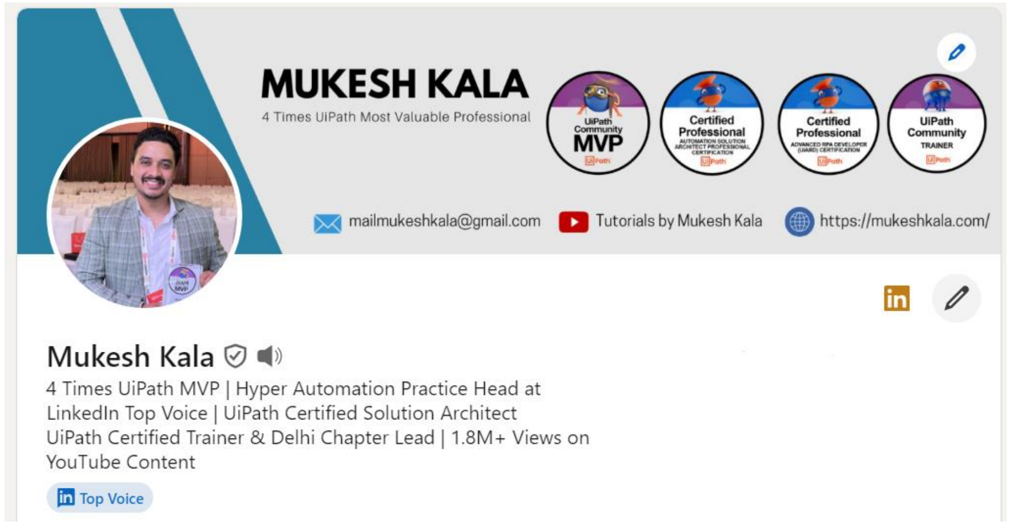
python™

Beginners Notes



MUKESH KALA

4 Times UiPath Most Valuable Professional

mailmukeshkala@gmail.com Tutorials by Mukesh Kala https://mukeshkala.com/

Mukesh Kala

4 Times UiPath MVP | Hyper Automation Practice Head at LinkedIn Top Voice | UiPath Certified Solution Architect UiPath Certified Trainer & Delhi Chapter Lead | 1.8M+ Views on YouTube Content

Top Voice

My LinkedIn Profile – Happy to connect with you !

https://www.linkedin.com/in/mukeshkala/

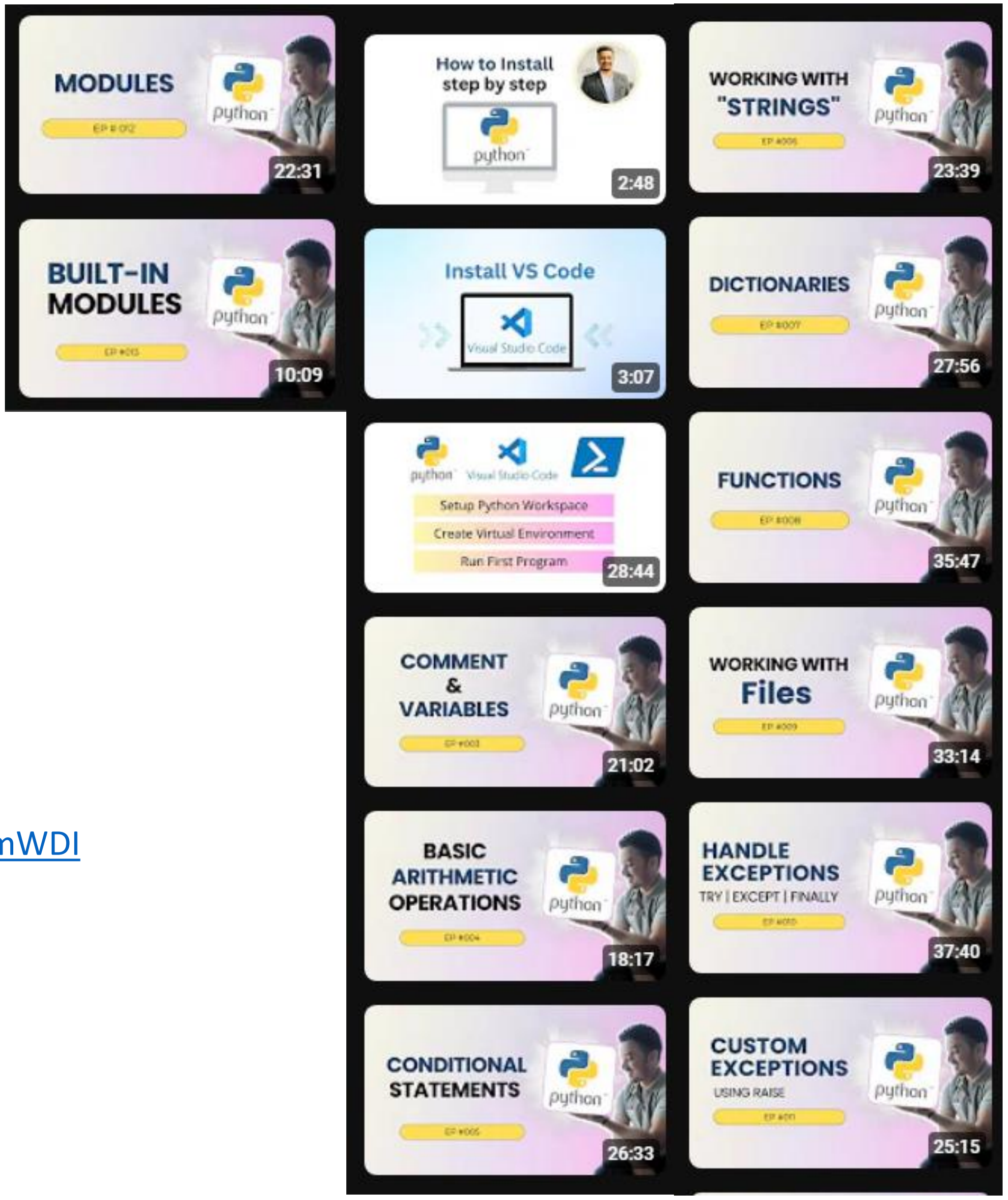
python™

For Better Understanding

1. Watch the Step-by-Step videos.
2. Open Editor and Practice along.
3. Complete the assignments

https://www.youtube.com/playlist?list=PLEYSwx3duQ2BWORdGXiaAl63di3hKmWDI

Click here for playlist



Chapter 1 : Introduction and Setup with VS code

Video Links :

https://youtu.be/-gNyDVDNrzk?si=fpH23bNdUuV2PAEW

https://youtu.be/_xgNOfbEiz4?si=4rK7yZYRvrEVKX5Z

Introduction to Python

- Python was created by Guido van Rossum and first released in 1991.
- The language was designed to emphasize code readability and simplicity.
- Python's name is derived from the British comedy series "Monty Python's Flying Circus," reflecting its creator's aim to make coding fun.

Key Features:

- **Easy to Read, Write, and Learn**: Python has a simple syntax similar to English, which makes it easy to read and understand.

- **Interpreted Language**: Python code is executed line by line, which makes debugging easier.

- **Dynamically Typed**: You don’t need to declare the type of variable because Python automatically assigns the data type during execution.

- **Versatile**: Python can be used for web development, data science, artificial intelligence, machine learning, automation, and more.

- **Extensive Standard Library**: Python has a vast standard library that includes modules and packages for various tasks.

- **Community Support**: Python has a large and active community, providing a wealth of resources and libraries.

Why Python is Popular

- **Readability and Simplicity**: Python’s syntax is clean and easy to understand, making it an excellent choice for beginners.

- **Versatility**: Python’s ability to handle various tasks makes it a preferred language for many domains.

- **Rich Ecosystem:** Python’s ecosystem includes numerous frameworks and libraries, such as Django for web development, TensorFlow for machine learning, and Pandas for data analysis.

- **Community and Support**: The large community contributes to a rich repository of tutorials, guides, and third-party modules, making it easy to find support and resources.

Install Python

1. Download Python Installer:

- Go to the official Python website https://www.python.org/
- Click on the "Downloads" tab and select "Download Python 3.x.x".

2. Run the Installer:

- Locate the downloaded installer in your system and double-click to run it.
- Check the box that says "Add Python 3.x to PATH".
- Click "Install Now" for a standard installation.

3. Verify Installation:

- Open Command Prompt.
- Type python --version and press Enter. You should see the Python version displayed.

https://youtu.be/-gNyDVDNrzk?si=okdf3j22FP0d1QIt

Chapter 2 : Set up Python | Activate Virtual Environment

Video Links :

https://youtu.be/Uwg1FP5bm-U?si=juLc5xLfTp2-_Nzp

Tutorials by Mukesh Kala

Setting up the Environment

Introduction to IDEs:

Integrated Development Environments (IDEs) provide comprehensive facilities to programmers for software development, including a code editor, debugger, and build automation tools.

PyCharm

Installation: Download from JetBrains.
Features: Smart code navigation, code refactoring, built-in terminal, version control, and support for web frameworks.

VSCode (Visual Studio Code)

Installation: Download from Visual Studio Code.
Features: Lightweight, extensible through plugins, integrated terminal, version control integration, and IntelliSense.

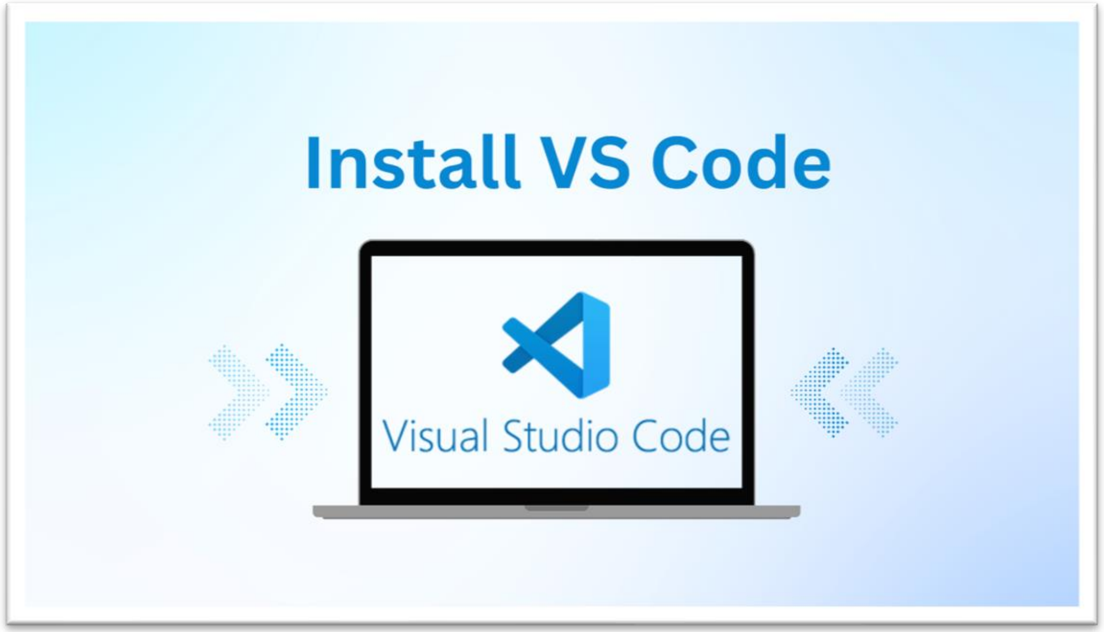
Jupyter Notebooks

Installation: Install via Anaconda Distribution: Download from Anaconda.
Or install using pip: pip install notebook.
Features: Interactive computing environment, support for live code, visualizations, and narrative text.

Install Vs Code

VSCode (Visual Studio Code)

Installation: Download from Visual Studio Code.
Features: Lightweight, extensible through plugins, integrated terminal, version control integration, and IntelliSense.



https://youtu.be/_xgNOfbEiz4?si=GnxAKxSIO4yWKFj4

Tutorials by Mukesh Kala

Why Workspace in VS Code

Creating a workspace in Visual Studio Code (VS Code) isn't strictly necessary for running Python scripts, but it offers several benefits that can enhance your development experience.

Organized Project Structure: keep all your project files and folders organized in one place and multiple folders in a single workspace,

Consistent Configuration: You can define workspace-specific settings that override the default settings, such as editor preferences, Python interpreter paths, and formatting rules.
You can configure extensions to behave differently for different projects, enhancing productivity.

Environment Management: Workspaces help in managing virtual environments for specific projects, ensuring that each project uses its own dependencies without conflicts.

Version Control Integration: Workspaces integrate seamlessly with version control systems like Git,

Extensions and Customization: Extensions can be configured to work specifically for your workspace, providing language support, linters, formatters, and more tailored to your project's needs.

- Open VS Code
- Launch Visual Studio Code on your machine.
- Create a New Workspace
- Click on File > Open Folder
- Navigate to the directory where you want to create your project and
- Click on New Folder.Name the folder (e.g., MyPythonProject) and click Open.

Virtual environment in Python

- A virtual environment in Python is a self-contained directory that contains a Python installation for a particular version of Python, plus a number of additional packages.
- It's a tool to keep dependencies required by different projects in separate places, by creating virtual Python environments for them.
- This is one of the most important tools that most Python developers use.
- A virtual environment isolates the dependencies for different projects. This means that the packages installed in one environment won't affect other projects.
- Avoid Conflicts: Different projects might require different versions of the same package. By using virtual environments, you can manage these dependencies separately and avoid version conflicts.
- Using a virtual environment ensures that the same dependencies (and their versions) are used every time the project is set up. This is crucial for consistency, especially when working in teams or deploying applications.
- You can easily replicate the environment by sharing a requirements.txt file, which lists all the packages and their versions. Anyone can create the same environment by installing the listed packages.
- Each project can have its own environment tailored to its specific needs without affecting the global Python installation or other projects.
- Virtual environments make it easy to set up and manage the Python environment for different projects.

Chapter 3 : Comments and Variables

Video Links :

https://youtu.be/qHAHb7TpEyo?si=rjM7_TGNYpXBAbVu

Tutorials by Mukesh Kala

Comments: Single-Line and Multi-Line

Single-Line Comments

Use the # symbol to add comments.

```
# This is a single-line comment
print("Hello, World!")  # This is an inline comment
```

Multi-Line Comments

Use triple quotes (''' or """) for multi-line comments.

```
"""
This is a multi-line comment.
It spans multiple lines.
"""
print("Hello, World!")
```

Declaring and Initializing Variables, Naming Conventions

Declaring and Initializing Variables:
No need to declare variables explicitly; just assign a value.

```
message = "Hello, World!"
number = 42
pi = 3.14159
```

Variable Naming Conventions:
- Use meaningful names.
- Start with a letter or underscore, followed by letters, digits, or underscores.
- Case-sensitive.

```
user_name = "Alice"
user_age = 30
_hidden_variable = "This is hidden"
```

Chapter 4 : Arithmetic and Assignment Operators

Video Links :

https://youtu.be/-pmlJdrYXEc?si=shgHAkPmBSgok492

Tutorials by Mukesh Kala

Code

```
# Arithmetic operations
a = 10
b = 3

addition = a + b
subtraction = a - b
multiplication = a * b
division = a / b
modulus = a % b
exponentiation = a ** b
floor_division = a // b

print("Addition:", addition)
print("Subtraction:", subtraction)
print("Multiplication:", multiplication)
print("Division:", division)
print("Modulus:", modulus)
print("Exponentiation:", exponentiation)
print("Floor Division:", floor_division)
```

```
# Assignment operations
x = 5
print("Initial x:", x)
x += 3
print("After x += 3:", x)
x -= 2
print("After x -= 2:", x)
x *= 4
print("After x *= 4:", x)
x /= 3
print("After x /= 3:", x)
```

Chapter 4 : Control Structures in Python

Video Links :

https://youtu.be/g_S7VT1dNW4?si=K7WSiM7vUArKISrH

Tutorials by Mukesh Kala

Brief Overview of Control Structures

- Control structures are fundamental concepts in programming that dictate the order in which statements and instructions are executed.
- They enable programmers to control the flow of execution based on certain conditions or repetitions.
- Common control structures include conditional statements (if, else, elif), loops (for, while), and branching statements (break, continue).

Importance of Conditionals in Programming:

- Conditional statements allow the program to make decisions and execute specific code blocks based on whether certain conditions are met.
- They are essential for creating dynamic and responsive programs that can handle different inputs and situations.
- By using conditionals, you can direct the program's flow, making it more interactive and capable of solving complex problems.

If Else If and Else

```
if condition:
    # Code to execute if condition is True
```

```
if condition1:
    # Code to execute if condition1 is True

elif condition2:
    # Code to execute if condition2 is True

else:
    # Code to execute if none of the conditions are True
```

Practical Example: Grading System

```
score = 85

if score >= 90:
    grade = 'A'
elif score >= 80:
    grade = 'B'
elif score >= 70:
    grade = 'C'
elif score >= 60:
    grade = 'D'
else:
    grade = 'F'

print(f"The grade is {grade}")
```

Nesting Conditionals

```
age = 20
citizen = True

if age >= 18:
    if citizen:
        print("Eligible to vote")
    else:
        print("Not a citizen, cannot vote")
else:
    print("Too young to vote")
```

Best Practices

- Keep conditions simple and readable
- Avoid deep nesting
- Use meaningful variable names
- Comment your code for clarity

Assignment

Write a Python program that checks the temperature and prints appropriate messages.

Conditions:

- If the temperature is above 30, print "It's hot!“
- If the temperature is between 20 and 30, print "It's warm.“
- If the temperature is between 10 and 20, print "It's cool.“
- If the temperature is below 10, print "It's cold!"

Chapter 5 : Manipulating Strings in Python

Video Links :

https://youtu.be/Xq3VoXrlWLU?si=5M9n4QCqKdtwGDQq

What are strings?

- A string is a sequence of characters.
- Strings are immutable in Python

It means that once a string is created, its content cannot be changed or modified. Any operation that seems to modify a string will actually create a new string object rather than altering the original one.

```
text = "Hello"
text[0] = "h"  # This will raise an error
```

Importance of string manipulation

Essential for text processing, data analysis, web development, and more.

Indexing Strings

- Accessing individual characters using their index.
- Zero-based indexing (first character is at index 0).

```
text = "Hello, World!"
first_char = text[0]  # 'H'
last_char = text[-1]  # '!'
```

Slicing Strings

- Extracting substrings using a range of indices.
- Syntax: string[**start:end**] (end index is exclusive).
- Step value for advanced slicing: string[**start:end:step**]

```
text = "Hello, World!"
hello = text[0:5]  # 'Hello'
world = text[7:12]  # 'World'
```

String Methods

- Common methods:upper(), lower(), strip(), replace(), find(), split(), join().

```
text = "  Hello, World!  "
uppercase_text = text.upper()  # '  HELLO, WORLD!  '
stripped_text = text.strip()  # 'Hello, World!'
replaced_text = text.replace("World", "Python")  # '  Hello, Python!  '
```

Assignments

Given a string text = "The quick brown fox jumps over the lazy dog",

extract and print the following:

- The word "quick“
- The word "lazy“
- The phrase "brown fox jumps"

Code from Video

```
# Indexing strings
text = "Hello, World!"

# Accessing individual characters
first_char = text[0]
last_char = text[-1]

print(f"First character: {first_char}")
print(f"Last character: {last_char}")
```

```
# Slicing strings
text = "Hello, World!"

# Extracting substrings
hello = text[0:5]
world = text[7:12]
reversed_text = text[::-1]

print(f"Substring 'Hello': {hello}")
print(f"Substring 'World': {world}")
print(f"Reversed text: {reversed_text}")
```

```
# String methods
text = "  Hello, World!  "

# Using string methods
uppercase_text = text.upper()
lowercase_text = text.lower()
stripped_text = text.strip()
replaced_text = text.replace("World", "Python")
found_index = text.find("World")
split_text = text.split(",")
joined_text = " ".join(split_text)

print(f"Uppercase: {uppercase_text}")
print(f"Lowercase: {lowercase_text}")
print(f"Stripped: {stripped_text}")
print(f"Replaced: {replaced_text}")
print(f"Found index of 'World': {found_index}")
print(f"Split text: {split_text}")
print(f"Joined text: {joined_text}")
```

python™

Chapter 6 : Data Structures - Dictionaries

Video Links :

https://youtu.be/2uRYj2iIcZg?si=SFhNrJvfEywunEGj

Tutorials by Mukesh Kala

Overview of Dictionaries

Definition:
A dictionary in Python is a collection of key-value pairs where each key is unique.
It is similar to a real-life dictionary where you have words (keys) and their definitions (values).

Syntax:
Dictionaries are defined using curly braces {} with key-value pairs separated by colons

```
student = {
    "name": "Alice",
    "age": 25,
    "courses": ["Math", "Computer Science"]
}
```

Characteristics

Keys:

Must be unique and immutable (e.g., strings, numbers, tuples).
Keys are used to access values in the dictionary.

Values:

Can be of any data type (strings, numbers, lists, other dictionaries, etc.).
Values can be repeated, but the keys must be unique.

Mutable:

Dictionaries are mutable, meaning you can change, add, or remove key-value pairs after the dictionary is created.

Unordered:

In Python versions before 3.7, dictionaries were unordered collections. From Python 3.7 onwards, dictionaries maintain the insertion order of keys.

Creating and Reading Dictionaries

```
student = {
    "name": "Alice",
    "age": 25,
    "courses": ["Math", "Computer Science"]
}
```

```
name = student["name"]
age = student["age"]
```

Modifying Dictionaries

Adding or Updating Key-Value Pairs

```
student["age"] = 26  # Update value
student["grade"] = "A"  # Add new key-value pair
```

Removing Key-Value Pairs

```
del student["courses"]
age = student.pop("age")
```

Example Code

```
# Creating a dictionary
student = {
    "name": "Alice",
    "age": 25,
    "courses": ["Math", "Computer Science"]
}

# Accessing values
print(student["name"])  # Output: Alice
print(student.get("age"))  # Output: 25

# Adding and updating key-value pairs
student["age"] = 26  # Update existing key
student["grade"] = "A"  # Add new key-value pair

# Removing key-value pairs
del student["courses"]

# Checking if a key exists
has_grade = "grade" in student
print(f"Grade key exists: {has_grade}")  # Output: Grade key exists: True

# Display the updated dictionary
print(student)
```

Chapter 7 : Functions in Python

Video Links :

https://youtu.be/gkFWi7XDtOM?si=pNgLXiS_bBMHJwlH

Tutorials by Mukesh Kala

What are functions?

- Functions are reusable blocks of code designed to perform a specific task.
- Think of them as mini-programs within your larger program.
- Reusable blocks of code designed to perform a specific task.
- Improve modularity and code reusability.

Importance of functions in programming

- **Modularity**: Breaking down a program into smaller, manageable parts (functions).
- **Reusability**: Functions can be called multiple times within a program, reducing code duplication.
- **Clear Structure**: Functions help organize code into logical sections, making it more readable.
- **Self-Documenting**: Well-named functions describe what they do, which makes the code easier to understand without extensive comments.

Defining Functions

- Functions are defined using the def keyword.
- They can take arguments and return values.
- Functions can have different types of arguments: **positional, keyword, default, and variable-length**.

```
def function_name(parameters):
    # Function body
    return value
```

Scope and Lifetime

- Local and global variables
- The scope of a variable determines where it can be accessed.
- Lifetime of a variable is the duration for which it exists in memory.

Defining Functions (Explanation)

```
# Defining a simple function
def greet(name):
    return f"Hello, {name}!"

print(greet("Alice"))  # Output: Hello, Alice!
```

- The f in f"Hello, {name}!" signifies that the string is an f-string, a feature that allows for easy and readable string formatting.
- F-strings evaluate expressions inside curly braces {} and embed the results into the string.
- They offer a concise and efficient way to create formatted strings in Python.

Function Arguments (Code from Video)

```
# Function with positional and default arguments
def add(a, b=10):
    return a + b

print(add(5))  # Output: 15
print(add(5, 20))  # Output: 25

# Function with variable-length arguments
def multiply(*args):
    result = 1
    for num in args:
        result *= num
    return result

print(multiply(1, 2, 3, 4))  # Output: 24

# Function with keyword arguments
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

display_info(name="Alice", age=25, city="New York")
```

Scope and Lifetime

```
# Scope and lifetime of variables
x = 10  # Global variable

def example():
    x = 5  # Local variable
    print("Inside function:", x)

example()
print("Outside function:", x)
```

Practical Code from Video

```
def is_even(n):
    return n % 2 == 0

print(is_even(4))  # Output: True
print(is_even(5))  # Output: False
```

```
def convert_temperature(celsius):
    return (celsius * 9/5) + 32

print(convert_temperature(0))  # Output: 32.0
print(convert_temperature(100))  # Output: 212.0
```

```
def max_of_three(a, b, c):
    return max(a, b, c)

# Test the function
print(max_of_three(3, 7, 5))  # Output: 7
```

Assignment

```
def convert_temperature(celsius):
    return (celsius * 9/5) + 32

print(convert_temperature(0))  # Output: 32.0
print(convert_temperature(100))  # Output: 212.0
```

Function to Calculate the Area of a Circle:

Write a function named circle_area that takes one argument radius and returns the area of a circle.

Use the formula: area = π * radius^2 (use 3.14 for π).

Chapter 8 : File Handling

Video Links :

https://youtu.be/m1XSUfKRJOk?si=PVHjYF7KhVl5iGUv

Tutorials by Mukesh Kala

What is File Handling?

- File handling refers to the process of creating, opening, reading, writing, and closing files in a program.

- It is a crucial skill for any programmer, as it allows you to work with external data and store results persistently..

Importance of File handling

- Working with files enables programs to save data, process external data sources, generate reports, and much more.

- Python provides a built-in way to handle files using simple and intuitive commands.

Opening Files with open()

- The open() function is used to open a file in Python.
- It requires at least one argument: the file name (and optionally, the mode in which to open the file).

```
file = open("filename.txt", "mode")
```

Modes:

- 'r': Read mode (default). Opens the file for reading.
- 'w': Write mode. Opens the file for writing (creates a new file or truncates an existing file)
- 'a': Append mode. Opens the file for appending (writes data at the end of the file).
- 'b': Binary mode. Used with 'r', 'w', or 'a' to work with binary files.

Reading Files

read(): Reads the entire content of the file

```
content = file.read()
```

readline(): Reads one line at a time.

```
line = file.readline()
```

readlines(): Reads all lines in the file and returns them as a list.

```
lines = file.readlines()
```

Writing to Files

write(): Writes a string to the file.

```
file.write("Hello, World!\n")
```

writelines(): Writes a list of strings to the file.

```
file.writelines(["Hello, World!\n", "Welcome to Python file handling.\n"])
```

Using with Statement for File Handling

The with statement ensures that files are properly closed after their suite finishes, even if an exception is raised.

```
with open("filename.txt", "mode") as file:
    # Perform file operations
```

The file is automatically closed when the block of code inside with is exited.

```
with open("example.txt", "r") as file:
    content = file.read()
    print(content)
# No need to explicitly close the file
```

Code from video

Create a text file named input.txt with the following content:

```
Hello, this is a test file.
It contains multiple lines of text.
Python file handling is easy and powerful.
```

```python
def process_file(input_file, output_file):
    try:
        with open(input_file, "r") as file:
            lines = file.readlines()
            line_count = len(lines)
            word_count = sum(len(line.split()) for line in lines)

        with open(output_file, "w") as file:
            file.write(f"Line count: {line_count}\n")
            file.write(f"Word count: {word_count}\n")

        print("File processed successfully.")
    except FileNotFoundError:
        print(f"Error: The file {input_file} was not found.")
    except IOError:
        print(f"Error: An error occurred while processing the file.")

# Run the function
process_file("input.txt", "output.txt")
```

- Read the file: The program reads all lines from input.txt.
- Process the file: It counts the number of lines and words.
- Write the results: The results are written to output.txt.

Assignment

Write a program that reads a file containing Your Name
and
writes “Welcome Mukesh Kala” to a new file.

Input file example : Mukesh Kala
Output file example : Welcome Mukesh Kala

Chapter 9 : Exceptionn Handling and Exceptions with raise

Video Links :

https://youtu.be/iryi9jRODd4?si=mI3L3tgb2xh7wQYd

https://youtu.be/ZsXISmVE2Mc?si=d5SrXyXKsVO-DBkf

What is Exception Handling?

What is Exception Handling?

- Exception handling is the process of responding to and managing Exceptions or exceptional conditions that occur during the execution of a program.

- It ensures that the program can handle unexpected situations gracefully, preventing crashes and providing meaningful feedback to the user.

Why is Exception Handling Important?

- Exceptions are inevitable in any program, especially when dealing with user inputs, file operations, or network requests.

- Proper Exception handling improves the robustness of the application by allowing it to manage and respond to Exceptions effectively, enhancing the overall user experience.

Basic Exception Handling with try and except

- try and except blocks are used to catch and handle exceptions (Exceptions) that occur during the execution of a program.
- The code that might raise an exception is placed inside the try block, and the code that handles the exception is placed inside the except block.

```
try:
   # Code that may raise an exception

except ExceptionType:
   # Code to handle the exception
```

```
try:
    result = 10 / 0  # This will raise a ZeroDivisionException
except ZeroDivisionException:
    print("Exception: Division by zero is not allowed.")
```

Using else and finally Blocks

- else block: Executes if the try block does not raise any exceptions.finally block: Executes regardless of whether an exception was raised or not.
- It is often used for cleanup actions (e.g., closing files or releasing resources).

```
try:
   # Code that may raise an exception
except ExceptionType:
   # Code to handle the exception
else:
   # Code to execute if no exception occurs
finally:
   # Code to execute no matter what (optional)
```

```
try:
    result = 10 / 2
except ZeroDivisionException:
    print("Exception: Division by zero is not allowed.")
else:
    print(f"Result: {result}")
finally:
    print("Execution completed.")
```

Raising Exceptions with raise

- The raise statement allows you to raise an exception manually if a certain condition is met.
- This is useful for enforcing certain conditions in your code or when a specific Exception condition needs to be flagged.

```
raise ExceptionType("Custom Exception
message")
```

```
def divide(a, b):
    if b == 0:
        raise ValueException("Cannot divide by zero.")
    return a / b

try:
    result = divide(10, 0)
except ValueException as e:
    print(f"Exception: {e}")
```

Code from Video

```
def add(a, b):
    return a + b

def subtract(a, b):
    return a - b

def multiply(a, b):
    return a * b

def divide(a, b):
    if b == 0:
        raise ZeroDivisionException("Cannot divide by zero.")
    return a / b
```

```
def calculator():
    try:
        a = float(input("Enter the first number: "))
        b = float(input("Enter the second number: "))
        operation = input("Enter the operation (+, -, *, /): ")

        if operation == "+":
            result = add(a, b)
        elif operation == "-":
            result = subtract(a, b)
        elif operation == "*":
            result = multiply(a, b)
        elif operation == "/":
            result = divide(a, b)
        else:
            raise ValueException("Invalid operation.")

        print(f"The result is: {result}")

    except ValueException as ve:
        print(f"Exception: {ve}")
    except ZeroDivisionException as zde:
        print(f"Exception: {zde}")
    finally:
        print("Thank you for using the calculator.")

# Run the calculator program
calculator()
```

Code from Video – Calculator Code

```
def calculator():
    try:
        a = float(input("Enter the first number: "))
        b = float(input("Enter the second number: "))
        operation = input("Enter the operation (+, -, *, /): ")

        if operation == "+":
            result = add(a, b)
        elif operation == "-":
            result = subtract(a, b)
        elif operation == "*":
            result = multiply(a, b)
        elif operation == "/":
            result = divide(a, b)
        else:
            raise ValueException("Invalid operation.")

        print(f"The result is: {result}")

    except ValueException as ve:
        print(f"Exception: {ve}")
    except ZeroDivisionException as zde:
        print(f"Exception: {zde}")
    finally:
        print("Thank you for using the calculator.")

# Run the calculator program
calculator()
```

Assignment

Re Create the Same Calculator Function

Chapter 10 : Modules in Python

Video Links :

https://youtu.be/oYg-mLf0cL4?si=TTsSlz4LunrNQ4Fi

What are Modules?

- A module is a file containing Python definitions and statements.
- The file name is the module name with the suffix .py added.
- Modules allow you to organize your code into manageable sections.

Why are Modules Important?

- Modularity: Modules and packages promote modular programming, where you can divide your code into separate components that can be developed, tested, and maintained independently.

- Reusability: Code written in modules can be reused across different projects, saving time and reducing errors.

- Maintainability: Organizing your code into modules and packages makes it easier to maintain and debug, as related functions and classes are grouped together.

Creating and Importing Modules

A module is simply a Python file (.py) that contains definitions, functions, classes, and executable code.

Creating a Module:

- To create a module, you write your Python code in a file and save it with a .py extension.
- Example: Create a file named mymodule.py with the following content

Importing a Module:
To use the functions and variables defined in a module, you need to import the module into your script.

```
# mymodule.py
def greet(name):
    return f"Hello, {name}!"

def add(a, b):
    return a + b
```

```
import mymodule

print(mymodule.greet("Alice"))
print(mymodule.add(5, 10))
```

Import Variants

Importing Specific Functions : You can import specific functions or variables from a module.

```
from mymodule import greet
print(greet("Alice"))
```

Using Aliases: Modules can be imported with an alias to shorten the name

```
import mymodule as mm
print(mm.greet("Alice"))
```

Chapter 10 : Built-In Modules in Python

Video Links :

https://youtu.be/aZx8U3qlBiQ?si=VAcjeGQes1H4gkHI

Tutorials by Mukesh Kala

Using Built-in Modules

Python comes with a rich set of built-in modules that provide various functionalities out of the box.

math: Provides mathematical functions

random: Generates random numbers and performs random operations

os: Provides functions to interact with the operating system

```
import math

print(math.sqrt(16))  # Output: 4.0
print(math.pi)        # Output: 3.141592653589793
```

```
import os

print(os.getcwd())  # Get the current working directory
os.mkdir('new_folder')  # Create a new directory
```

```
import random

print(random.randint(1, 10))  # Random integer between 1 and 10
print(random.choice(['apple', 'banana', 'cherry']))  # Random choice from a list
```

Assignment

Read | Utilize the Built In Modules | Comment

Chapter 11 : Password Generator Project

Video Links :

https://youtu.be/UTRbC2LJbDk?si=SV8EWwIZMJrZkYe3

Tutorials by Mukesh Kala

Introduction

Why Password Generators?

- A strong password is essential for online security.
- Manual creation of passwords can be time-consuming and prone to errors.
- A password generator automates this process, ensuring randomness and strength.

What You'll Learn in This Project:

- Using Python's built-in modules (random and string).
- String manipulation and loops.
- Writing and testing a function.

Codes

Step 1: Import Necessary Modules

Why Use Modules?

- The string module provides easy access to characters like letters, digits, and punctuation.
- The random module helps in generating randomness.

Step 2: Define the Character Set

- Passwords are typically a mix of letters, numbers, and special characters.
- The string module provides predefined constants: ascii_letters, digits, and punctuation.

Codes

Step 3: Create the Password Generator Function

- A function encapsulates the logic and makes the code reusable.
- The function will:
    - Take the password length as input.
    - Randomly select characters from the predefined set.
    - Return the generated password.

Step 4: Add Input Handling

- The user should be able to specify the desired password length.
- Validate that the password length is at least 6 (a common security recommendation).

Codes

Step 4: Add Input Handling

- The user should be able to specify the desired password length.
- Validate that the password length is at least 6 (a common security recommendation).

```
def password_generator():
    print("Welcome to the Password Generator!")
    try:
        length = int(input("Enter the desired password length: "))
        if length < 6:
            print("Password length should be at least 6 characters.")
        else:
            password = generate_password(length)
            print(f"Your generated password is: {password}")
    except ValueError:
        print("Invalid input. Please enter a number.")
```

Full Code

```python
#import Modules
import string
import random

# Define Character Set
characters = string.ascii_letters + string.digits + string.punctuation

#Create Password Generator Function
def generate_password(length=12):
    password = ""
    for i in range(length):
        password+=random.choice(characters)
    return password

# Add Input Handling
def password_generator():
    print(".......Welcome to the password generator program by TBMK ..........")
    length = int(input("Provide me the Password Length ?"))
    if length <6:
        print("Password length should be atleast 6 Characters ...")
    else:
        password= generate_password(length)
        print(f"Your Password is :  {password}")


# Run the Program
password_generator()
```

Tutorials by Mukesh Kala

Assignment

Add Password Strength Criteria:

- Ensure at least one uppercase letter, one digit, and one special character.

Save Passwords to a File:

- Allow the user to save generated passwords for later use.

Recap

Recap the Key Learnings:

- How to use the random and string modules.
- Writing functions and handling user input.

Assignment:

- Modify the project to add Exception Handling.
- Add Password Strength Criteria: Ensure at least one uppercase letter, one digit, and one special character.
- Save Passwords to a File: Allow the user to save generated passwords for later use.