set items can appear in a different order every time you use them, and cannot be referred to by index or key.
- Once a set is created, you cannot change its items, but you can add new items.

# Sets

- Creating a set
- Sets are created using the curly braces but instead of adding key-value pairs, you just pass values to it.
- `my_set = {1, 2, 3, 4, 5, 5, 5} #create set`
- `print(my_set)`
- Output:
- `{1, 2, 3, 4, 5}`

# Sets

- Adding elements
- To add elements, you use the add() function and pass the value to it.
  
- `my_set = {1, 2, 3}`
- `my_set.add(4) #add element to set`
- `print(my_set)`
  
- **Output:**
- `{1, 2, 3, 4}`

# Operations in sets

- The different operations on set such as union, intersection and so on are shown below.
- `my_set = {1, 2, 3, 4}`
- `my_set_2 = {3, 4, 5, 6}`
- `print(my_set.union(my_set_2), '-----', my_set | my_set_2)`
- `print(my_set.intersection(my_set_2), '-----', my_set & my_set_2)`
- `print(my_set.difference(my_set_2), '-----', my_set - my_set_2)`
- `print(my_set.symmetric_difference(my_set_2), '-----', my_set ^ my_set_2)`
- `my_set.clear()`
- `print(my_set)`

# Sets

- The union() function combines the data present in both sets.
- The intersection() function finds the data present in both sets only.
- The difference() function deletes the data present in both and outputs data present only in the set passed.
- The symmetric\_difference() does the same as the difference() function but outputs the data which is remaining in both sets.
- **Output:**
  - $\{1, 2, 3, 4, 5, 6\} \text{ ----- } \{1, 2, 3, 4, 5, 6\}$
  - $\{3, 4\} \text{ ----- } \{3, 4\}$
  - $\{1, 2\} \text{ ----- } \{1, 2\}$
  - $\{1, 2, 5, 6\} \text{ ----- } \{1, 2, 5, 6\}$
  - `set()`

# Set

- | Operator :
- This operator is used to return the union of two or more sets just like union() method. The difference between | operator and union() method is that, the former can work only with set objects while latter can work with any iterable objects like list, string, set.
- Syntax : < Set Object 1 > | < Set Object 2 > | < Set Object 2 >... : < Set Object >

# Set

- |= Operator :
- This operator is used to return the union of two or more sets just like update() method.
- The difference between |= operator and update() method is that, the former can work only with set objects while latter can work with any iterable objects like list, string, set.
- Syntax : < Set Object 1 > |= < Set Object 2 > | < Set Object 2 >...

# Set

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

# Set

- Example
- Remove "banana" by using the discard() method:
  - `thisset = {"apple", "banana", "cherry"}`
  - `thisset.discard("banana")`
  - `print(thisset)`
  - If the item to remove does not exist, discard() will NOT raise an error.

# Set

- The clear() method empties the set:
- `thisset = {"apple", "banana", "cherry"}`
- `thisset.clear()`
- `print(thisset)`

# Set

- The del keyword will delete the set completely:
- `thisset = {"apple", "banana", "cherry"}`
- `del thisset`
- `print(thisset)`

# Set

- You can use the **union()** method that returns a new set containing all items from both sets, or the **update()** method that inserts all the items from one set into another.
- Set1 = {1,2,3}
- Set2 ={'a','b'}
- Set3 = Set1.union(Set2)
  
- Set1.update(Set2)

# Set

- The **intersection\_update()** method will keep only the items that are present in both sets.
- Example
- Keep the items that exist in both set x, and set y:
  - `x = {"apple", "banana", "cherry"}`
  - `y = {"google", "microsoft", "apple"}`
  - `x.intersection_update(y)`
  - `print(x)`

# Set

- The **symmetric\_difference\_update()** method will keep only the elements that are NOT present in both sets.
- Example
- Keep the items that are not present in both sets:
- `x = {"apple", "banana", "cherry"}`
- `y = {"google", "microsoft", "apple"}`
- `x.symmetric_difference_update(y)`
- `print(x)`

# Set

- The **symmetric\_difference()** method will return a new set, that contains only the elements that are NOT present in both sets.
- Example
- Return a set that contains all items from both sets, except items that are present in both:
  - `x = {"apple", "banana", "cherry"}`
  - `y = {"google", "microsoft", "apple"}`
  - `z = x.symmetric_difference(y)`
  - `print(z)`

# Set

- The add() method adds an element to the set.
- If the element already exists, the add() method does not add the element.
- Syntax
- `set.add(ele)`

# Introduction

- Data structures organize storage in computers so that we can efficiently access the data and also change the data.
- Stacks and Queues are simple to learn and easy to implement.
- It's common for Stacks and Queues to be implemented with an Array or Linked List.
- Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out).

# Stack

- Stacks are linear Data Structures which are based on the principle of Last-In-First-Out (LIFO) where data which is entered last will be the first to get accessed.
- It is built using the array structure and has operations namely, pushing (adding) elements, popping (deleting) elements and accessing elements only from one point in the stack called as the TOP.
- This TOP is the pointer to the current position of the stack.

# Stacks

- In stack, a new element is added at one end and an element is removed from that end only.
- The insert and delete operations are often called push and pop.
- The functions associated with stack are:
  - **empty()** – Returns whether the stack is empty
  - **size()** – Returns the size of the stack
  - **top()** – Returns a reference to the top most element of the stack
  - **push(e)** – Adds the element 'e' at the top of the stack
  - **pop()** – Deletes the top most element of the stack

# Introduction

- Stack in Python can be implemented using following ways:
- list
- collections.deque
- queue.LifoQueue

# Introduction

- Implementation using list:
- Python's built-in data structure **list** can be used as a stack. Instead of `push()`, **append()** is used to add elements to the top of stack while **pop()** removes the element in LIFO order.
- Unfortunately, list has a few shortcomings.
- The biggest issue is that it can run into speed issue as it grows.
- The items in list are stored next to each other in memory, if the stack grows bigger than the block of memory that currently hold it, then Python needs to do some memory allocations. This can lead to some `append()` calls taking much longer than other ones.

# Introduction

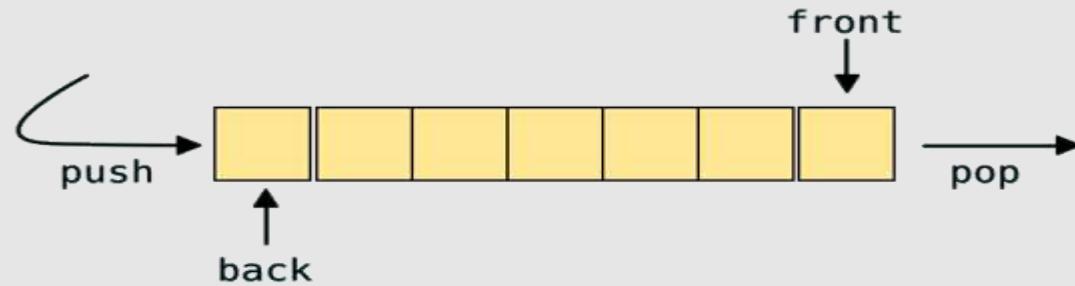
- # Python program to demonstrate stack implementation using list
- stack = []
- # append() function to push element in the stack
- stack.append('a')
- stack.append('b')
- stack.append('c')
- print('Initial stack')
- print(stack)

# Introduction

- # pop() function to pop element from stack in LIFO order
- print('\nElements popped from stack:')
- print(stack.pop())
- print(stack.pop())
- print(stack.pop())
  
- print('\nStack after elements are popped:')
- print(stack)
  
- # uncommenting print(stack.pop()) will cause an IndexError as the stack is now empty

# Queue

- A queue is also a linear data structure which is based on the principle of First-In-First-Out (FIFO) where the data entered first will be accessed first.
- It is built using the array structure and has operations which can be performed from both ends of the Queue, that is, head-tail or front-back.
- Operations such as adding and deleting elements are called En-Queue and De-Queue and accessing the elements can be performed.
- Queues are used as Network Buffers for traffic congestion management, used in Operating Systems for Job Scheduling and many more.



# Queue

- Queue works on the principle of “First-in, first-out”.
- Below is list implementation of queue.
- We use `pop(0)` to remove the first item from a list.
- An item that is added (`enqueue`) at the end of a queue will be the last one to be accessed (`dequeue`).

# Queue

- # Python code to demonstrate Implementing
- # Queue using list
- queue = ["A", "B", "C"]
- queue.append("D")
- queue.append("E")
- print(queue)
  
- # Removes the first item
- print(queue.pop(0))
  
- print(queue)

# Queue

- # Removes the first item
- `print(queue.pop(0))`
- `print(queue)`

# List Comprehension

- List comprehension offers a shorter syntax when you want to create a new list based on the values of an existing list.
- Example:
- Based on a list of fruits, you want a new list, containing only the fruits with the letter "a" in the name.
- Without list comprehension you will have to write a for statement with a conditional test inside:

# List Comprehension

- fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
- newlist = []
  
- for x in fruits:
- if "a" in x:
- newlist.append(x)
  
- print(newlist)

# List Comprehension

- With list comprehension you can do all that with only one line of code:
- Example
- fruits = ["apple", "banana", "cherry", "kiwi", "mango"]
- newlist = [x for x in fruits if "a" in x]
- print(newlist)

# List Comprehension

- **Syntax :**
- newlist = [expression for item in iterable if condition == True]
- The return value is a new list, leaving the old list unchanged.
- Condition
- The condition is like a filter that only accepts the items that evaluate to True.

# List Comprehension

- Example :
- Only accept items that are not "apple":
- newlist = [x for x in fruits if x != "apple"]
- The condition if x != "apple" will return True for all elements other than "apple", making the new list contain all fruits except "apple".
- The condition is optional and can be omitted:

## References :

- <https://realpython.com/>
- <https://beginnersbook.com>
- <https://www.openbookproject.net/>
- <https://docs.python.org/>
- <https://www.javatpoint.com/>
- <https://www.w3schools.com/>
- <https://www.tutorialspoint.com/>
- <https://stackabuse.com/>
- <https://dotnettutorials.net/>
- <https://developers.google.com/>
- <https://www.techbeamers.com/>
- <https://www.guru99.com/>
- <https://www.geeksforgeeks.org/>



# FILE HANDLING IN PYTHON

Unit V

# Introduction

- Python provides us with an important feature for reading data from the file and writing data into a file.
- Mostly, in programming languages, all the values or data are stored in some variables which are volatile in nature.
- Because data will be stored into those variables during run-time only and will be lost once the program execution is completed, it is better to save these data permanently using files.

# Introduction

- **Types Of File in Python**
- There are two types of files in Python and each of them are explained below in detail with examples for your easy understanding.
  
- They are:
- Binary file
- Text file
  
- Binary files in Python
- Most of the files that we see in our computer system are called binary files.

# Introduction

- Example:
- Document files: .pdf, .doc, .xls etc.
- Image files: .png, .jpg, .gif, .bmp etc.
- Video files: .mp4, .3gp, .mkv, .avi etc.
- Audio files: .mp3, .wav, .mka, .aac etc.
- Database files: .mdb, .accde, .frm, .sqlite etc.
- Archive files: .zip, .rar, .iso, .7z etc.
- Executable files: .exe, .dll, .class etc.

# Introduction

- All binary files follow a specific format. We can open some binary files in the normal text editor, but we can't read the content present inside the file. That's because all the binary files will be encoded in the binary format, which can be understood only by a computer or machine.
- For handling such binary files we need a specific type of software to open it.
- For Example, You need Microsoft word software to open .doc binary files. Likewise, you need a pdf reader software to open .pdf binary files and you need a photo editor software to read the image files and so on.

# Python File Handling Operations

- Most importantly there are 4 types of operations that can be handled by Python on files:
- **Open**
- **Read**
- **Write**
- **Close**
- Other operations include:
- Rename
- Delete

# Python File Handling Operations

- Python Create and Open a File
- Python has an in-built function called **open()** to open a file.
- It takes a minimum of one argument as mentioned in the below syntax. The open method returns a file object which is used to access the write, read and other in-built methods.
- Syntax:
- `file_object = open(file_name, mode)`

# Python File Handling Operations

- Here, file\_name is the name of the file or the location of the file that you want to open, and file\_name should have the file extension included as well.
- For e.g. in example.txt – the term example is the name of the file and .txt is the extension of the file.

# Python File Handling Operations

- The mode in the open function syntax will tell Python as what operation you want to do on a file.
- ‘**r**’ – Read Mode: Read mode is used only to read data from the file.
- ‘**w**’ – Write Mode: This mode is used when you want to write data into the file or modify it. Remember write mode overwrites the data present in the file.
- ‘**a**’ – Append Mode: Append mode is used to append data to the file. Remember data will be appended at the end of the file pointer.
- ‘**r+**’ – Read or Write Mode: This mode is used when we want to write or read the data from the same file.
- ‘**a+**’ – Append or Read Mode: This mode is used when we want to read data from the file or append the data into the same file.
- Note: The above-mentioned modes are for opening, reading or writing text files only.

# Python File Handling Operations

- While using binary files, we have to use the same modes with the letter ‘b’ at the end. So that Python can understand that we are interacting with binary files.
- ‘**wb**’ – Open a file for write only mode in the binary format.
- ‘**rb**’ – Open a file for the read-only mode in the binary format.
- ‘**ab**’ – Open a file for appending only mode in the binary format.
- ‘**rb+**’ – Open a file for read and write only mode in the binary format.
- ‘**ab+**’ – Open a file for appending and read-only mode in the binary format.

# Python File Handling Operations

- Here's an example of Python open function and Python readlines for reading files line by line.
- Here's a code snippet to open the file using file handing in Python.
  - `f= open('demofile.txt', 'r')`
  - `f.readline()`
- With the help of the open function of Python read text file, saves it in a file object and read lines with the help of the readline function. Remember that `f.readline()` reads a single line from the file object. Also, this function leaves a newline character (`\n`) at the end of the string.

# Python File Handling Operations

- Writing into a File
- The write() method is used to write a string into a file.
  
- Syntax of the Python write function:
- File\_object.write("string")
- Example:
  
- `i=open("demotext.txt","w")`
- `i.write("Hello Python")`
- Here, we are opening the demotext.txt file into a file object called 'i'. Now, we can use the write function in order to write something into the file.

# Python File Handling Operations

- Reading from a File
- The read() method is used to read data from a file.
  
- Syntax of the Python read function:
- File\_object.read(data)
- Example:
  
- `j=open("demo.txt","r")`
- `k=j.read()`
- `print(k)`
- Output:
  
- Hello Python

# Python File Handling Operations

- Closing a File
- The close() function is used to close a file.
- Syntax of the Python close function:
- File\_object.close()
- Example:
- `j=open("demo.txt","r")`
- `k=j.read()`
- `print(k)`
- `j.close()`
- Output:
- Hello Python

# The write() Method

The write() method writes any string to an open file.

The write() method does not add a newline character ('\n') to the end of the string –

## Syntax

```
fileObject.write(string)
```

Here, passed parameter is the content to be written into the opened file.

## # Open a file

```
fo = open("demo.txt", "wb")
```

```
fo.write( "Python is a great language.\nYeah its great!!\n")
```

## # Close open file

```
fo.close()
```

# Renaming and Deleting Files

Python os module provides methods that help you perform file-processing operations, such as renaming and deleting files.

To use this module you need to import it first and then you can call any related functions.

## The **rename()** Method

The rename() method takes two arguments, the current filename and the new filename.

Syntax

```
os.rename(current_file_name, new_file_name)
```

# Renaming and Deleting Files

```
import os  
  
# Rename a file from test1.txt to test2.txt  
  
os.rename( "test1.txt", "test2.txt" )
```

## The **remove()** Method

You can use the **remove()** method to delete files by supplying the name of the file to be deleted as the argument.

### Syntax

**os.remove(file\_name)**

E.g. **os.remove("text2.txt")**

# Renaming and Deleting Files

## **The mkdir() Method**

You can use the `mkdir()` method of the `os` module to create directories in the current directory. You need to supply an argument to this method which contains the name of the directory to be created.

### Syntax

```
os.mkdir("newdir")
```

# Methods of File Handling in Python

- There are different file handling in Python which are as follows:
- **rename():** This is used to rename a file.
  - import os
  - os.rename(existing\_file\_name, new\_file\_name)
- **remove():** This method is used to delete a file in Python.
  - import os
  - os.remove("abc.txt")
- **chdir():** This method is used to change the current directory.
  - import os
  - os.chdir("new directory path")

# Methods of File Handling in Python

- **mkdir():** This method is used to create a new directory.
  - import os
  - os.mkdir("new directory path ")
- **rmdir():** This method is used to remove a directory.
  - import os
  - os.rmdir("new directory path")
- **getcwd():** This method is used to show the current working directory.
  - import os
  - print(os.getcwd())

# Other Methods of File Handling in Python

Method	Description
close()	To close an open file. It has no effect if the file is already closed
flush()	To flush the write buffer of the file stream
read(n)	To read at most <i>n</i> characters from a file. Remember that it reads till end of file if it is <b>negative or none</b>
readline( <i>n</i> =-1)	To read and return one line from a file. Remember that it reads at most <i>n</i> bytes, if specified
readlines( <i>n</i> =-1)	To read and return a list of lines from a file. Remember that it reads at most <i>n</i> bytes/characters if specified
seek( <i>offset,from=SEEK_SET</i> )	It changes the file position to <i>offset</i> bytes, in reference to (start, current, or end)
tell()	It returns the current file location
writable()	It returns true if the file stream can be written to
write(s)	To write string <i>s</i> into a file and return the number of characters written
writelines(lines)	To write a list of lines into a file

# Logging in Python

- Logging is a means of tracking events that happen when some software runs.
- Logging is important for software developing, debugging and running.
- If you don't have any logging record and your program crashes, there are very little chances that you detect the cause of the problem.
- And if you detect the cause, it will consume a lot of time.
- With logging, you can leave a trail of breadcrumbs so that if something goes wrong, we can determine the cause of the problem.
- Python has a built-in module logging which allows writing status messages to a file or any other output streams.
- The file can contain the information on which part of the code is executed and what problems have been arisen.

# Levels of Log Message

- There are built-in levels of the log message.
- Debug : These are used to give Detailed information, typically of interest only when diagnosing problems.
- Info : These are used to Confirm that things are working as expected
- Warning : These are used as an indication that something unexpected happened, or indicative of some problem in the near future
- Error : This tells that due to a more serious problem, the software has not been able to perform some function
- Critical : This tells serious error, indicating that the program itself may be unable to continue running.

# Levels of Log Message

- If required, developers have the option to create more levels but these are sufficient enough to handle every possible situation. Each built-in level has been assigned its numeric value.

Level	Numeric Value
NOTSET	0
DEBUG	10
INFO	20
WARNING	30
ERROR	40
CRITICAL	50

# Levels of Log Message

- Logging module is packed with several features. It has several constants, classes, and methods. The items with all caps are constant, the capitalize items are classes and the items which start with lowercase letters are methods.
  - There are several logger objects offered by the module itself.
- 
- **Logger.info(msg)** : This will log a message with level INFO on this logger.
  - **Logger.warning(msg)** : This will log a message with level WARNING on this logger.
  - **Logger.error(msg)** : This will log a message with level ERROR on this logger.
  - **Logger.critical(msg)** : This will log a message with level CRITICAL on this logger.
  - **Logger.log(lvl,msg)** : This will Logs a message with integer level lvl on this logger.
  - **Logger.exception(msg)** : This will log a message with level ERROR on this logger.
  - **Logger.setLevel(lvl)** : This function sets the threshold of this logger to lvl. This means that all the messages below this level will be ignored.

# Levels of Log Message

- **Logger.addFilter(filt)** : This adds a specific filter filt to the to this logger.
- **Logger.removeFilter(filt)** : This removes a specific filter filt to the to this logger.
- **Logger.filter(record)** : This method applies the logger's filter to the record provided and returns True if record is to be processed. Else, it will return False.
- **Logger.addHandler(hdlr)** : This adds a specific handler hdlr to the to this logger.
- **Logger.removeHandler(hdlr)** : This removes a specific handler hdlr to the to this logger.
- **Logger.hasHandlers()** : This checks if the logger has any handler configured or not.

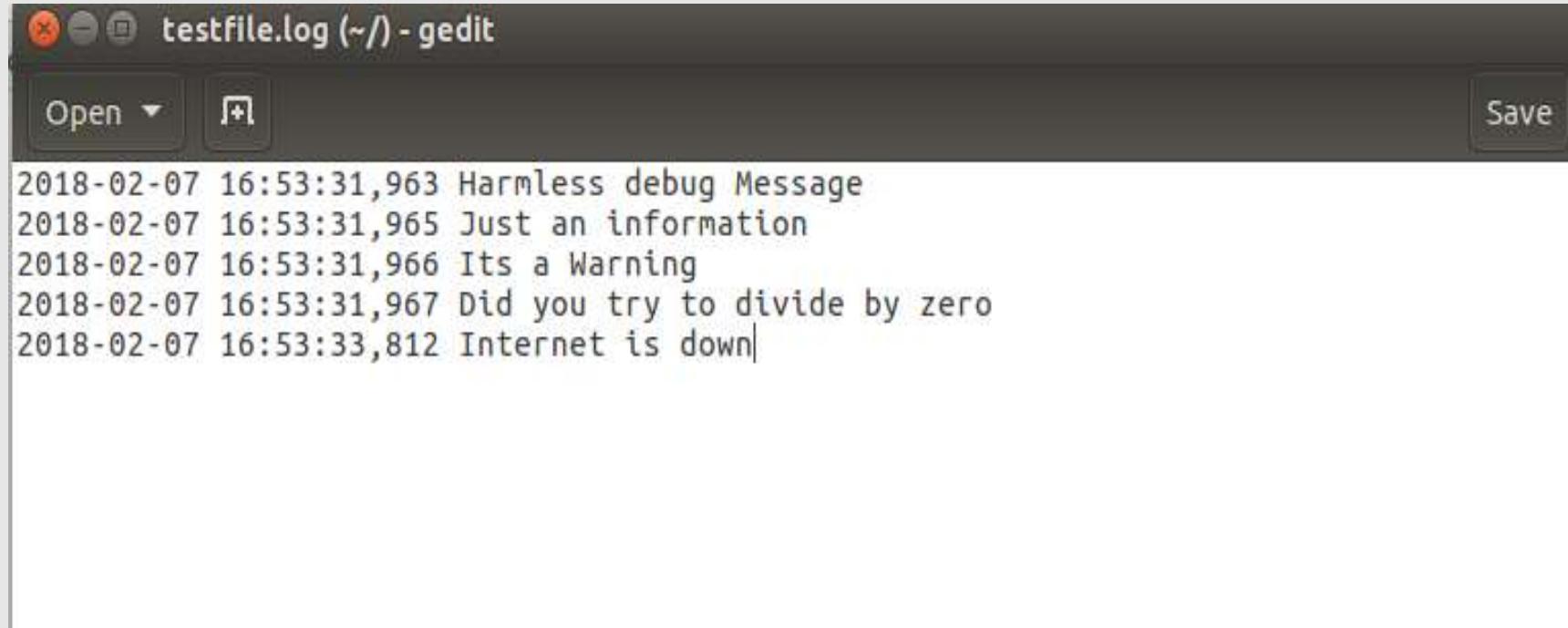
# Levels of Log Message

- Basics of using the logging module to record the events in a file are very simple.
- For that, simply import the module from the library.
  
- Create and configure the logger. It can have several parameters. But importantly, pass the name of the file in which you want to record the events.
- Here the format of the logger can also be set. By default, the file works in append mode but we can change that to write mode if required.
- Also, the level of the logger can be set which acts as the threshold for tracking based on the numeric values assigned to each level.
- There are several attributes which can be passed as parameters.
- The list of all those parameters is given in Python Library. The user can choose the required attribute according to the requirement.
- After that, create an object and use the various methods as shown in the example.

# Levels of Log Message

- #importing module
- import logging
- #Create and configure logger
- logging.basicConfig(filename="newfile.log", format='%(asctime)s %(message)s', filemode='w')
- #Creating an object
- logger=logging.getLogger()
- #Setting the threshold of logger to DEBUG
- logger.setLevel(logging.DEBUG)
- #Test messages
- logger.debug("Harmless debug Message")
- logger.info("Just an information")
- logger.warning("Its a Warning")
- logger.error("Did you try to divide by zero")
- logger.critical("Internet is down")

# Levels of Log Message

A screenshot of the gedit text editor window titled "testfile.log (~/) - gedit". The window has a dark theme. The menu bar includes "File", "Edit", "View", "Search", "Format", "Tools", and "Help". The toolbar contains "Open", "Save", and other icons. The main text area displays the following log entries:

```
2018-02-07 16:53:31,963 Harmless debug Message
2018-02-07 16:53:31,965 Just an information
2018-02-07 16:53:31,966 Its a Warning
2018-02-07 16:53:31,967 Did you try to divide by zero
2018-02-07 16:53:33,812 Internet is down|
```

The cursor is positioned at the end of the last line.

# References :

- <https://realpython.com/>
- <https://beginnersbook.com>
- <https://www.openbookproject.net/>
- <https://docs.python.org/>
- <https://www.javatpoint.com/>
- <https://www.w3schools.com/>
- <https://www.tutorialspoint.com/>
- <https://stackabuse.com/>
- <https://dotnettutorials.net/>
- <https://developers.google.com/>
- <https://www.techbeamers.com/>
- <https://www.guru99.com/>
- <https://www.geeksforgeeks.org/>
- <https://www.softwaretestinghelp.com/>
- <https://intellipaat.com/>



# OBJECT ORIENTED PROGRAMMING IN PYTHON

Unit VI

# Introduction

- Object Oriented Programming is a way of computer programming using the idea of “objects” to represents data and methods.
- It is also, an approach used for creating neat and reusable code instead of a redundant one. the program is divided into self-contained objects or several mini-programs.
- Every Individual object represents a different part of the application having its own logic and data to communicate within themselves.

# Difference between Object-Oriented and Procedural Oriented Programming

Object-Oriented Programming (OOP)	Procedural-Oriented Programming (Pop)
Bottom-up approach	Top down approach
Program is divided into objects	Program is divided into functions
Makes use of access specifiers	Does not use access specifiers
It is more secure	It is less secure
Object can move freely within member functions	Data can move freely from function to function within programs
It supports inheritance	It does not support inheritance

# Object oriented programming concepts

- Major OOP (object-oriented programming) concepts in Python include Class, Object, Method, Inheritance, Polymorphism, Data Abstraction, and Encapsulation.
- A class is a collection of objects or you can say it is a blueprint of objects defining the common attributes and behavior.
- Objects are an instance of a class. It is an entity that has state and behavior. In a nutshell, it is an instance of a class that can access the data.

# Object oriented programming concepts

- The first method `__init__()` is a special method, which is called class constructor or initialization method that Python calls when you create a new instance of this class.
- Starting an attribute or method name with an underscore (`_`) is a convention which we use to indicate that it is a “private” internal property and should not be accessed directly.
- Initialising all our attributes in `__init__`, even if we just set them to empty values, makes our code less error-prone. It also makes it easier to read and understand – we can see at a glance what attributes our object has.
- An `__init__` method doesn’t have to take any parameters (except `self`) and it can be completely absent.

# Object oriented programming concepts

- class Employee :
- def \_\_init\_\_(self, name, age, id, salary):
- self.name = name
- self.age = age
- self.salary = salary
- self.id = id
  
- Emp = Employee(EmpName,EmpAge,Empld,EmpSalary)

# Object oriented programming concepts

- The self parameter is a reference to the current instance of the class, and is used to access variables that belongs to the class.
- It does not have to be named self , you can call it whatever you like, but it has to be the first parameter of any function in the class.

# Object oriented programming concepts

- You can modify properties on objects like this:
- Set the age of Employee to 40:
- `emp.age = 40`
- You can delete properties on objects by using the del keyword:
- Delete the age property from the Employee object:
- `del emp.age`

# Object oriented programming concepts

- You can delete objects by using the del keyword:
- Example
- Delete the emp object:
- `del emp`

# Object oriented programming concepts

- **Class attributes** are the variables defined directly in the class that are shared by all objects of the class.
- **Instance attributes** are attributes or properties attached to an instance of a class. Instance attributes are defined in the constructor.
- class Student:
  - count = 0
  - def \_\_init\_\_(self):
    - Student.count += 1

# Object oriented programming concepts

- count is an attribute in the Student class. Whenever a new object is created, the value of count is incremented by 1.
  
- std1=Student()
- Student.count
- 1
  
- std2 = Student()
- Student.count
- 2

# Object oriented programming concepts

- We can also set default values to the instance attributes. The following code sets the default values of the constructor parameters. So, if the values are not provided when creating an object, the values will be assigned latter.
- Example: Setting Default Values of Attributes Copy
- class Student:
- def \_\_init\_\_(self, name="Guest", age=25)
- self.name=name
- self.age=age

# Object oriented programming concepts

- class Student:
- def \_\_init\_\_(self, name="Guest", age=25)
- self.name=name
- self.age=age
- **Output**
- std = Student()
- std.name
- 'Guest'
- std.age
- 25

# Object oriented programming concepts

- The private variables don't actually exist in Python.
- There are simply norms to be followed.
- The language itself doesn't apply any restrictions.

# Object oriented programming concepts

- **Class Properties**
- The @property decorator is a built-in decorator in Python for the property() function. Use @property decorator on any method in the class to use the method as a property.
- You can use the following three decorators to define a property:
  - @property: Declares the method as a property.
  - @<property-name>.setter: Specifies the setter method for a property that sets the value to a property.
  - @<property-name>.deleter: Specifies the delete method as a property that deletes a property.

# Object oriented programming concepts

- The following declares the method as a property. This method must return the value of the property.
- class Student:
  - def \_\_init\_\_(self, name):
  - self.\_\_name = name
  - @property
  - def name(self):
  - return self.\_\_name

# Object oriented programming concepts

- Above, @property decorator applied to the name() method.
  - The name() method returns the private instance attribute value \_\_name.
  - So, we can now use the name() method as a property to get the value of the \_\_name attribute, as shown below.
- 
- >>> s = Student('Anjali')
  - >>> s.name
  - 'Anjali'

# Object oriented programming concepts

- **Property Setter**
- Above, we defined the name() method as a property.
- We can only access the value of the name property but cannot modify it.
- To modify the property value, we must define the setter method for the name property using @property-name.setter decorator, as shown below.

# Object oriented programming concepts

- **class Student:**
- **def \_\_init\_\_(self, name):**
- **self.\_\_name=name**
- **@property**
- **def name(self):**
- **return self.\_\_name**
- **@name.setter #property-name.setter decorator**
- **def name(self, value):**
- **self.\_\_name = value**

# Object oriented programming concepts

- Above, we have two overloads of the name() method.
- One is for the getter and another is the setter method.
- The setter method must have the value argument that can be used to assign to the underlying private attribute.
- Now, we can retrieve and modify the property value, as shown below.
- `>>> s = Student('Anjali')`
- `>>> s.name`
- `'Anjali'`
- `>>> s.name = 'Aparna'`
- `'Aparna'`

# Object oriented programming concepts

- **Property Deleter**
- Use the @property-name.deleter decorator to define the method that deletes a property, as shown below.
- class Student:
- def \_\_init\_\_(self, name):
- self.\_\_name = name
- @property
- def name(self):
- return self.\_\_name
- @name.setter
- def name(self, value):
- self.\_\_name=value
- @name.deleter #property-name.deleter decorator
- def name(self, value):
- print('Deleting..')
- del self.\_\_name

# Object oriented programming concepts

- class Student:
- def \_\_init\_\_(self):
- self.\_\_name=""
- def setname(self, name):
- print('setname() called')
- self.\_\_name=name
- def getname(self):
- print('getname() called')
- return self.\_\_name
- name=property(getname, setname)
  
- s1 = Student()
- s1.name = 'Anjali'
- print(s1.name)

# Object oriented programming concepts

- In the above example, property(getname, setname) returns the property object and assigns it to name.
- Thus, the name property hides the private instance attribute \_\_name.
- The name property is accessed directly, but internally it will invoke the getname() or setname() method

# Object oriented programming concepts

- Object-Oriented Programming methodologies deal with the following concepts.
- Inheritance
- Polymorphism
- Encapsulation
- Abstraction

# Inheritance

- Inheritance allows programmers to create classes that are built upon existing classes, and this enables a class created through inheritance to inherit the attributes and methods of the parent class.
- This means that inheritance supports code reusability.
- The child class can add a few more definitions or redefine a base class method.
- The class from which a class inherits is called the **parent** or **superclass**.
- A class which inherits from a superclass is called a **subclass**, also called **heir** class or **child** class.
- **Superclasses** are sometimes called **ancestors** as well.
- There exists a hierarchical relationship between classes.

# Inheritance

- The syntax for a subclass definition looks like this:
  
- class DerivedClassName(BaseClassName):
- any methods/variables of the class
  
- **\_\_init\_\_( ) Function**
- The \_\_init\_\_() function is called every time a class is being used to make an object.
- When we add the \_\_init\_\_() function in a parent class, the child class will no longer be able to inherit the parent class's \_\_init\_\_() function.
- The child's class \_\_init\_\_() function overrides the parent class's \_\_init\_\_() function.

# Inheritance

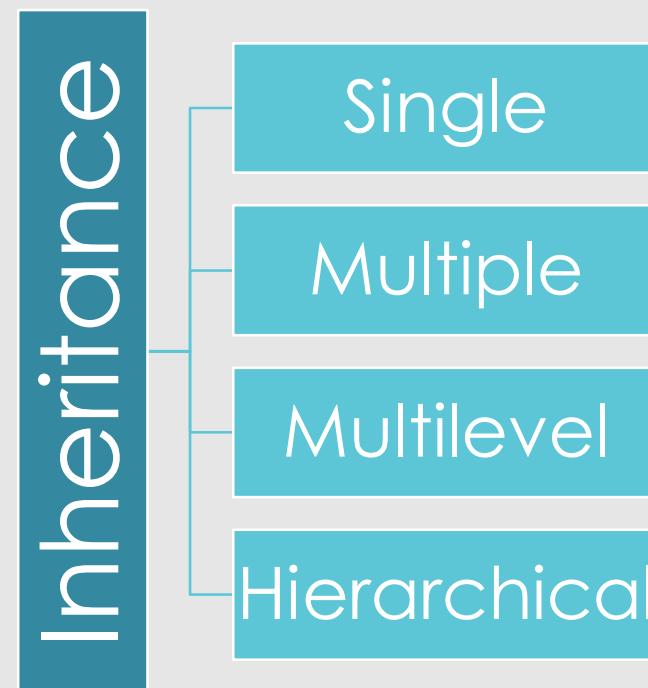
- class Robot:
- def \_\_init\_\_(self, name):
- self.name = name
- def say\_hi(self):
- print("Hi, I am " + self.name)
- class PhysicianRobot(Robot):
- pass
- x = Robot("Sachin")
- y = PhysicianRobot("Arjun")
- print(x, type(x))
- print(y, type(y))
- y.say\_hi()

# Inheritance

- class Parent:
- def \_\_init\_\_(self , fname, fage):
- self.firstname = fname
- self.age = fage
- def view(self):
- print(self.firstname , self.age)
- class Child(Parent):
- def \_\_init\_\_(self , fname , fage):
- Parent.\_\_init\_\_(self, fname, fage)
- self.lastname = "SURNAME"
- def view(self):
- print(f"First name : {self.firstname} ,Age : {self.age} , and Last name: {self.lastname} ")
- ob = Child("FIRST" , '22')
- ob.view()

# Inheritance

- **Types of Inheritance in Python**
- Types of Inheritance depends upon the number of child and parent classes involved. There are four types of inheritance in Python:



# Inheritance

- **Single Inheritance**
- When a child class inherits only a single parent class.
- class Parent:
  - def func1(self):
  - print("this is function one")
- class Child(Parent):
  - def func2(self):
  - print(" this is function 2 ")
- ob = Child()
- ob.func1()
- ob.func2()

# Inheritance

- **Multiple Inheritance**
- When a child class inherits from more than one parent class.
- class Parent:
- def func1(self):
- print("this is function 1")
- class Parent2:
- def func2(self):
- print("this is function 2")
- class Child(Parent , Parent2):
- def func3(self):
- print("this is function 3")

# Inheritance

- ob = Child()
- ob.func1()
- ob.func2()
- ob.func3()

# Inheritance

- Multilevel Inheritance
- When a child class becomes a parent class for another child class.
- `class Parent:`
- `def func1(self):`
- `print("this is function 1")`
- `class Child(Parent):`
- `def func2(self):`
- `print("this is function 2")`
- `class Child2(Child):`
- `def func3("this is function 3")`

# Inheritance

- ob = Child2()
- ob.func1()
- ob.func2()
- ob.func3()

# Inheritance

- Hierarchical Inheritance
- Hierarchical inheritance involves multiple inheritance from the same base or parent class.
- class Parent:
  - def func1(self):
    - print("this is function 1")
- class Child(Parent):
  - def func2(self):
    - print("this is function 2")
- class Child2(Parent):
  - def func3(self):
    - print("this is function 3")

# Inheritance

- ob = Child()
- ob1 = Child2()
- ob.func1()
- ob.func2()

# Inheritance

- **Super() Function**
- Super function allows us to call a method from the parent class.
  
- class Parent:
- def func1(self):
- print("this is function 1")
- class Child(Parent):
- def func2(self):
- Super().func1()
- print("this is function 2")
- ob = Child()
- ob.func2()

# Inheritance

- Methods of the parent class are available for use in the inherited class. However, if needed, we can modify the functionality of any base class method.
- For that purpose, the inherited class contains a new definition of a method (with the same name and the signature already present in the base class).
- Naturally, the object of a new class will have access to both methods, but the one from its own class will have precedence when invoked.
- This is called **method overriding**.

# Abstract classes

- In fact, Python on its own doesn't provide abstract classes.
- Yet, Python comes with a module which provides the infrastructure for defining Abstract Base Classes (ABCs).
- This module is called **abc**.
- A class that is derived from an abstract class cannot be instantiated unless all of its abstract methods are overridden.
- ABC works by decorating methods of the base class as abstract and then registering concrete classes as implementations of the abstract base.
- A method becomes abstract when decorated with the keyword `@abstractmethod`.

# Abstract classes

- To consider any class as an abstract class, the class has to inherit ABC metaclass from the python built-in abc module.
  - abc module imports the ABC metaclass.
  - Abstract methods are the methods that are declared without any implementations.
- 
- from abc import ABC, abstract method
  - Class Absclass(ABC):
  - @abstractmethod
  - def mymethod(self):
  - #empty body
  - pass

## Abstract classes

- ABC module is used to create the abstract classes, @abstractmethod is the decorator used to declare the method abstract.
- ABC module establishes a contract between the base class and the concrete class.

# Polymorphism

- Polymorphism defines the ability to take different forms.
- Polymorphism in Python allows us to define methods in the child class with the same name as defined in their parent class.
- Method Overloading, a way to create multiple methods with the same name but different arguments, is not possible in Python.
- Python does not support method overloading like Java or C++. We may overload the methods, but we can only use the latest defined method.
- We need to provide optional arguments or \*args in order to provide a different number of arguments on calling. But this is not the most efficient way.

# Polymorphism

- By Using Multiple Dispatch Decorator
- Multiple Dispatch Decorator Can be installed by:
- pip3 install multipledispatch
- In Backend, Dispatcher creates an object which stores different implementation and on runtime, it selects the appropriate method as the type and number of parameters passed.

# Polymorphism

- Built-in implementation of Polymorphism
  - **a. Polymorphism in ‘+’ operator**
- 
- You might have used the ‘+’ arithmetic python operator multiple times in your programs.
  - The same + symbol is used when you want to add two integers, or concatenate two strings, or extend two lists.
  - The + operator acts differently depending on the type of objects it is operating upon.
  - For integers, it performs arithmetic addition and returns an integer:
- 
- `>>> x = 1 + 2`
  - `>>> print(x)`

# Polymorphism

- **b. Polymorphism in built-in method**
  - Python also implements polymorphism using methods.
  - For instance, take the `len()` method that returns the length of an object.
  - The `len()` method is capable of processing objects of different data types.
    - # `len()` being used for a string
    - `print(len("python"))`
    - # `len()` being used for a list
    - `print(len([1, 2, 3]))`

# Polymorphism

- User defined polymorphism
- # A simple Python function to demonstrate Polymorphism
- ```
def add(p, q, r = 0):  
    return p + q+ r
```
- # Driver code
- ```
print(add(4, 5))
```
- ```
print(add(2, 5, 7))
```

# Polymorphism

- Polymorphism in user-defined methods
- In the below example, we have two classes ‘Rectangle’ and ‘Square’.
- Both these classes have a method definition ‘area’, that calculates the area of the corresponding shapes.
- class Rectangle:
  - def \_\_init\_\_(self, length, breadth):
  - self.l = length
  - self.b = breadth
  - def area(self):
  - return self.l \* self.b

# Polymorphism

- class Square:
- def \_\_init\_\_(self, side):
- self.s = side
- def area(self):
- return self.s \*\* 2
- rec = Rectangle(10, 20)
- squ = Square(10)
- print("Area of rectangle is: ", rec.area())
- print("Area of square is: ", squ.area())

# Polymorphism

- We can also achieve polymorphism with inheritance.
- A child class inherits all the attributes and methods of its parent class.
- But we can provide one or more methods with a different method definition within the child class.
- We call this process “method overriding” and such methods “overridden” methods.
- So overridden methods have the exact same external interface (method name, number and type of method parameters) as the parent class’s method but have a different internal implementation.

# Polymorphism

- **object - The Base Class**
- In Python, all classes inherit from the object class implicitly. It means that the following two class definitions are equivalent.
  
- class MyClass:
- pass
  
- class MyClass(object):
- pass
- It turns out that the object class provides some special methods with two leading and trailing underscores which are inherited by all the classes.

# Polymorphism

- `__new__()`
  - `__init__()`
  - `__str__()`
- 
- The `__new__()` method creates the object. After creating the object it calls the `__init__()` method to initialize attributes of the object.
  - Finally, it returns the newly created object to the calling program. Normally, we don't override the `__new__()` method, however, if you want to significantly change the way an object is created, you should definitely override it.

# Polymorphism

- The `__str__()` method is used to return a nicely formatted string representation of the object. The object class version of the `__str__()` method returns a string containing the name of the class and its memory address in hexadecimal. For example:
- class Smile:
- def laugh(self):
- return print("laugh() called")
- obj = Smile()
- print(obj)

# Polymorphism

- class Smile:
- def laugh(self):
- return "laugh() called"
- def \_\_str\_\_(self):
- return "A more helpful description"
- obj = Smile()
- print(obj)

# Polymorphism

- **Polymorphism with a Function and objects:**
- It is also possible to create a function that can take any object, allowing for polymorphism.
- In this example, let's create a function called "state()" which will take an object which we will name "obj".
- Though we are using the name 'obj', any instantiated object will be able to be called into this function.

# Polymorphism

- In this case lets call the three methods, viz., capital(), language() and famousfor(), each of which is defined in the two classes ‘Maharashtra’ and ‘Gujrat’.
- Now create instantiations of both the ‘Maharashtra’ and ‘Gujrat’ classes.
- With those, we can call their action using the same func() function:

# Polymorphism

- class Maharashtra():
  - def capital(self):
    - print("Mumbai is the capital of Maharashtra.")
  - def language(self):
    - print("Marathi is the most widely spoken language of Maharashtra.")
  - def famousfor(self):
    - print("Maharashtra is famous for its culture.")

# Polymorphism

- class Gujrati():
  - def capital(self):
    - print("Gandhinagar is the capital of Gujrati.")
  - def language(self):
    - print("Gujrati is the primary language of Gujrati.")
  - def famousfor(self):
    - print("Gujrat is a developing state.")

# Polymorphism

- def state(obj):
  - obj.capital()
  - obj.language()
  - obj.famousfor()
- obj\_mah = Maharastra()
- obj\_guj = Gujrat()
  
- state(obj\_mah)
- state(obj\_guj)

# Polymorphism : Operator Overloading

- Operator Overloading lets you redefine the meaning of operator respective to your class.
- It is the magic of operator overloading that we were able to use to + operator to add two numbers objects, as well as two string objects.
- When used with numbers it is interpreted as an addition operator whereas with strings it is interpreted as concatenation operator.
- In other words, we can say that + operator is overloaded for int class and str class.

# Polymorphism : Operator Overloading

- Operator Overloading is achieved by defining a special method in the class definition.
- The names of these methods starts and ends with double underscores (`__`).
- The special method used to overload the `+` operator is called `__add__()`.
- Both `int` class and `str` class implements `__add__()` method.
- The `int` class version of the `__add__()` method simply adds two numbers whereas the `str` class version concatenates the string.

# Polymorphism : Operator Overloading

- If the expression is for the form  $x + y$ , Python interprets it as  $x.\_\underline{add}\_\underline{(y)}$ .
  - The version of the  $\_\underline{add}\_\underline{()}$  method called depends upon the type of  $x$  and  $y$ .
  - If  $x$  and  $y$  are  $\text{int}$  objects then  $\text{int}$  class version of  $\_\underline{add}\_\underline{()}$  is called.
  - On the other hand, if  $x$  and  $y$  are  $\text{list}$  objects then the  $\text{list}$  class version of the  $\_\underline{add}\_\underline{()}$  method is called.
- 
- `>>> x, y = 10, 20`
  - `>>> x + y`
  - `30`
  - `>>> x.\_\underline{add}\_\underline{(y)} # same as x + y`
  - `30`
  - `>>> x, y = [11, 22], [1000, 2000]`
- 
- `>>> x.\_\underline{add}\_\underline{(y)} # same as x + y`
  - `[11, 22, 1000, 2000]`

# Polymorphism : Operator Overloading

| Operator | Special Method             | Description                |
|----------|----------------------------|----------------------------|
| +        | __add__(self, object)      | Addition                   |
| -        | __sub__(self, object)      | Subtraction                |
| *        | __mul__(self, object)      | Multiplication             |
| **       | __pow__(self, object)      | Exponentiation             |
| /        | __truediv__(self, object)  | Division                   |
| //       | __floordiv__(self, object) | Integer Division           |
| %        | __mod__(self, object)      | Modulus                    |
| ==       | __eq__(self, object)       | Equal to                   |
| !=       | __ne__(self, object)       | Not equal to               |
| >        | __gt__(self, object)       | Greater than               |
| >=       | __ge__(self, object)       | Greater than or equal to   |
| <        | __lt__(self, object)       | Less than                  |
| <=       | __le__(self, object)       | Less than or equal to      |
| len()    | __len__(self)              | Calculate number of items  |
| str()    | __str__(self)              | Convert object to a string |

# Example for Operator Overloading

- class Complex:
  - # defining init method for class
  - def \_\_init\_\_(self, r, i):
    - self.real = r
    - self.img = i
- # overloading the add operator using special function
  - def \_\_add\_\_(self, sec):
    - r = self.real + sec.real
    - i = self.img + sec.img
    - return complx(r,i)

## Example for Operator Overloading

- # string function to print object of Complex class
- def \_\_str\_\_(self):
- return str(self.real) + ' + ' + str(self.img) + 'i'
  
- c1 = Complex(5,3)
- c2 = Complex(2,4)
- print("sum = ",c1+c2)
  
-

# Exception Handling

- We can make certain mistakes while writing a program that lead to errors when we try to run it.
- A python program terminates as soon as it encounters an unhandled error. These errors can be broadly classified into two classes:
  - Syntax errors
  - Logical errors (Exceptions)

# Exception Handling : Syntax error

- Error caused by not following the proper structure (syntax) of the language is called syntax error or parsing error.
- Let's look at one example:
- >>> if a < 3
- File "<interactive input>", line 1
- if a < 3
- ^
- SyntaxError: invalid syntax
- As shown in the example, an arrow indicates where the parser ran into the syntax error.
- We can notice here that a colon : is missing in the if statement.

# Exception Handling : Logical error

- Errors that occur at runtime (after passing the syntax test) are called exceptions or logical errors.
- For instance, they occur when we try to open a file(for reading) that does not exist (`FileNotFoundException`), try to divide a number by zero (`ZeroDivisionError`), or try to import a module that does not exist (`ImportError`).
- Whenever these types of runtime errors occur, Python creates an exception object. If not handled properly, it prints a traceback to that error along with some details about why that error occurred.

# Exception Handling : Logical error

- >>> 1 / 0
- Traceback (most recent call last):
  - File "<string>", line 301, in runcode
  - File "<interactive input>", line 1, in <module>
  - ZeroDivisionError: division by zero
  
- >>> open("imaginary.txt")
- Traceback (most recent call last):
  - File "<string>", line 301, in runcode
  - File "<interactive input>", line 1, in <module>
  - FileNotFoundError: [Errno 2] No such file or directory: 'imaginary.txt'

# Exception Handling : Logical error

- Commonly occurring exceptions are usually defined in the compiler/interpreter. These are called built-in exceptions.
- Python's standard library is an extensive collection of built-in exceptions that deals with the commonly occurring errors (exceptions) by providing the standardized solutions for such errors.
- On the occurrence of any built-in exception, the appropriate exception handler code is executed which displays the reason along with the raised exception name.
- The programmer then must take appropriate action to handle it.

## Exception Handling : Logical error

- In Python, exceptions can be handled using a try statement.
- The critical operation which can raise an exception is placed inside the try clause.
- The code that handles the exceptions is written in the except clause.

# Exception Handling : Logical error

- # import module sys to get the type of exception
- import sys
- randomList = ['a', 0, 2]
- for entry in randomList:
- try:
- print('The entry is', entry)
- r = 1/int(entry)
- break
- except:
- print("Oops!", sys.exc\_info()[0], "occurred.")
- print("Next entry.")
- print()
- print("The reciprocal of", entry, "is", r)

## Exception Handling : Logical error

- In this program, we loop through the values of the randomList list. As previously mentioned, the portion that can cause an exception is placed inside the try block.
- If no exception occurs, the except block is skipped and normal flow continues(for last value). But if any exception occurs, it is caught by the except block (first and second values).
- Here, we print the name of the exception using the exc\_info() function inside sys module. We can see that a causes ValueError and 0 causes ZeroDivisionError.

# Exception Handling : Logical error

- # import module sys to get the type of exception
- import sys
- randomList = ['a', 0, 2]
- for entry in randomList:
- try:
- print("The entry is", entry)
- r = 1/int(entry)
- break
- except Exception as e:     #every exception inherits from base class Exception
- print("Oops!", e.\_\_class\_\_, "occurred.")
- print("Next entry.")
- print()
- print("The reciprocal of", entry, "is", r)

## Exception Handling :

- A try clause can have any number of except clauses to handle different exceptions, however, only one will be executed in case an exception occurs.
- We can use a tuple of values to specify multiple exceptions in an except clause. Here is an example pseudo code.

# Exception Handling :

- try:
- # do something
- pass
  
- except ValueError:
- # handle ValueError exception
- pass
  
- except (TypeError, ZeroDivisionError):
- # handle multiple exceptions
- # TypeError and ZeroDivisionError
- pass
  
- except:
- # handle all other exceptions
- pass

# Python except clause

- 1. Python Multiple Excepts
- It is possible to have multiple except blocks for one try block.
  - >>> a,b=1,0
  - >>> try:
    - print(a/b)
    - print("This won't be printed")
    - print('10'+10)
    - except TypeError:
      - print("You added values of incompatible types")
    - except ZeroDivisionError:
      - print("You divided by 0")
  - **Output**  
You divided by 0

# Python except clause

- When the interpreter encounters an exception, it checks the except blocks associated with that try block.
- These except blocks may declare what kind of exceptions they handle. When the interpreter finds a matching exception, it executes that except block.
- In our example, the first statement under the try block gave us a ZeroDivisionError.
- We handled this in its except block, but the statements in try after the first one didn't execute.
- This is because once an exception is encountered, the statements after that in the try block are skipped.
- And if an appropriate except block or a generic except block isn't found, the exception isn't handled.

# Python except clause

- In this case, the rest of the program won't run. But if you handle the exception, the code after the excepts and the finally block will run as expected.
- Let's try some code for this.
- a,b=1,0
- try:
- print(a/b)
- except:
- print("You can't divide by 0")
- print("Will this be printed?")
- **Output**
- == RESTART: C:\Users\lifei\AppData\Local\Programs\Python\Python36-32\try.py ==
- You can't divide by 0
- Will this be printed?

# Python except clause

- 2. Python Multiple Exception in one Except
- You can also have one except block handle multiple exceptions. To do this, use parentheses. Without that, the interpreter will return a syntax error.
- try:
- print('10'+10)
- print(1/0)
- except (TypeError,ZeroDivisionError):
- print("Invalid input")
- **Output**
- Invalid input

# Python except clause

- 3. A Generic except After All Excepts
- Finally, you can compliment all specific except blocks with a generic except at the end. This block will serve to handle all exceptions that go undetected by the specific except blocks. But there can only be one generic or default except block for one try block.
- `try:`
- `print('1'+1)`
- `print(sum)`
- `print(1/0)`
- `except NameError:`
- `print("sum does not exist")`
- `except ZeroDivisionError:`
- `print("Cannot divide by 0")`
- `except:`
- `print("Something went wrong")`
- **Output**
- `Something went wrong`

# Python try with else clause

- In Python programming, exceptions are raised when errors occur at runtime. We can also manually raise exceptions using the raise keyword.
- We can optionally pass values to the exception to clarify why that exception was raised.
- In python, you can also use else clause on the try-except block which must be present after all the except clauses.
- The code enters the else block only if the try clause does not raise an exception.

# Python try with else clause

- In some situations, you might want to run a certain block of code if the code block inside try ran without any errors. For these cases, you can use the optional else keyword with the try statement.
- Note: Exceptions in the else clause are not handled by the preceding except clauses.
- Let's look at an example:

# Python try with else clause

- # program to print the reciprocal of even numbers
- try:
- num = int(input("Enter a number: "))
- assert num % 2 == 0
- except:
- print("Not an even number!")
- else:
- reciprocal = 1/num
- print(reciprocal)
  
- Output
- If we pass an odd number:
  
- Enter a number: 1
- Not an even number!

# Python try with else clause

- If we pass an even number, the reciprocal is computed and displayed.
- Enter a number: 4
- 0.25
- However, if we pass 0, we get ZeroDivisionError as the code block inside else is not handled by preceding except.
- Enter a number: 0
- Traceback (most recent call last):
  - File "<string>", line 7, in <module>
  - reciprocal = 1/num
  - ZeroDivisionError: division by zero

# Python finally clause

- Python provides a keyword `finally`, which is always executed after `try` and `except` blocks. The `finally` block always executes after normal termination of `try` block or after `try` block terminates due to some exception.
- Syntax:
- `try:`
- `# Some Code....`
- `except:`
- `# optional block`
- `# Handling of exception (if required)`
- `else:`
- `# execute if no exception`
- `finally:`
- `# Some code .....(always executed)`

# Python finally clause

- The try statement in Python can have an optional finally clause. This clause is executed no matter what and is generally used to release external resources.
- For example, we may be connected to a remote data center through the network or working with a file or a Graphical User Interface (GUI).
- In all these circumstances, we must clean up the resource before the program comes to a halt whether it successfully ran or not. These actions (closing a file, GUI or disconnecting from network) are performed in the finally clause to guarantee the execution.

# Python finally clause

- try:
- f = open("test.txt",encoding = 'utf-8')
- # perform file operations
- finally:
- f.close()
  
- This type of construct makes sure that the file is closed even if an exception occurs during the program execution.

# Python Custom Exceptions

- For creating a custom exception, we derive a new class from the Exception class.
- ```
>>> class MyError(Exception):
```
- ```
    print("This is a problem")
```
- ```
>>> raise MyError("MyError happened")
```
- **Output :**
- Traceback (most recent call last):File “<pyshell#179>”, line 1, in <module>
- ```
raise MyError("MyError happened")
```
- MyError: MyError happened

# Python Custom Exceptions

- Raising Exception
- The raise statement allows the programmer to force a specific exception to occur.
- The sole argument in raise indicates the exception to be raised.
- This must be either an exception instance or an exception class (a class that derives from Exception).

# Python Custom Exceptions

- class Error(Exception):
- """Base class for other exceptions"""
- pass
  
- class ValSmallErr(Error):
- """Raised when the input value is too small"""
- pass
  
- class ValLargeErr(Error):
- """Raised when the input value is too large"""
- pass

# Python Custom Exceptions

- # you need to guess this number
- number = 10
- # user guesses a number until he/she gets it right
- while True:
  - try:
    - i\_num = int(input("Enter a number: "))
    - if i\_num < number:
      - raise ValSmallErr
    - elif i\_num > number:
      - raise ValLargeErr
    - break
  - except ValSmallErr:
    - print("This value is too small, try again!")
    - print()
  - except ValLargeErr:
    - print("This value is too large, try again!")
    - print()
  - print("Congratulations! You guessed it correctly.")

# References :

- <https://realpython.com/>
- <https://beginnersbook.com>
- <https://techvidvan.com/>
- <https://docs.python.org/>
- <https://www.javatpoint.com/>
- <https://www.w3schools.com/>
- <https://www.tutorialspoint.com/>
- <https://stackabuse.com/>
- <https://www.python-course.eu/>
- <https://developers.google.com/>
- <https://www.techbeamers.com/>
- <https://www.guru99.com/>
- <https://www.geeksforgeeks.org/>
- <https://www.programiz.com/>
- <https://intellipaat.com/>
- <https://data-flair.training/>

# Regular Expressions

Unit VII

# Introduction

- Consider a scenario where you have log file with large amount of data, and you only want to fetch the date and time from it.
- Consider another scenario where you are a salesperson, and you have number of email ids from which there are email ids which are fake.
- In both the scenarios you can use regular expressions to find the genuine email id and the date and time with a certain pattern.

# Introduction

- Regular Expressions (sometimes shortened to regexp, regex, or re) are a tool for matching patterns in text.
- In Python, we have the re module.
- Regular Expression (re) is a sequence with special characters.
- It can help to examine every character in a string to see if it matches a certain pattern: what characters appearing at what positions by how many times.

# Introduction

- The most common uses of regular expressions are:
- Search a string (search and match)
- Finding a string (findall)
- Break string into a sub strings (split)
- Replace part of a string (sub)
- Let's look at the methods that library “re” provides to perform these tasks.

# Introduction

- We can handle many basic patterns in Python using ordinary characters. Ordinary characters are the simplest regular expressions.
- They match themselves exactly and do not have a special meaning in their regular expression syntax.
- Examples are 'A', 'a', 'X', '5'.
- Ordinary characters can be used to perform simple exact matches:
- pattern = r"Example"
- sequence = "Example"
- if re.match(pattern, sequence):
- print("Match!")
- else: print("Not a match!")

# Introduction

- The `match()` function returns a match object if the text matches the pattern.
- Otherwise, it returns `None`.
- The `re` module also contains several other functions.
  - `re.match()`
  - `re.search()`
  - `re.findall()`
  - `re.split()`
  - `re.sub()`
  - `re.compile()`

# Introduction

- **re.match(pattern, string):**
- This method finds match if it occurs at start of the string.
- For example, calling match() on the string ‘SA Analytics Pune’ and looking for a pattern ‘SA’ will match.
- However, if we look for only Analytics, the pattern will not match.

# Introduction

- **re.search(pattern, string):**
- It is similar to match() but it doesn't restrict us to find matches at the beginning of the string only. Unlike previous method, here searching for pattern 'Analytics' will return a match.
- `result = re.search(r'Analytics', 'SA Analytics Pune SA')`
- `print result.group(0)`
- **Output:**
- `Analytics`
- `search()` method is able to find a pattern from any position of the string but it only returns the first occurrence of the search pattern.

# Introduction

- **re.findall (pattern, string):**
- It helps to get a list of all matching patterns.
- It has no constraints of searching from start or end.
- If we will use method findall to search 'SA' in given string it will return occurrence of SA.
  
- result = re.findall(r'SA', 'SA Analytics Pune SA ')
- print result
- Output:
- ['SA', 'SA']

# Introduction

- **re.split(pattern, string, [maxsplit=0]):**
- This methods helps to split string by the occurrences of given pattern.
- result=re.split(r'y','Analytics')
- result
- **Output:**
- ['Anal', 'tics']

## Introduction

- Method split() has another argument “maxsplit”. It has default value of zero.
- In this case it does the maximum splits that can be done, but if we give value to maxsplit, it will split the string.
- Let's look at the example below:
- ```
result=re.split(r'i','Analytics Pimpri Pune')
```
- print result
- Output:
- ```
['Analyt', 'cs P', 'mpr', 'Pune'] #It has performed all the splits that can be done by pattern "i".
```

# Introduction

- Example
- `result=re.split(r'i','Analytics Pimpri',maxsplit=1)`
- `result`
- Output:
- `['Analyt', 'cs Pimpri']`

# Introduction

- **re.sub(pattern, repl, string):**
- It helps to search a pattern and replace with a new sub string. If the pattern is not found, string is returned unchanged.
- result=re.sub(r'India','the World','SA is largest Analytics community of India')
- result
- **Output:**
- ‘SA is largest Analytics community of the World’

# Introduction

- **re.compile(pattern, repl, string):**
- We can combine a regular expression pattern into pattern objects, which can be used for pattern matching. It also helps to search a pattern again without rewriting it.
- import re
- pattern=re.compile('SA')
- result=pattern.findall('SA Analytics Pune SA')
- print result
- result2=pattern.findall('SA is largest analytics community of India')
- print result2
- Output:
- ['SA', 'SA']
- ['SA']

# Introduction

| Operators | Description                                                                                                                                  |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------|
| .         | Matches with any single character except newline '\n'.                                                                                       |
| ?         | match 0 or 1 occurrence of the pattern to its left                                                                                           |
| +         | 1 or more occurrences of the pattern to its left                                                                                             |
| *         | 0 or more occurrences of the pattern to its left                                                                                             |
| \w        | Matches with a alphanumeric character whereas \W (upper case W) matches non alphanumeric character.                                          |
| \d        | Matches with digits [0-9] and /D (upper case D) matches with non-digits.                                                                     |
| \s        | Matches with a single white space character (space, newline, return, tab, form) and \S (upper case S) matches any non-white space character. |
| \b        | boundary between word and non-word and /B is opposite of /b                                                                                  |
| [..]      | Matches any single character in a square bracket and [^..] matches any single character not in square bracket                                |

# Introduction

| Operators  | Description                                                                                                                                                                    |
|------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| \          | It is used for special meaning characters like \. to match a period or \+ for plus sign.                                                                                       |
| ^ and \$   | ^ and \$ match the start or end of the string respectively                                                                                                                     |
| {n,m}      | Matches at least n and at most m occurrences of preceding expression if we write it as {,m} then it will return at least any minimum occurrence to max m preceding expression. |
| a   b      | Matches either a or b                                                                                                                                                          |
| ( )        | Groups regular expressions and returns matched text                                                                                                                            |
| \t, \n, \r | Matches tab, newline, return                                                                                                                                                   |

# Regular Expressions

## r — python raw string

Here, r means python raw string. Specifying 'r' means we don't want Python to treat '\' as escape characters but just a normal character.

## \d — all the digits but only one digit

If the text contains a digit, it will be considering matching the '\d' pattern. the '+' after '\d' means ONE or MORE digit. So it will not scan the numbers one digit by one digit but take the whole package.

## {3} — 3 digits, {7} — digits

The above code means we are looking for a string that consists of 3 digits, followed by a '-' and then 7 digits.

## \w — similar as d for digits, w is for any characters: digit, character, \_

## Practice Assignments

- Problem 1: Return the first word of a given string.
- Problem 2: Return the first two character of each word.
- Problem 3: Return all words of a string those starts with vowel
- Problem 4: Split a string with multiple delimiters.
- Problem 5: Replace the multiple delimiters in a string with ; say space or any other character.
- Problem 6: Extract all characters after “@”
- Problem 7: Extract date from a given string(dd-mm-yyyy)

## Generating an iterator:

- Generating an iterator is the simple process of finding out and reporting the starting and the ending index of the string. Consider the following example:
- ```
import re

Str = "we need to inform him with the latest information"

for i in re.finditer("inform.", Str):
    locTuple = i.span()
    print(locTuple)
```
- For every match found, the starting and the ending index is printed.

## Matching words with patterns:

- Consider an input string where you have to match certain words with the string. To elaborate, check out the following example code:
- import re
- 
- Str = "Sat, hat, mat, pat"
- 
- allStr = re.findall("[shmp]at", Str)
- 
- for i in allStr:
- print(i)

## Matching words with patterns:

- What is common in the string?
- The letters ‘a’ and ‘t’ are common among all of the input strings.
- [shmp] in the code denotes the starting letter of the words to be found.
- So any substring starting with the letters s, h, m or p will be considered for matching.
- Any among that and compulsorily followed by ‘at’ at the end

## Matching series of range of characters:

- We wish to output all the words whose first letter should start in between h and m and compulsorily followed by at.

```
■ import re  
■  
■ Str = "sat, hat, mat, pat"  
■  
■ someStr = re.findall("[h-m]at", Str)  
■  
■ for i in someStr:  
■     print(i)
```

## Matching series of range of characters:

- Let us now change the above program very slightly to obtain a very different result. Check out the below code and try to catch the difference between the above one and the below one:
- import re
- Str = "sat, hat, mat, pat"
- 
- someStr = re.findall("[^h-m]at", Str)
- 
- for i in someStr:
- print(i)
- The caret symbol negates the effect of whatever it follows. Instead of giving us the output of everything starting with h to m, we will be presented with the output of everything apart from that.

## Replacing a string:

- import re
- 
- Food = "hat rat bat cat"
- 
- regex = re.compile("[r]at")
- 
- Food = regex.sub("food", Food)
- 
- print(Food)
- In the above example, the word rat is replaced with the word food.

## Matching a single character:

- A single character from a string can be individually matched using Regular Expressions easily.  
Check out the following code snippet:
- import re
- rstr = "45678"
- 
- print("Matches: ", len(re.findall("d{6}", rstr)))
- The expected output is the 5th number that occurs in the given input string.

## Removing Newline Spaces:

- We can remove the newline spaces using Regular Expressions easily in Python. Consider another snippet of code as shown here:

```
■ import re  
■ rstr = ""  
■ You Should Never  
■ Give Up Easily  
■ Keep trying.  
■ ""  
■ print(rstr)  
■ regex = re.compile("\n")  
■ randstr = regex.sub(" ", rstr)  
■ print(rstr)
```

# Practical Use Cases Of Regular Expressions

- **Phone Number Verification:**
- Problem Statement – The need to easily verify phone numbers in any relevant scenario.
- Consider the following Phone numbers:
  - 444-122-1234
  - 123-122-78999
  - 111-123-23
  - 67-7890-2019

# Practical Use Cases Of Regular Expressions

- The general format of a phone number is as follows:
- Starts with 3 digits and ‘-’ sign
- 3 middle digits and ‘-’ sign
- 4 digits in the end
- We will be using w in the example below. Note that **w = [a-zA-Z0-9\_]**
- import re
- phn = "412-555-1212"
- if re.search("w{3}-w{3}-w{4}", phn):
- print("Valid phone number")

# Practical Use Cases Of Regular Expressions

- **E-mail Verification:**
- Problem statement – To verify the validity of an E-mail address in any scenario.
- Consider the following examples of email addresses:
  - Anish@gmail.com
  - Anupama @ com
  - AC .com
  - 123 @.com

# Practical Use Cases Of Regular Expressions

- **Guidelines:**
- **All E-mail addresses should include:**
- 1 to 20 lowercase and/or uppercase letters, numbers, plus . \_ % +
- An @ symbol
- 2 to 20 lowercase and uppercase letters, numbers and plus
- A period symbol
- 2 to 3 lowercase and uppercase letters

# Practical Use Cases Of Regular Expressions

- **Code:**
- import re
- 
- email = "abc@abl.com md@.com @ceo.com pc@.com"
- print("Email Matches: ", len(re.findall("[w.\_%+-]{1,20}@[\w.-]{2,20}.[A-Za-z]{2,3}", email)))
  
- **Output:**
- Email Matches: 1

# Practical Use Cases Of Regular Expressions

- . - A period. Matches any single character except the newline character.
- \$ - Matches the end of string.
  - `re.search(r'cake$', "Cake! Let's eat cake").group()`
  - `'cake'`
  - ## The next search will return the NONE value, try it:
  - `# re.search(r'cake$', "Let's get some cake on our way home!").group()`
- [abc] - Matches a or b or c.
- [a-zA-Z0-9] - Matches any letter from (a to z) or (A to Z) or (0 to 9).
- {Characters that are not within a range can be matched by complementing the set. If the first character of the set is ^, all the characters that are not in the set will be matched.}

# Practical Use Cases Of Regular Expressions

- `re.search(r'[0-6]', 'Number: 5').group()`
- `'5'`
- `## Matches any character except 5`
- `re.search(r'Number: [^5]', 'Number: 0').group()`
- `## This will not match and hence a NONE value will be returned`
- `#re.search(r'Number: [^5]', 'Number: 5').group()`
- `'Number: 0'`

# Practical Use Cases Of Regular Expressions

- \t - Lowercase t. Matches tab.
- \n - Lowercase n. Matches newline.
- \r - Lowercase r. Matches return.
- \A - Uppercase a. Matches only at the start of the string. Works across multiple lines as well.
- \Z - Uppercase z. Matches only at the end of the string.
- \b - Lowercase b. Matches only the beginning or end of the word.

# Repetitions

- + - Checks if the preceding character appears one or more times starting from that position.
  - `re.search(r'Co+kie', 'Coooookie').group()`
  - 'Coooookie'
- \* - Checks if the preceding character appears zero or more times starting from that position.
  - # Checks for any occurrence of a or o or both in the given sequence
  - `re.search(r'C*a*o*kie', 'Cookie').group()`
  - 'Cookie'
- ? - Checks if the preceding character appears exactly zero or one time starting from that position.
  - # Checks for exactly zero or one occurrence of a or o or both in the given sequence
  - `re.search(r'Colou?r', 'Color').group()`
  - 'Color'

# Repetitions

- To check for an exact number of sequence repetition:
- For example, checking the validity of a phone number in an application. re module handles this very gracefully as well using the following regular expressions:
  - {x} - Repeat exactly x number of times.
  - {x,} - Repeat at least x times or more.
  - {x, y} - Repeat at least x times but no more than y times.
    - `re.search(r'\d{9,10}', '0987654321').group()`
    - `'0987654321'`

## **^ - usage**

- The only other special character inside square brackets (character class choice) is the caret "^".
- If it is used directly after an opening square bracket, it negates the choice.
- `[^0-9]` denotes the choice "any character but a digit".
- The position of the caret within the square brackets is crucial.
- If it is not positioned as the first character following the opening square bracket, it has no special meaning.
- `[^abc]` means anything but an "a", "b" or "c" `[a^bc]` means an "a", "b", "c" or a "^"

# Grouping in Regular Expressions

- The group feature of regular expression allows you to pick up parts of the matching text.
- Parts of a regular expression pattern bounded by parenthesis () are called groups.
- The parenthesis does not change what the expression matches, but rather forms groups within the matched sequence.
- If we are validating email addresses and want to check the user name and host.
- Then we will create separate groups within the matched text.
- We can group a part of a regular expression by surrounding it with parenthesis (round brackets).
- This way we can apply operators to the complete group instead of a single character.

# Grouping in Regular Expressions

- statement = 'Please contact us at: placements@fergusson.edu'
- match = re.search(r'([\w\.-]+)@([\w\.-]+)', statement)
- if statement:
  - print("Email address:", match.group()) # The whole matched text
  - print("Username:", match.group(1)) # The username (group 1)
  - print("Host:", match.group(2)) # The host (group 2)
- Email address: placements@fergusson.edu
- Username: placements
- Host: fergusson.edu

# Grouping in Regular Expressions

- `re.search()` returns a match object if it matches and `None` otherwise.
- The match object contains a lot of data about what has been matched, positions and so on.
- A match object contains the methods `group()`, `span()`, `start()` and `end()`,

# Grouping in Regular Expressions

- `span()` returns a tuple with the start and end position, i.e. the string index where the regular expression started matching in the string and ended matching.
- The methods `start()` and `end()` are in a way superfluous as the information is contained in `span()`, i.e. `span()[0]` is equal to `start()` and `span()[1]` is equal to `end()`.
- `group()`, if called without argument, it returns the substring, which had been matched by the complete regular expression.
- With the help of `group()` we are also capable of accessing the matched substring by grouping parentheses, to get the matched substring of the n-th group, we call `group()` with the argument n: `group(n)`.
- We can also call `group` with more than integer argument, e.g. `group(n,m)`. `group(n,m)` - provided there exists a subgroup n and m - returns a tuple with the matched substrings. `group(n,m)` is equal to `(group(n), group(m))`:

# Grouping in Regular Expressions

- import re
- mo = re.search("([0-9]+).\*: (.\*)", "Customer number: 254678, Date: March 31, 2021")
- mo.group()
- Output:  
■ '254678, Date: March 31, 2021 '
- mo.group(1)
- Output:  
■ '254678'
- mo.group(2)
- Output:  
■ 'March 31, 2021'
- mo.group(1,2)
- Output:  
■ ('254678', 'March 31, 2021')

# Grouping in Regular Expressions

- For example:
- ```
>>> s = 'Today is 31-03-2021'
```
- ```
>>> mo = re.search(r'\d{2}-\d{2}-\d{4}', s)
```
- ```
>>> print(mo.group())
```
- `31-03-2021`
- We can *capture* various parts of this regular expression by putting them in parentheses:
- `(\d{2})-(\d{2})-(\d{4})`
- If Python matches this regular expression, we can then retrieve each *captured group* separately.

# Grouping in Regular Expressions

- >>> mo = re.search(r'(\d{2})-(\d{2})-(\d{4})', s)
- >>> # Note: The entire matched string is still available
- >>> print(mo.group())
- 31-03-2021
- >>> # The first captured group is the date
- >>> print(mo.group(1))
- 31
- >>> # And this is its start/end position in the string
- >>> print('%s %s' % (mo.start(1), mo.end(1)))
- 9 11

# Grouping in Regular Expressions

- >>> # The second captured group is the month
- >>> print(mo.group(2))
- 03
- >>> # The third captured group is the year
- >>> print(mo.group(3))
- 2021
- When you start writing more complex regular expressions, with lots of captured groups, it can be useful to refer to them by a meaningful name rather than a number. The syntax is (...), where ... is the regular expression to be captured, and name is the name you want to give to the group.

# Grouping in Regular Expressions

- >>> s = "Ram's ID: abc123"
- >>> # A normal captured group
- >>> mo = re.search(r'ID: (.+)', s)
- >>> print(mo.group(1))
- abc123
- >>> # A named captured group
- >>> mo = re.search(r'ID: (?P<id>.+)', s)
- >>> print(mo.group('id'))
- abc123

# Grouping in Regular Expressions

- **Re-using Captured Groups with Regular Expressions**
- We can also take captured groups and re-use them later in the regular expression! (?P=name) means *match whatever was previously matched in the named group*. For example:
- >>> s = 'abc 123 def 456 def 789'
- >>> mo = re.search(r'(?P<foo>def) \d+', s)
- >>> print(mo.group())
- def 456
- >>> print(mo.group('foo'))
- def

# Grouping in Regular Expressions

- >>> # Capture 'def' in a group
- >>> mo = re.search(r'(?P<foo>def) \d+ (?P=foo)', s)
- >>> print(mo.group())
- def 456 def
- >>> mo.group('foo')
- def

# Grouping in Regular Expressions

- Another way of doing the same is with the use of <> brackets.
- This will create named groups.
- Named groups will make the code more readable.
- The syntax for creating named group is: (?P<name>...).
- Replace the name part with the name to be given to the group.
- The ... represent the rest of the matching syntax

# Grouping in Regular Expressions

- statement = 'Please contact us at: placements@fergusson.edu '
- match = re.search(r'(?P<email>(?P<username>[\w\.-]+)@(?P<host>[\w\.-]+))', statement)
- if statement:
  - print("Email address:", match.group('email'))
  - print("Username:", match.group('username'))
  - print("Host:", match.group('host'))
- Email address: placements@fergusson.edu
- Username: placements
- Host: fergusson.edu

# Grouping in Regular Expressions

- let's assume we have a file (called "tags.txt") with content like this:
  - <composer> Salil Kulkarni </composer>
  - <author> PLDeshpande </author>
  - <city> Pune </city>
- We want to rewrite this text automatically to
  - composer: Salil Kulkarni
  - author: PLDeshpande
  - city: Pune

# Grouping in Regular Expressions

- This regular expression works like this:
- It tries to match a less than symbol "<".
- After this it is reading lower case letters until it reaches the greater than symbol.
- Everything encountered within "<" and ">" has been stored in a back reference which can be accessed within the expression by writing \1.
- Let's assume \1 contains the value "composer".
- When the expression has reached the first ">", it continues matching, as the original expression had been "(.\*)":

# Grouping in Regular Expressions

- import re
- fh = open("tags.txt")
- for i in fh:
  - res = re.search(r"<([a-z]+)>(.\*)</\1>",i)
  - print(res.group(1) + ":" + res.group(2))
- If there are more than one pair of parenthesis (round brackets) inside the expression, the backreferences are numbered \1, \2, \3, in the order of the pairs of parenthesis.

# Grouping in Regular Expressions

| Character(s) | What it does                                                                                                                                                                                                                                                                                                                                               |
|--------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| .            | A period. Matches any single character except the newline character.                                                                                                                                                                                                                                                                                       |
| ^            | A caret. Matches a pattern at the start of the string.                                                                                                                                                                                                                                                                                                     |
| \A           | Uppercase A. Matches only at the start of the string.                                                                                                                                                                                                                                                                                                      |
| \$           | Dollar sign. Matches the end of the string.                                                                                                                                                                                                                                                                                                                |
| \Z           | Uppercase Z. Matches only at the end of the string.                                                                                                                                                                                                                                                                                                        |
| []           | Matches the set of characters you specify within it.                                                                                                                                                                                                                                                                                                       |
| \            | <ul style="list-style-type: none"><li>If the character following the backslash is a recognized escape character, then the special meaning of the term is taken.</li><li>Else the backslash () is treated like any other character and passed through.</li><li>It can be used in front of all the metacharacters to remove their special meaning.</li></ul> |

# Grouping in Regular Expressions

| Character(s) | What it does                                                                          |
|--------------|---------------------------------------------------------------------------------------|
| \w           | Lowercase w. Matches any single letter, digit, or underscore.                         |
| \W           | Uppercase W. Matches any character not part of \w (lowercase w).                      |
| \s           | Lowercase s. Matches a single whitespace character like: space, newline, tab, return. |
| \S           | Uppercase S. Matches any character not part of \s (lowercase s).                      |
| \d           | Lowercase d. Matches decimal digit 0-9.                                               |

# Compiling Regular Expressions

- If you want to use the same regexp more than once in a script, it might be a good idea to use a regular expression object, i.e. the regex is compiled.
- The general syntax:
- `re.compile(pattern[, flags])`
- `compile` returns a regex object, which can be used later for searching and replacing. The expressions behaviour can be modified by specifying a flag value.

# Compiling Regular Expressions

| Abbreviation | Full name     | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|--------------|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| re.I         | re.IGNORECASE | Makes the regular expression case-insensitive                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| re.L         | re.LOCALE     | The behaviour of some special sequences like \w, \W, \b, \s, \S will be made dependent on the current locale, i.e. the user's language, country etc.                                                                                                                                                                                                                                                                                                                                                                                                |
| re.M         | re.MULTILINE  | ^ and \$ will match at the beginning and at the end of each line and not just at the beginning and the end of the string                                                                                                                                                                                                                                                                                                                                                                                                                            |
| re.S         | re.DOTALL     | The dot "." will match every character <b>plus the newline</b>                                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| re.U         | re.UNICODE    | Makes \w, \W, \b, \B, \d, \D, \s, \S dependent on Unicode character properties                                                                                                                                                                                                                                                                                                                                                                                                                                                                      |
| re.X         | re.VERBOSE    | Allowing "verbose regular expressions", i.e. whitespace are ignored. This means that spaces, tabs, and carriage returns are not matched as such. If you want to match a space in a verbose regular expression, you'll need to escape it by escaping it with a backslash in front of it or include it in a character class.<br># are also ignored, except when in a character class or preceded by an non-escaped backslash. Everything following a "#" will be ignored until the end of the line, so this character can be used to start a comment. |

## Example

- Here is an example. We write a regular expression, which can recognize the postal codes or postcodes of the UK.
- Postcode units consist of between five and seven characters, which are separated into two parts by a space. The two to four characters before the space represent the so-called outward code or out code intended to directly mail from the sorting office to the delivery office. The part following the space, which consists of a digit followed by two uppercase characters, comprises the so-called inward code, which is needed to sort mail at the final delivery office. The last two uppercase characters do not use the letters CIKMOV, so as not to resemble digits or each other when hand-written.
- The outward code can have the following form: One or two uppercase characters, followed by either a digit or the letter R, optionally followed by an uppercase character or a digit. (We do not consider all the detailed rules for postcodes, i.e only certain character sets are valid depending on the position and the context.) A regular expression for matching this superset of UK postcodes looks like this:
- `r"\b[A-Z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}\b"`
- The following Python program uses the regexp above:

# Example

```
import re

example_codes = ["SW1A 0AA", # House of Commons
                 "EC1V 8DS", # Clerkenwell, London

                 "SW1A 1AA", # Buckingham Palace
                 "WC1X 9DT", # WC1X 9DT

                 "SW1A 2AA", # Downing Street
                 "B42 1LG", # Birmingham

                 "BX3 2BB", # Barclays Bank
                 "B28 9AD", # Birmingham

                 "DH98 1BT", # British Telecom
                 "W12 7RJ", # London, BBC News Centre

                 "N1 9GU", # Guardian Newspaper
                 "BBC 007" # a fake postcode

                 "E98 1TT", # The Times
                 pc_re = r"[A-z]{1,2}[0-9R][0-9A-Z]? [0-9][ABD-HJLNP-UW-Z]{2}"

                 "TIM E22", # a fake postcode
                 for postcode in example_codes:

                 "A B1 A22", # not a valid postcode
                 r = re.search(pc_re, postcode)

                 "EC2N 2DB", # Deutsche Bank
                 if r:

                 "SE9 2UG", # University of Greenwich
                 print(postcode + " matched!")

                 "N1 OUY", # Islington, London
                 else:
                     print(postcode + " is not a valid postcode!")
```

## Practice Assignments

- Write a regular expression which matches strings which starts with a sequence of digits - at least one digit - followed by a blank.
- Write a regular expression to find and replace certain values in the string.
- Write a regular expression to read contents of a file based on a pattern.
- Write a program to calculate how many matches were found for a pattern.
- Write a program to extract the phone number from a given string.

# References :

- <https://realpython.com/>
- <https://beginnersbook.com>
- <https://towardsdatascience.com/>
- <https://docs.python.org/>
- <https://www.w3schools.com/>
- <https://www.tutorialspoint.com/>
- <https://stackabuse.com/>
- <https://www.python-course.eu/>
- <https://developers.google.com/>
- <https://www.techbeamers.com/>
- <https://www.guru99.com/>
- <https://www.geeksforgeeks.org/>
- <https://www.programiz.com/>
- <https://intellipaat.com/>
- <https://data-flair.training/>
- <https://www.learnpython.org/>
- <https://www.analyticsvidhya.com/>
- <https://www.edureka.co/>
- <https://www.pythoncentral.io>

# DATABASES WITH PYTHON

Using SQLite

# Introduction

- All software applications interact with data, most commonly through a database management system (DBMS).
- Some programming languages come with modules that you can use to interact with a DBMS, while others require the use of third-party packages.
- Connecting a program with a database is a tough task.
- It's used to connect the front-end of your application with the back-end database.
- Python have its native builtin modules with which the things become easy.

# Introduction

- Databases have various functionalities by which one can manage large amount of information easily over the web, and huge/voluminous, data input and output over a typical file such as a text file.
- SQL is a query language and is very popular in databases.
- SQLite is a “light” version that works over syntax very much similar to SQL.

# Introduction

- SQLite is a self-contained, high-reliability, embedded, full-featured, public-domain, SQL database engine.
- It is the most used database engine in the world wide web.
- Python has a library to access SQLite databases, called sqlite3, intended for working with this database which has been included with Python package since version 2.5.

# Introduction

- SQLite features include:
- **No need for servers:** That is, there are no server processes that need to start, stop or be configured. There is no need for database administrators to assign instances or manage user access permissions.
- **Simple Database Files:** A SQLite database is a single ordinary disk file that can be stored in any directory. This also makes it simple to share since database files can easily be copied onto a memory stick or sent through email.
- **Manifest Typing:** The majority of SQL database engines rely on static typing. That is, you can only store values of the same datatype that is associated with a particular column. However, SQLite uses manifest typing, which allows the storage of any amount of any data type into any column without no matter the column's declared datatype. Note that there are some exceptions to this rule. One such exception is integer primary key columns, which can only store integers.

# Introduction

- Here are some scenarios that you could use SQLite.
- Embedded devices and IoT
- Data Analysis
- Data Transferring
- File archive and/or data container
- Internal or temporary databases
- Stand-in for an enterprise database during demos or testing
- Education, training and testing

# Introduction

- Most importantly, SQLite is actually built-in as a Python library
- You don't need to install any server-side/client-side software, and you don't need to keep something running as a service, as long as you imported the library in Python and start coding, then you have a relational database management system!
- Simply import it by:
- `import sqlite3 as sl`

# Introduction

- Important point to be noted is that SQLite is case insensitive, but there are some commands, which are case sensitive like **GLOB** and **glob** have different meaning in SQLite statements.

# Introduction

- Structured Query Language(SQL) as we all know is the database language by the use of which we can perform certain operations on the existing database and also we can use this language to create a database.
- SQL uses certain commands like Create, Drop, Insert etc. to carry out the required tasks.
- These SQL commands are mainly categorized into four categories as:
  - DDL – Data Definition Language
  - DQL – Data Query Language
  - DML – Data Manipulation Language
  - DCL – Data Control Language

# Introduction

- DDL(Data Definition Language) : DDL or Data Definition Language actually consists of the SQL commands that can be used to define the database schema. It simply deals with descriptions of the database schema and is used to create and modify the structure of database objects in the database.
- Examples of DDL commands:
- CREATE – is used to create the database or its objects (like table, index, function, views, store procedure and triggers).
- DROP – is used to delete objects from the database.
- ALTER-is used to alter the structure of the database.
- TRUNCATE—is used to remove all records from a table, including all spaces allocated for the records are removed.
- COMMENT –is used to add comments to the data dictionary.
- RENAME –is used to rename an object existing in the database.

# Introduction

- DQL (Data Query Language) :
- DML statements are used for performing queries on the data within schema objects. The purpose of DQL Command is to get some schema relation based on the query passed to it.
- Example of DQL:
- SELECT – is used to retrieve data from the a database.

# Introduction

- DML(Data Manipulation Language) : The SQL commands that deals with the manipulation of data present in the database belong to DML or Data Manipulation Language and this includes most of the SQL statements.
- Examples of DML:
  - INSERT – is used to insert data into a table.
  - UPDATE – is used to update existing data within a table.
  - DELETE – is used to delete records from a database table.

# Introduction

- DCL(Data Control Language) : DCL includes commands such as GRANT and REVOKE which mainly deals with the rights, permissions and other controls of the database system.
- Examples of DCL commands:
  - GRANT-gives user's access privileges to database.
  - REVOKE-withdraw user's access privileges given by using the GRANT command.

# Introduction

- TCL(transaction Control Language) : TCL commands deals with the transaction within the database.
- Examples of TCL commands:
  - COMMIT– commits a Transaction.
  - ROLLBACK– rollbacks a transaction in case of any error occurs.
  - SAVEPOINT–sets a savepoint within a transaction.
  - SET TRANSACTION–specify characteristics for the transaction.

# Introduction

- Python has a native library for SQLite. To use SQLite, we must import sqlite3.
- Then create a connection using connect() method and pass the name of the database you want to access.
- If there is a file with that name, it will open that file otherwise, Python will create a file with the given name.
- After this, a cursor object is called which is capable of sending commands to the SQL.
- **Cursor** is a control structure used to traverse and fetch the records of the database.
- Cursor has a major role in working with Python. All the commands will be executed using cursor object only.

# Introduction

- The SQLite3 cursor is a method of the connection object.
- To create a table in the database, create an object and write the SQL command in it.
- Example:- **sql\_comm = "SQL statement"**
- And executing the command is very easy.
- Call the cursor method execute and pass the name of the sql command as a parameter in it.
- Save a number of commands as the sql\_comm and execute them.
- After you perform all your activities, save the changes in the file by committing those changes and then lose the connection.

## Create a Connection to DB

- One can create an SQLite database and have a connection object as simple as:
- **con = sl.connect('Testdb.db')**
- After this line of code is run, the database is created and connected to it.
- If the database is not existing, it will create a database with that name.
- Otherwise, we can use exactly the same code to connect to an existing database.

# Create a Connection to DB

- When you create a connection with SQLite, that will create a database file automatically if it doesn't already exist.
- This database file is created on disk; we can also create a database in RAM by using :memory: with the connect function.
- This database is called in-memory database.
- Consider the code below in which we have created a database with a try, except and finally blocks to handle any exceptions:

# Create a Connection to DB

```
◦ import sqlite3  
◦ from sqlite3 import Error  
◦ def sql_connection():  
◦     try:  
◦         con = sqlite3.connect(':memory:')  
◦         print("Connection is established: Database is created in memory")  
◦     except Error:  
◦         print(Error)  
◦     finally:  
◦         con.close()  
◦ sql_connection()
```

## Create a Table

- To create a table in SQLite3, you can use the Create Table query in the execute() method.
- Consider the following steps:
  - Create a connection object.
  - From the connection object, create a cursor object.
  - Using the cursor object, call the execute method with create table query as the parameter.

# Create a Table

- let's create a table.
- with con:
- `con.execute("""`
- **CREATE TABLE Employee**
- **(ID INT PRIMARY KEY NOT NULL,**
- **NAME TEXT NOT NULL,**
- **AGE INT NOT NULL,**
- **ADDRESS CHAR(50),**
- **SALARY REAL) """)**

## Insert Records

- We can also pass values/arguments to an INSERT statement in the execute() method. We can use the question mark (?) as a placeholder for each value.
- Let's insert some records into the USER table we just created.
- Suppose we want to insert multiple entries in one go.
- **sql = 'INSERT INTO Employee (ID,NAME,AGE,ADDRESS,SALARY) values(?, ?, ?, ?, ?)'**
- **data = [(5, 'Arati', 32, 'Delhi', 20000.00 ), (6, 'Mahesh', 25, 'Mumbai', 15000.00 )]**
- with con:
- **con.executemany(sql, data)**
- This is the bulk insert into the table.

## Update Table

- To update the table, simply create a connection, then create a cursor object using the connection and finally use the UPDATE statement in the execute() method.
- `con.execute("UPDATE Employee set NAME = 'Shweta' where ID = 5")`

# Fetching data

- We can use the select statement to select data from a particular table.
  - If we want to select all the columns of the data from a table, we can use the asterisk (\*).
  - The syntax for this will be as follows:
- 
- **select \* from table\_name**
  - In SQLite3, the SELECT statement is executed in the execute method of the cursor object.

## Query the Table

- Let's query the table to get the sample rows back.
- with con:
- data = con.execute("SELECT \* FROM Employee WHERE ID <= 6")
- for row in data:
- print(row)

## Query the Table

- To execute queries in SQLite, use cursor.execute().
- .execute() can execute any query passed to it in the form of string.
- If you want to select a few columns from a table, then specify the columns like the following:
- **select column1, column2 from table\_name**

## Query the Table

- The select statement selects the required data from the database table, and if you want to fetch the selected data, the **fetchall()** method of the cursor object is used.
- To fetch the data from a database, we will execute the SELECT statement and then will use the **fetchall()** method of the cursor object to store the values into a variable.
- After that, we will loop through the variable and print all values.

# Query the Table

- import sqlite3
- con = sqlite3.connect('mydatabase.db')
- def sql\_fetch(con):
- cursorObj = con.cursor()
- cursorObj.execute('SELECT \* FROM Employee')
- rows = cursorObj.fetchall()
- for row in rows:
- print(row)
- sql\_fetch(con)

## Query the Table

- You can also use the fetchall() in one line as follows:
- **[print(row) for row in cursorObj.fetchall()]**

## SQLite3 rowcount

- The SQLite3 rowcount is used to return the number of rows that are affected or selected by the latest executed SQL query.
- When we use rowcount with the SELECT statement, -1 will be returned as how many rows are selected is unknown until they are all fetched.
- Consider the example below:
- `print(cursorObj.execute('SELECT * FROM employees').rowcount)`

## SQLite3 rowcount

- Therefore, to get the row count, you need to fetch all the data, and then get the length of the result:
- **rows = cursorObj.fetchall()**
- **print len (rows)**
  
- When you use the DELETE statement without any condition (a where clause), that will delete all the rows in the table, and it will return the total number of deleted rows in rowcount.
- **print(cursorObj.execute('DELETE FROM employees').rowcount)**
- If no row is deleted, it will return zero.

## List tables

- To list all tables in an SQLite3 database, you should query the sqlite\_master table and then use the fetchall() to fetch the results from the SELECT statement.
- The sqlite\_master is the master table in SQLite3, which stores all tables.

```
import sqlite3
con = sqlite3.connect('mydatabase.db')
def sql_fetch(con):
    cursorObj = con.cursor()
    cursorObj.execute('SELECT name from sqlite_master where type= "table"')
    print(cursorObj.fetchall())
sql_fetch(con)
```

## Check if a table exists or not

- When creating a table, we should make sure that the table is not already existing.
- Similarly, when removing/ deleting a table, the table should exist.
- To check if the table doesn't already exist, we use “if not exists” with the CREATE TABLE statement as follows:
- **create table if not exists table\_name (column1, column2, ..., columnN)**

## Check if a table exists or not

- Similarly, to check if the table exists when deleting, we use “if exists” with the DROP TABLE statement as follows:
- **drop table if exists table\_name**

## Check if a table exists or not

- We can also check if the table we want to access exists or not by executing the following query:
- `cursorObj.execute('SELECT name from sqlite_master WHERE type = "table" AND name = "Employee"')`
- `print(cursorObj.fetchall())`
- If the employees' table exists, it will return its name otherwise an empty array will be returned

## Drop table

- We can drop/delete a table using the DROP statement. The syntax of the DROP statement is as follows:
  - **drop table table\_name**
  - To drop a table, the table should exist in the database.
- Therefore, it is recommended to use “if exists” with the drop statement as follows:
  - **drop table if exists table\_name**

## SQLite3 exceptions

- Exceptions are the run time errors.
- In Python programming, all exceptions are the instances of the class derived from the BaseException.
- In SQLite3, we have the following main Python exceptions:
- **DatabaseError**
- Any error related to the database raises the DatabaseError.

# SQLite3 exceptions

- **OperationalError**
- This exception is raised when the database operations are failed, for example, unusual disconnection. This is not the fault of the programmers.
  
- **NotSupportedError**
- When we use some methods that aren't defined or supported by the database, that will raise the NotSupportedError exception.

# Close Connection

- Once we are done with our database, it is a good practice to close the connection. We can close the connection by using the close() method.
- To close a connection, we use the connection object and call the close() method as follows:
- `con = sqlite3.connect('mydatabase.db')`
- `#program statements`
- `con.close()`

# SQLite3 datetime

- In the Python SQLite3 database, we can easily store date or time by importing the `datetime` module. The following formats are the most common formats you can use for `datetime`:
- **YYYY-MM-DD**
- **YYYY-MM-DD HH:MM**
- **YYYY-MM-DD HH:MM:SS**
- **YYYY-MM-DD HH:MM:SS.SSS**
- **HH:MM**
- **HH:MM:SS**
- **HH:MM:SS.SSS**
- **now**

# SQLite3 datetime

- import sqlite3
- import datetime
- con = sqlite3.connect('mydatabase.db')
- cursorObj = con.cursor()
- cursorObj.execute('create table if not exists assignments(id integer, name text, date date)')
- data = [(1, "Ridesharing", datetime.date(2019, 1, 2)), (2, "Water Purifying", datetime.date(2020, 3, 4))]
- cursorObj.executemany("INSERT INTO assignments VALUES(?, ?, ?)", data)
- con.commit()

## References :

- <https://realpython.com/>
- <https://beginnersbook.com>
- <https://towardsdatascience.com/>
- <https://docs.python.org/>
- <https://www.w3schools.com/>
- <https://www.tutorialspoint.com/>
- <https://stackabuse.com/>
- <https://www.python-course.eu/>
- <https://developers.google.com/>
- <https://www.techbeamers.com/>
- <https://www.guru99.com/>
- <https://www.geeksforgeeks.org/>
- <https://www.programiz.com/>
- <https://intellipaat.com/>
- <https://data-flair.training/>
- <https://www.learnpython.org/>
- <https://www.analyticsvidhya.com/>
- <https://www.edureka.co/>
- <https://www.pythontutorialcentral.io>
- <https://likegeeks.com/>

# Title Lorem Ipsum



LOREM IPSUM DOLOR SIT AMET,  
CONSECTETUER ADIPISCING ELIT.



NUNC VIVERRA IMPERDIET ENIM.  
FUSCE EST. VIVAMUS A TELLUS.



PELLENTESQUE HABITANT MORBI  
TRISTIQUE SENECTUS ET NETUS.