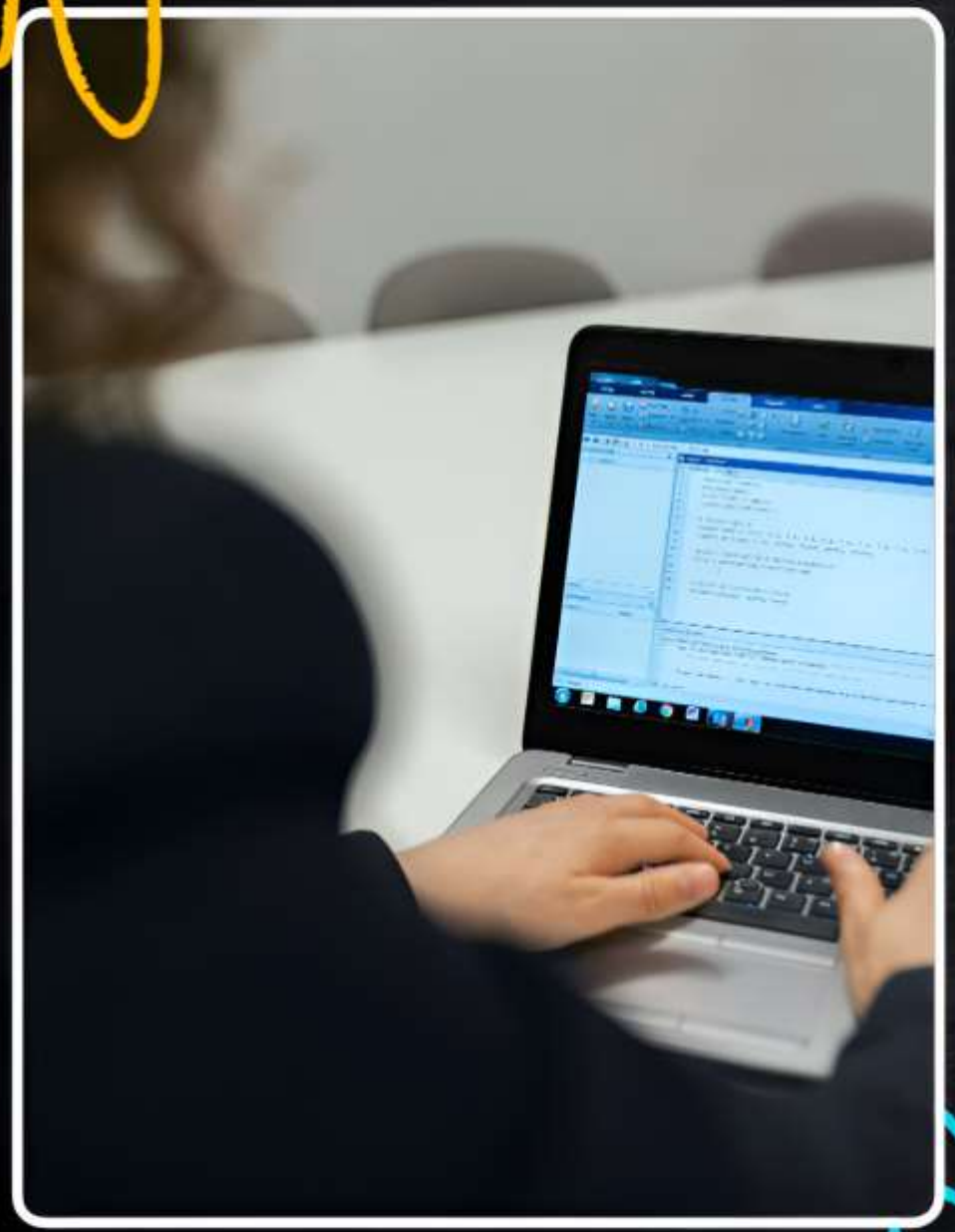


Understanding Built-in Modules in Python

Explore the key features of Python's built-in modules and import statements to enhance your programming skills effectively.

PYTHON MODULES

Understanding Modules in Python

Key Concepts and Definitions

01

Definition of a Module

A module is a Python file with a .py extension that can be imported into other scripts, encapsulating reusable code.

02

Purpose of Modules

Modules facilitate the separation of functionality within a program, making it easier to manage larger applications and promote code reusability.

03

Types of Modules

There are three primary types of modules: built-in modules, user-defined modules created by programmers, and third-party modules from external libraries.

04

Importance of Leveraging Modules

Utilizing modules is essential for effective programming in Python, as they help maintain a clean workspace and enhance code organization.

BUILT-IN MODULES

What are Built-in Modules?

Definition and Examples of Python's Essential Functionalities

Definition of Built-in Modules

Built-in modules are pre-installed components of Python that provide essential functionalities, allowing developers to utilize them without any additional installation.

Mathematical Functions

The `math` module offers a variety of mathematical functions, including trigonometric and logarithmic calculations, enhancing computational capabilities.

Operating System Interfaces

The `os` module allows interaction with the operating system, enabling file and directory manipulation, process management, and environment variable access.

System-Specific Parameters

The `sys` module provides access to system-specific parameters and functions, including command-line arguments and Python interpreter information.

Date and Time Manipulation

The `datetime` module is essential for working with dates and times, providing classes for manipulating dates, times, and intervals.

Enhancing Productivity

Utilizing built-in modules significantly boosts productivity by providing optimized and thoroughly tested functions, allowing developers to focus on core application logic.

IMPORT STATEMENTS

Understanding Import Statements in Python

Key Concepts and Syntax

Definition of Import Statement

An import statement is a syntax used in Python to include modules, enabling access to their functions and classes.

Syntax

The typical syntax of an import statement is 'import module_name', where 'module_name' is the name of the module you wish to include.

Purpose of Import Statements

Import statements facilitate code reuse and organization by allowing programmers to utilize pre-defined functions and classes from other modules.

Significance in Python Programming

Import statements are essential for leveraging the power of modules, leading to cleaner, more efficient, and maintainable code in Python applications.

MODULE IMPORTS

Different Ways to Import Modules in Python

An Overview of Import Methods

01

Basic Import

Use this method to import an entire module, granting access to all its functions and classes. Example: `import math`.

02

Import with Alias

This allows you to rename a module for easier reference. A common practice is to shorten module names. Example: `import numpy as np`.

03

Import Specific Functions

This method imports only selected functions from a module, minimizing memory usage and improving code clarity. Example: `from math import sqrt, pi`.

04

Import All Functions

Use this approach to import everything from a module, but be cautious as it can lead to conflicts. Example: `from math import *`.

BUILT-IN MODULES

Examples of Using Built-in Modules in Python

Practical Applications of Python's Built-in Modules

Math Module ⟶ Used for mathematical operations, including calculating square roots. Example: 'import math; print(math.sqrt(16)) # Output: 4.0'.

OS Module ⟶ Provides functions to interact with the operating system, such as retrieving the current working directory. Example: 'import os; print(os.getcwd())'.

Sys Module ⟶ Allows access to system-specific parameters and functions, including the Python version. Example: 'import sys; print(sys.version)'.

BUILT-IN MODULES

Overview of Commonly Used Built-in Modules

Essential Python Libraries for Efficient Development

Module	Description
math	Provides mathematical functions for complex calculations.
os	Interacts with the operating system to perform file operations.
sys	Accesses system-specific parameters and functions.
datetime	Manipulates dates and times for accurate time management.
random	Generates random numbers for simulations and testing.
json	Works with JSON data for web applications and APIs.

IMPORT BEST PRACTICES

Best Practices for Import Statements

Key Recommendations for Code Clarity

01

Import at the Top

Always place all import statements at the beginning of your file to enhance readability and maintain a clear structure.

02

Use Aliases for Clarity

When dealing with long module names, utilize aliases to improve code readability and make it easier to understand at a glance.

03

Import Only What You Need

Avoid wildcard imports to keep your code clear. Import only the necessary components to minimize confusion and enhance performance.

04

Organize Imports

Group your imports into categories: standard library, third-party, and local imports, to make your code organized and easier to navigate.

IMPORT ERRORS

Import Errors in Python

Understanding and Handling Common Errors

ModuleNotFoundError

This error indicates that a module you are trying to import does not exist in the Python environment. Ensure that the module is installed and the name is spelled correctly.

01

ImportError

An ImportError occurs when the import statement in your code fails. This could be due to a misspecified function or class name. Double-check your imports to make sure they are correct.

02

Circular Imports

Circular imports happen when two or more modules depend on each other, leading to an import loop. To resolve this, consider reorganizing your code structure to eliminate circular dependencies.

03

USER-DEFINED MODULES

Creating Your Own Modules

How to Create User-defined Modules for Code Reuse

Step 1: Create a Python File

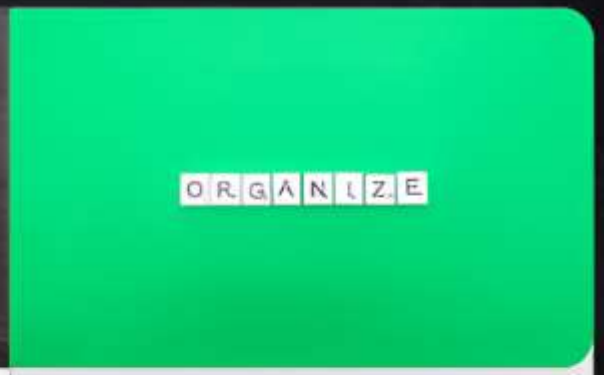
To start, save your reusable functions in a dedicated .py file, such as 'my_module.py'. This file will serve as your module.

Step 2: Define Functions or Classes

Inside your module, write the functions or classes you wish to reuse. For example, a simple greeting function can be defined as follows: ```python def greet(name): return f'Hello, {name}' ```

Step 3: Import Your Module

Once your module is created, you can import it in other scripts. Use the following syntax: ```python import my_module print(my_module.greet('Alice')) # Output: Hello, Alice ``` *This demonstrates how to call a function from your module.*



Benefits of User-defined Modules

Creating your own modules promotes better code organization and enhances code reuse, making your programming tasks more efficient and manageable.

IMPORT TECHNIQUES

Advanced Techniques for Importing Modules

Exploring Complex Import Scenarios in Python

Relative Imports

01 Allows importing modules based on the current module's location, enhancing modularity.

Dynamic Imports

02 Enables importing modules at runtime using the __import__() function for greater flexibility.

Conditional Imports

03 Facilitates importing modules based on specific runtime conditions, optimizing resource usage.

PYTHON MODULES

Key Takeaways on Python Modules

Essential Insights for Effective Code

01

Modules

Modules are essential for organizing and reusing code in Python, which enhances maintainability and collaboration.

02

Built-in Modules

Python includes several built-in modules that provide ready-to-use functionalities, saving development time and effort.

03

Import Statements

Import statements are crucial for accessing module functionalities; understanding different import methods is vital for effective coding.

04

Best Practices

Adhere to best practices by importing modules wisely, handling errors gracefully, and considering user-defined modules for custom functionalities.

PYTHON MODULES

Conclusion on Python Modules and Imports

Key Takeaways

01

Significance of Built-in Modules

Built-in modules in Python provide essential functionalities that simplify coding tasks and enhance program efficiency.

02

Effective Use of Import Statements

Understanding how to properly use import statements can significantly improve code organization and reusability.

03

Enhancing Coding Skills

Mastering module management is crucial for developers looking to enhance their coding skills and streamline their workflow.

04

Encouragement to Experiment

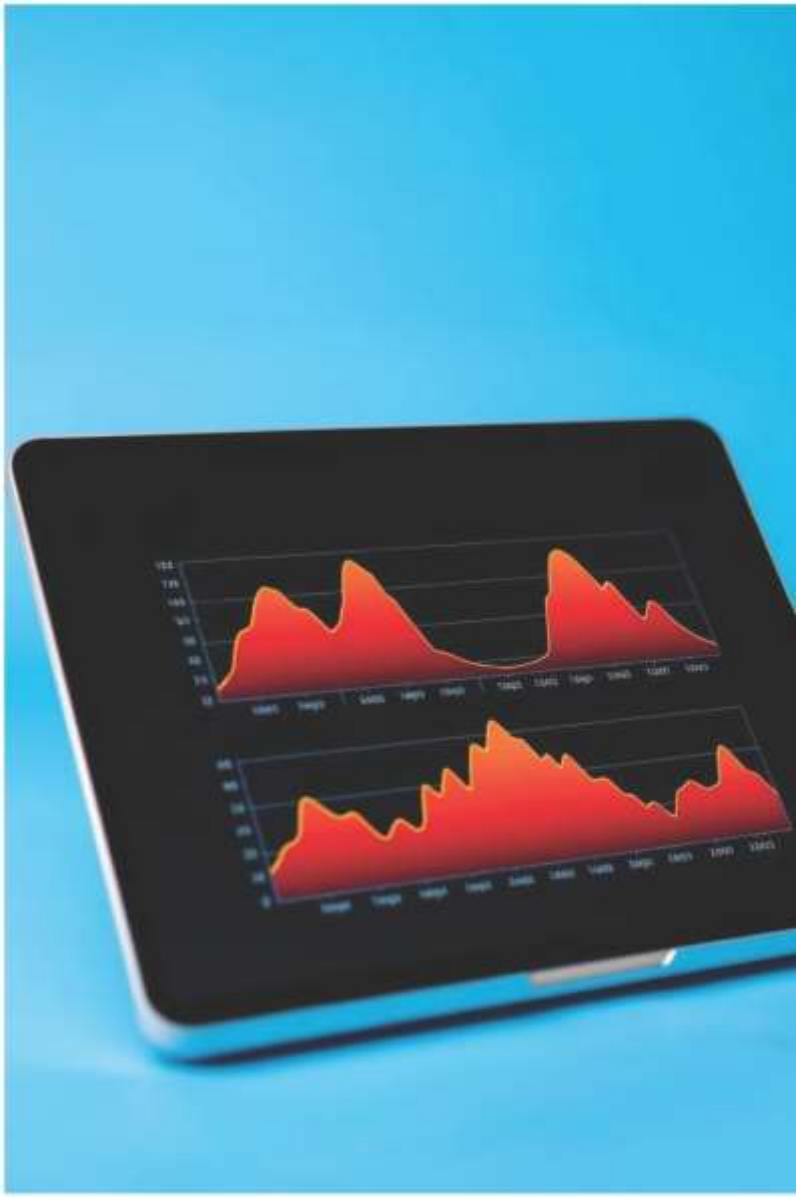
Experimenting with different modules and import techniques fosters creativity and helps you discover new solutions to coding challenges.

05

Final Message

As you apply these concepts, always remember to enjoy the coding journey. Happy coding

Packages and Date/Time in Python



Introduction to Python Packages

Code Reusability

Packages are collections of modules that provide reusable code.

Organized Structure

They help to organize and structure your Python projects.

Extensive Functionality

Packages offer a vast range of functions and classes.

Standard Library Packages

OS

Operating system interactions (file management, paths).

math

Mathematical functions (trigonometry, logarithms).

random

Generating random numbers and sequences.

json

Working with JSON data (encoding, decoding).

Installing Third-Party Packages

)_

pip

Python's package installer.

@_

Package Index

Repository of available packages (PyPI).

Working with the datetime Module

1. **Current Time**

   Get the current date and time.

2. **Date Components**

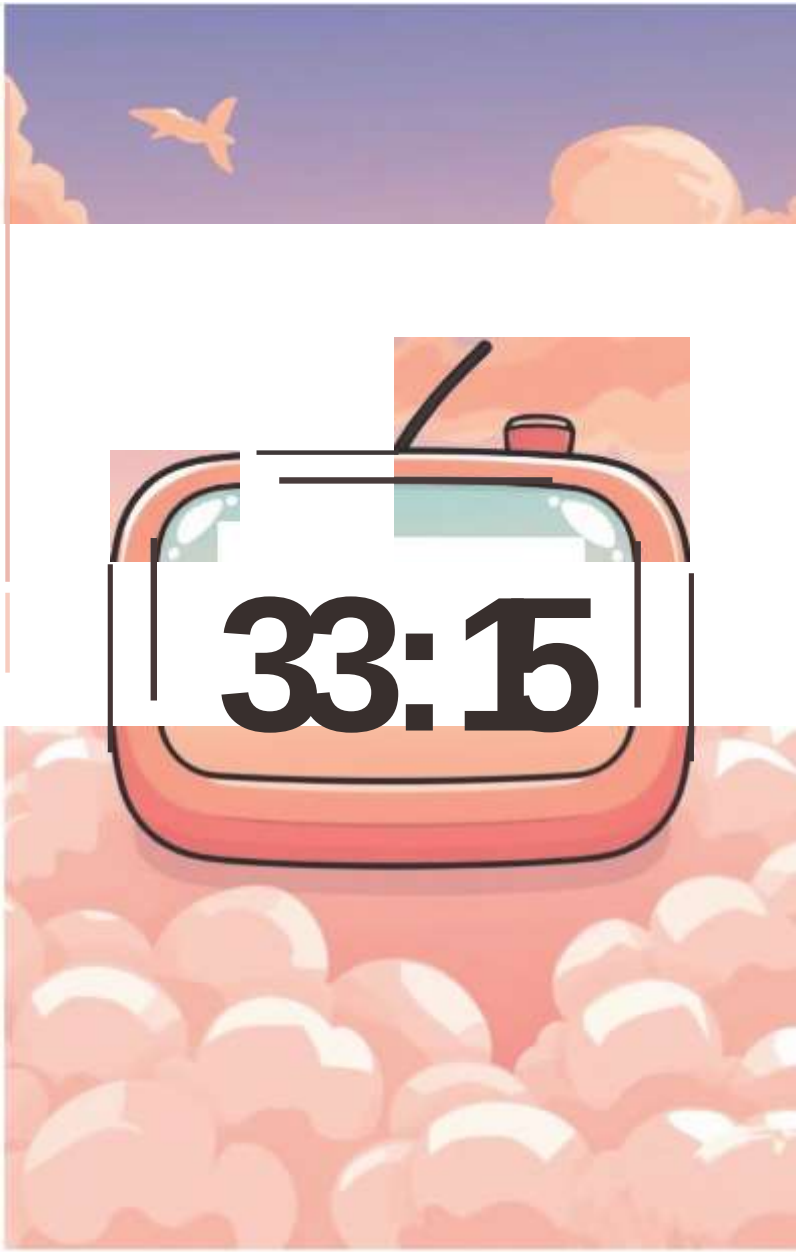
   Extract year, month, day, etc.

3. **Time Components**

   Extract hour, minute, second, etc.

4. **Date and Time Formatting**

   Customize output string format.



Date and Time Objects

datetime.date

Represents a date (year, month, day).

datetime.time

Represents a time (hour, minute, second).

datetime.datetime

Combines date and time information.

Time Delta Calculations

1

Duration

Calculate the difference between two dates or times.

2

Adding Time

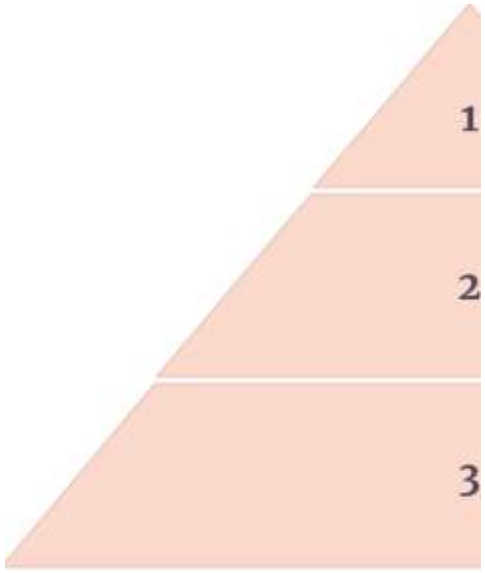
Add a time delta to a date or time object.

3

Subtracting Time

Subtract a time delta from a date or time object.

Formatting Dates and Times



strftime()

Convert datetime objects to strings.

strptime()

Parse date/time strings into datetime objects.

Custom Formats

Use format codes to create specific output.

Time Zones and Localization

1

Timezone

Represent time zones and conversions.

2

Localization

Adjust date and time formats for different regions.

3

UTC (Coordinated Universal Time)

A standard time reference.

Sample Code: Working with Dates and Times

```
import datetime

# Get current date and time now = datetime.datetime.now()

# Format the date and time
formatted_date = now.strftime("o/oY-o/om-o/od
o/oH:o/oM:o/oS")

# Print the formatted date and time print("Current Date and
Time:", formatted_date)

# Create a datetime object for a specific date birthday =
datetime.datetime(2024, 12, 25)

# Calculate the time difference between now and the birthday
time_until_birthday = birthday - now

# Print the time difference
print("Time until Birthday:", time_until_birthday)
```

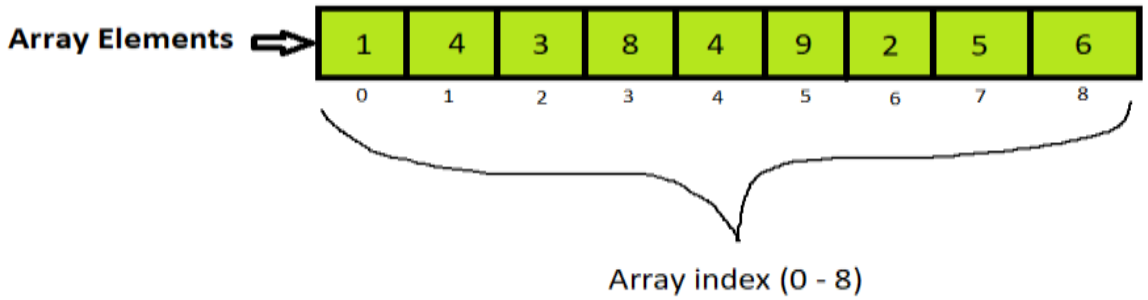
CONTENTS

- ARRAYS AND ITS OPERATIONS
- HANDLING STRINGS AND CHARACTERS
- LIST OPERATIONS

ARRAY AND ITS OPERATIONS

- WHAT IS AN ARRAY ?

An array is a collection of items stored at contiguous locations. In other words an array is a special variable which can hold more than one value at a time.



DIFFERENT OPERATIONS IN ARRAYS :

1. append( )-Adds an element at the end of the list
2. clear( )-Removes all the elements from the list.
3. copy( )-Returns a copy of the list.
4. count( )-Returns the number of elements with the specified value.
5. extend( )-Adds the element to the end of the current list.
6. index( )-Returns the index of the first element with the specified value.
7. insert( )-Adds an element at the specified position.
8. pop( )-Removes the element at the specified position.
9. remove( )-Removes the first item with the specified value.
10. reverse( )-Reverses the order of the list.
11. sort( )-Sorts the lists.

HANDLING STRINGS AND CHARACTERS

- WHAT IS A STRING ?

A string is a sequence of characters enclosed in either single quotes ‘.’ or double quotes “.”. It is used for representing textual data.

CREATING A STRING:

Strings can be created using either single quotes or double quotes.

For Example- s1=‘geeks for geeks’

s2=“geeks for geeks”

→Multi-line Strings: If we need to span multiple lines then we can use triple quotes.

For Examples: s= “ “ “ I am learning

coding from youtube ” ” ”

OPERATIONS IN STRING:

Let us consider a string s="HELLO"

→Access Characters: s[0] gives 'H'.

→String Slicing: s[1:4] gives 'ell'.

Concatenation: "Hi" + "there" gives "Hi there".

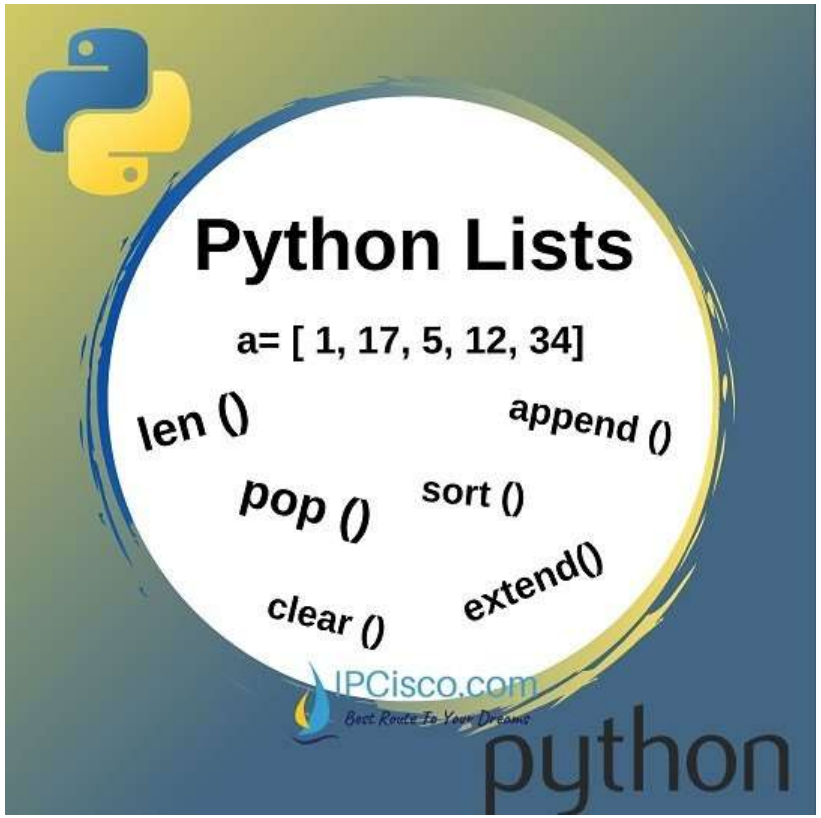
METHODS: We have a string s,

→ s.upper( )- Converts to uppercase.

→ s.lower( )- Converts to lowercase

→ s.spilt(" ")- Spilts by spaces into a list.

→ " ".join(['Hi', 'there'])- Joins list into a string.



LIST OPERATIONS

- WHAT IS A LIST ?

A list is a built-in dynamic sized array that is used to store an ordered collection of items. We can store all types of items(including another list) in a list. A list may contain mixed type of items , this is possible because a list mainly stores references at contiguous locations and actual items maybe stored at different location.

OPERATIONS IN LIST:

1. **LIST SLICING**: Extract a portion of the list.

   For example- lst[1:4] returns elements from index 1 to 3.

2. **BOUNDS**: Refers to the valid indices of a list . Accessing an index outside the bounds raises an error.

   For example- lst[10] if the list has only 5 elements.

3. **CLONING**: Creating a copy of a list.

   cloned_list = lst[:] creates a shallow copy of the list.

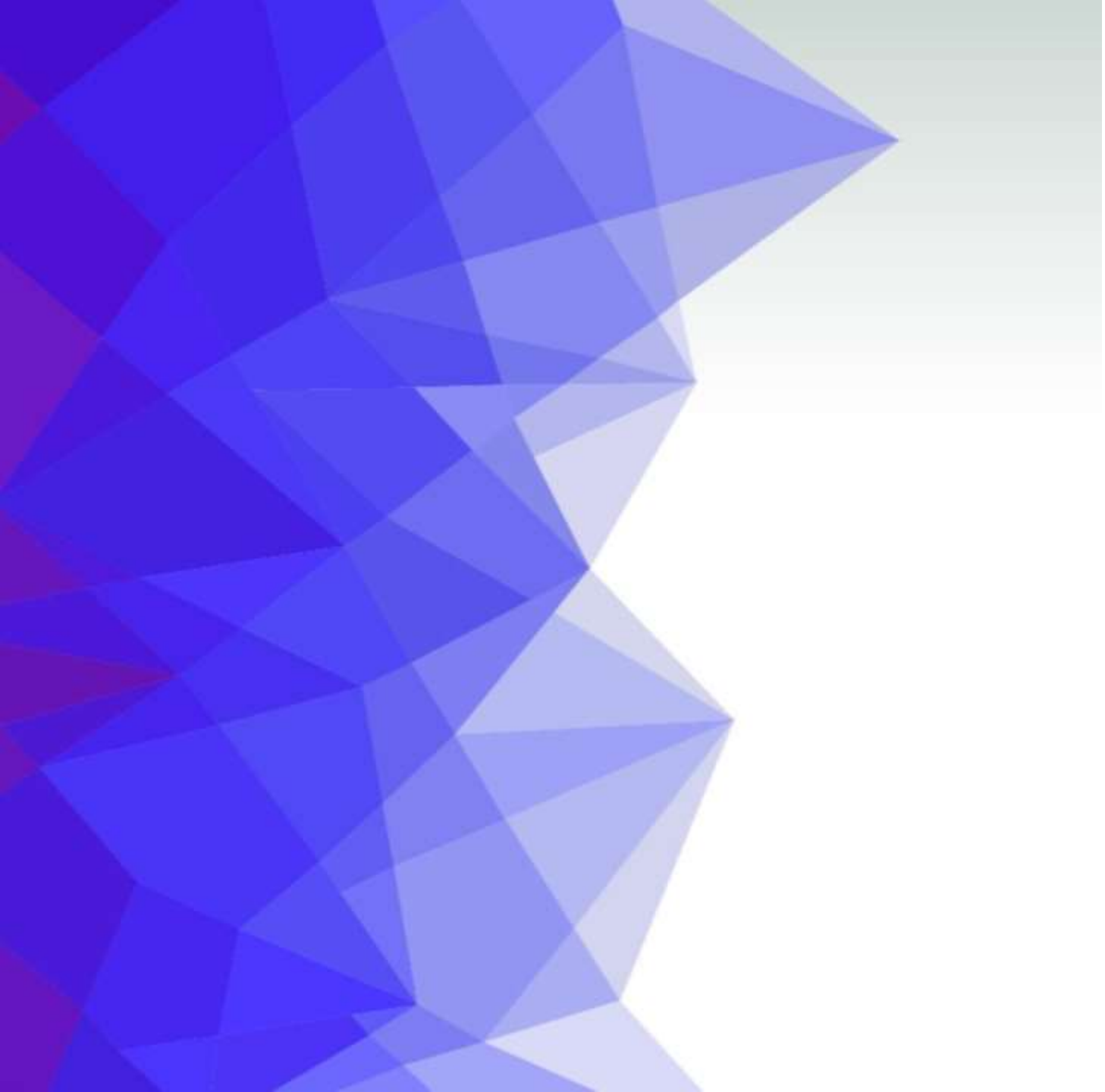
4. **NESTED LIST**: A list inside another list

   nested=[[1,2],[3,4]]- Access with nested[0][1](gives 2)

LIST METHODS :

1. append( )-Adds an element to the end of the list.
2. copy( )-Returns a shallow copy of the list.
3. clear( )-Removes all elements from the list.
4. count( )-Returns the number of times a specified element appears in the list.
5. extend( )-Adds elements from another list to the end of the current list.
6. index( )-Returns the index of the first occurrence of a specified element.
7. insert( )-Inserts an element at a specified position.
8. pop( )-Removes and returns the elements at the specified position(or the last element if no index is specified).

1. remove( )-Removes the first occurrence of a specified element.
2. reverse( )-Reverses the order of the element in the list.
3. sort( )-Sorts the list in ascending order(by default).



ADDING ELEMENT & MUTABILITY IN PYTHON

MAP()

The map() function applies a given function to every item in a list (or any iterable) and returns a map object (an iterator) .

SYNTAX :

map(function, iterable)

EXAMPLE :

numbers =[1,2, 3,4]

squared =map(lambda x:x **2, numbers) print(list(squared))

Output :[1,4,9,16]

FILTER()

The filter() function is used to filter items from an iterable based on a condition (function that returns True or False) .

SYNTAX :

filter(function, iterable)

EXAMPLE :

- numbers =[1,2, 3,4, 5,6]
- even =filter(lambda x:x % 2 ==0, numbers) print(list(even))

- Output :[2,4,6]

APPEND()

The append() method in Python is used to add a single item to the end of list. This method modifies the original list and does not return a new list.

Syntax of append() method :

list.append(element)

Parameter :

Element: The item to be appended to the list. This can be of any data type(integer, string, list, etc.) ,the parameter is mandatory and omitting it can cause an error.

Return Type :

The append() method does not return any value, it just modifies the original list in place.

EXTEND()

The Python List extend() method adds items of an iterable (list, tuple, dictionary, etc) at the end of a list.

Syntax of extend() method :

list_name.extend(iterable)

Parameter :

Iterable: Any iterable (list, set, tuple, etc.)

Return Type :

Python list sort() returns none.

COUNT()

The count() method is used to find the number of times a specific element occurs in a list. It is very useful in scenarios where we need to perform frequency analysis on the data.

Syntax of count() method :

list_name.count(value)

Parameter :

list_name: The list object where we want to count an element.

value: The element whose occurrences need to be counted.

Return Type :

The count() method returns an integer value, which represents the number of times the specified element appears in the list.

INDEX()

List index() method searches for a given element from the start of the list and returns the position of the first occurrence.

Syntax of count() method :

list_name.index(element, start, end)

Parameter :

element – The element whose lowest index will be returned.

start (Optional) – The position from where the search begins.

end (Optional) – The position from where the search ends.

Return Type :

Returns the lowest index where the element appears.

INSERT()

Python List insert() method inserts an item at a specific index in a list.

Syntax of insert() method :

list_name.insert(index, element)

Parameter :

index: the index at which the element has to be inserted.

element: the element to be inserted in the list.

Return Type :

The insert() method returns None. It only updates the current list.

SORT()

The sort() method in Python is a built-in function that allows us to sort the elements of a list in ascending or descending order and it modifies the list in place which means there is no new list created.

Syntax of insert() method :

list_name.sort(key=None, reverse=False)

Parameter :

Key (Optional): This is an optional parameter that allows we to specify a function to be used for sorting. For example, we can use the len() function to sort a list of strings based on their length.

Reverse (Optional): This is an optional Boolean parameter. By default, it is set to False to sort in ascending order. If we set reverse=True, the list will be sorted in descending order.

REVERSE()

The reverse() method is an inbuilt method in Python that reverses the order of elements in a list.

Syntax of reverse() method :

list_name.reverse()

Parameter :

It doesn't take any parameters.

Return Type :

It doesn't return any value.

REMOVE()

Python list remove() function removes the first occurrence of a given item from list.

Syntax of reverse() method :

list_name.remove(obj)

Parameter :

obj: object to be removed from the list.

Return Type :

The method does not return any value but removes the given object from the list.

CLEAR()

This method modifies the list in place and removes all its elements

Syntax of reverse() method :

list_name.clear()

Parameter :

The clear() method doesn't take any parameters.

Return Type :

The clear() method only empties the given list. It doesn't return any value.

POP()

The list pop() method removes the item at the specified index. The method also returns the removed item.

Syntax of reverse() method :

list_name.pop(index)

Parameter :

The pop() method takes a single argument (index).

The argument passed to the method is optional. If not passed, the default index -1 is passed as an argument (index of the last item).

If the index passed to the method is not in range, it throws IndexError: pop index out of range exception.

Return Type :

The pop() method returns the item present at the given index. This item is also removed from the list.