UNIT – 1

Why do people use python?

People use python for a wide range of purposes due to its versatility, simplicity, and extensive ecosystem of libraries and frameworks. Here are some common reasons why people use python –

1. Ease of learning and readability – Python is known for its clean and readable syntax. Its code is like plain English, making it easier for beginners to learn and write code quickly.
2. Versatility – Python can be used for various types of programming, including web development, data analysis, machine learning, artificial intelligence, scientific computing, automation, and more. Its versatility makes it a go-to language for a wide range of tasks.
3. Large standard library – Python comes with a comprehensive standard library that includes modules and packages for common tasks, which reduces the need to write code from scratch. This library covers everything from file handling to networking and web development.
4. Rich ecosystem – Python has a vast ecosystem of third-party libraries and frameworks. For example, Django and Flask for web development, NumPy and Pandas for data manipulation, TensorFlow and PyTorch for machine learning, and Matplotlib and Seaborn for data visualization.
5. Community and Support – Python has a large and active community of developers who contribute to its growth. This means you can find solution to problems easily through online forums, documentation etc.
6. Cross-Platform Compatibility: Python is available on multiple platforms, including Windows, macOS, and Linux. Code written in Python can be easily ported across different operating systems.
7. Scalability: Python is used by both startups and large organizations due to its scalability. It can handle small scripts and large-scale applications alike.
8. Integration Capabilities: Python can easily integrate with other programming languages, making it a valuable tool for bridging gaps between different technologies and systems.
9. Data Science and Machine Learning: Python is a dominant language in the fields of data science and machine learning, thanks to libraries like SciPy, Scikit-Learn, and Jupyter Notebooks. Its simplicity and powerful data handling capabilities make it a top choice for data analysis.
10. Automation and Scripting: Python is often used for automation tasks and scripting. It can interact with operating systems, files, and databases, making it suitable for automating repetitive tasks.
11. Open Source: Python is an open-source language, which means it is free to use, and the community continually improves and updates it.
12. Education: Python is commonly used as a teaching language in computer science and programming courses due to its simplicity and readability. It helps students grasp fundamental programming concepts without getting bogged down by complex syntax.

USES OF PYTHON

Python is a versatile programming language that finds applications in various domains and industries. Here are some common uses of Python –

1. Web development – Python is used for server-side web development with frameworks like Django, Flask, and Pyramid. It can be used for building websites, web applications, and RESTful APIs.
2. Data analysis and Visualization – Python is popular for data analysis and manipulation using libraries like Pandas, NumPy, and SciPy. Data visualization is made easy with libraries such as Matplotlib, Seaborn, and Plotly.
3. Machine learning and Artificial intelligence – Python is a leading language for machine learning and AI development with libraries like TensorFlow, PyTorch, scikit-learn, and Keras. It is used for tasks like image and speech recognition, natural language processing (NLP), and recommendation systems.
4. Scientific computing – Python is widely used in scientific research and simulations using libraries such as SciPy and SymPy. Its employed in fields like physics, chemistry, biology and engineering.
5. Automation and Scripting – Python is commonly used for scripting and automation tasks on various platforms and operating systems.
6. Game development – Python is used for game development with libraries like Pygame and Panda3D. It is suitable for creating 2D and 3D games.
7. Desktop GUI applications – Python can be used for developing desktop applications with GUIs using libraries like Tkinter, PyQt, and wxPython.
8. Networking – Python is used for network programming, building network applications, and managing network devices.
9. Education – Python is commonly used as a teaching language in universities and schools due to its simplicity and readability.
10. Natural language processing (NLP) – Python is widely used for NLP tasks such as text analysis, sentiment analysis, and language translation.
11. Web Scraping – Python's libraries, like beautiful soup and Scrapy, make it a powerful tool for web scraping and data extraction from websites.
12. Artificial intelligence in healthcare – Python is employed in healthcare for tasks like medical image analysis, drug discovery, and patient data analysis.

NEED OF PYTHON

1. **Versatility**: Python is a versatile language that can be applied to a wide range of tasks, from web development and data analysis to machine learning and scientific computing. Its versatility makes it adaptable to different project requirements.

2. **Ease of Learning**: Python's clean and readable syntax makes it one of the easiest programming languages to learn, which is particularly appealing to beginners and those new to programming.

3. **Large Standard Library**: Python comes with a comprehensive standard library that includes modules and packages for a wide array of functions, reducing the need for developers to write code from scratch. This extensive library accelerates development and simplifies common tasks.

4. **Open Source and Community-Driven**: Python is an open-source language with an active and supportive community. This means that developers worldwide contribute to its development, create third-party libraries, and provide assistance through online forums and documentation.
5. **Cross-Platform Compatibility**: Python is available on multiple platforms, including Windows, macOS, and Linux. This ensures that Python code can be easily run and deployed on various operating systems.
6. **Data Science and Machine Learning**: Python has become the de facto language for data science and machine learning due to libraries such as Pandas, NumPy, SciPy, scikit-learn, TensorFlow, and PyTorch. The need for data analysis and AI/ML capabilities has significantly boosted Python's demand.
7. **Rapid Prototyping**: Python's simplicity and concise syntax allow developers to quickly prototype and test ideas, making it an excellent choice for startups and innovative projects.
8. **Web Development**: Python's web frameworks, such as Django and Flask, simplify web development tasks. They provide tools and conventions that enable developers to build web applications efficiently.
9. **Automation and Scripting**: Python is widely used for scripting and automation tasks. It can interact with operating systems, databases, and external APIs, making it valuable for automating repetitive processes.
10. **Education**: Python is commonly used as an educational language, both in schools and universities, due to its accessibility and ease of understanding. It helps students learn programming concepts without struggling with complex syntax.
11. **Community and Support**: Python's large and active community ensures that developers can find solutions to problems, share knowledge, and stay up-to-date with the latest developments in the language and its ecosystem.
12. **Scalability**: Python is suitable for both small-scale projects and large-scale applications, which makes it a valuable tool for startups and established enterprises.
13. **Embedded Systems and IoT**: Python can be used in embedded systems and IoT projects using platforms like MicroPython and CircuitPython.
14. **Scientific Research**: Python is employed in scientific research for simulations, data analysis, and data visualization, particularly in fields such as physics, biology, and chemistry.
15. **DevOps and System Administration**: Python is used in DevOps for tasks like automation, configuration management, and monitoring, making it a crucial tool for system administrators.

PYTHON’S TECHNICAL STRENGTHS

Python possesses several technical strengths that contribute to its popularity and effectiveness as a programming language. Here are some of its key technical strengths –

1. Readable and expressive syntax – Python’s clean and readable syntax resembles plain English, making it easy to learn and write code. This readability helps developers write code quickly and maintain it more efficiently.
2. Versatility – Python is a versatile language that can be used for a wider range of applications, from web development and data analysis to scientific computing and machine learning.
3. Large standard library – Python comes with a comprehensive standard library that contains modules and packages for various tasks, such as file handling, networking, and web development. This extensive library reduces the need to write code from scratch and accelerates development.
4. Dynamic typing – Python is dynamically types, meaning you don’t need to declare variable types explicitly. This flexibility simplifies code and helps in rapid development.
5. Interpreted language – Python is an interpreted language, which means you can run code without the need for a separate compilation step. This makes it easy to test and experiment with code.
6. Strong community and ecosystem – Python has a vast and active community of developers who contribute to its growth. This community support results in a rich ecosystem of third-party libraries, frameworks, and tools that extends python’s capabilities.
7. Cross-platform compatibility – Python is available on multiple platforms, including windows, macOS, and Linus, ensuring that Python code can be easily deployed on different operating systems.
8. Object-oriented language – Python supports object-oriented programming, allowing developers to create reusable and organized code structures.
9. Memory management – Python’s memory management is automatic and includes garbage collection, which helps developers avoid memory leaks and memory-related bugs.
10. Extensibility – Python can be extended with C/C++ libraries and modules, allowing developers to integrate existing code written in other languages into their python applications.
11. Data science and machine learning – Python has become language for data science and machine learning due to libraries like NumPy, Pandas, scikit-learn, TensorFlow, and PyTorch. These libraries provide powerful tools for data analysis and AI/ML development.
12. Web Development Frameworks: Python offers popular web development frameworks like Django and Flask, which simplify web application development and encourage best practices.
13. Community Support and Documentation: Python's strong community provides extensive documentation, tutorials, and forums, making it easy for developers to find solutions to problems and stay up-to-date with best practices.
14. Security: Python has built-in security features that help protect against common security vulnerabilities, such as buffer overflows.

KEYWORDS IN PYTHON

Keywords in Python are reserved words that have special meanings and purposes within the language. These words cannot be used as identifiers (variable names, function names, etc.) because they are reserved for specific tasks or operations. Python has a set of standard keywords that are part of the language's syntax.

List of python's standard keywords –

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

IDENTIFIERS IN PYTHON

In python, an identifier is a name used to identify a variable, function, class, module, or other objects. Identifiers are case-sensitive, meaning that 'myVariable' and 'myvariable' are considered different identifiers.

Rules for Identifiers:

1. **Character Set**: Identifiers can consist of letters (both uppercase and lowercase), digits, and underscores (_).
2. **Starting with a Letter**: An identifier must start with a letter (a-z, A-Z) or an underscore (_). It cannot start with a digit (0-9).
3. **Case-Sensitivity**: Python is case-sensitive, so identifiers **myVariable** and **myvariable** are treated as different.
4. **Keywords**: Identifiers cannot be the same as Python's reserved keywords (also known as reserved words). You cannot use reserved words as identifiers for your variables or other objects.

INDENTATION IN PYTHON

In python, indentation is a fundamental aspect of the language's syntax. Unlike many other programming languages that use braces or other delimiters to define code blocks, Python uses indentation to indicate the grouping of statements. Indentation is a way of visually representing the structure and hierarchy of your code. Here are the key points to understand about indentation in python –

1. Whitespace indentation – Python uses whitespace, specifically spaces or tabs, to define indentation levels. A consistent level of indentation is essential for python to recognize the structure of your code correctly.
2. Code blocks – Indentation is primarily used to define code blocks. A code block is a group of statements that belong together and are executed together. Common code blocks include those inside functions, loops, conditional statements, and classes.
3. Indentation levels – Indentation levels are typically increased by four spaces or one-tab characters for each level of nesting. The number of spaces or tabs should be consistent throughout the code. Mixing spaces and tabs for indentation can lead to errors.
4. Colon – In python, a colon (:) is used to indicate the beginning of a code block. After a colon, you should indent the following lines to create the code.

VARIABLES IN PYTHON

Variables in python are used to store and manipulate data. They act as symbolic names or references to values stored in memory. In python, you can create variables and assign values to them using a simple assignment statement. Here are some key points about variables in Python –

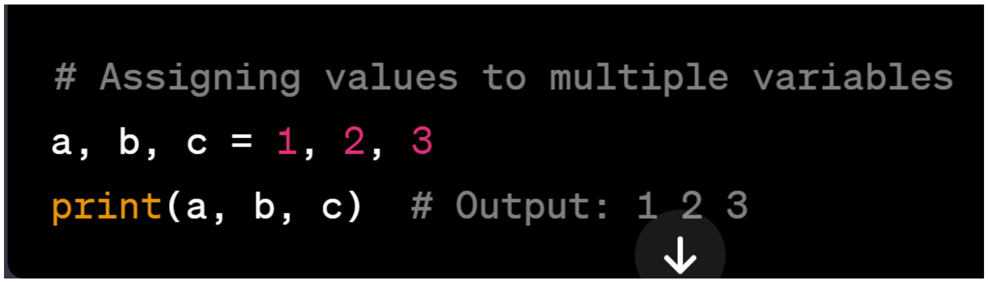
1. **Variable Names**:
    - Variable names must start with a letter (a-z, A-Z) or an underscore **_**. They cannot start with a digit (0-9).
    - Variable names can contain letters, digits, and underscores.
    - Variable names are case-sensitive, meaning **myVar** and **myvar** are considered different variables.
    - Variable names cannot be Python keywords (reserved words).
2. **Assignment**:
    - You can assign values to variables using the **=** operator.
    - The data type of the variable is determined by the value assigned to it.
3. **Dynamic Typing**:
    - Python is dynamically typed, meaning you don't need to specify the data type of a variable when declaring it. Python automatically supposes the data type based on the assigned value.
4. **Data Types**:
    - Python supports various data types, including integers, floating-point numbers, strings, lists, tuples, dictionaries, and more.
    - You can change the value and data type of a variable by assigning a new value of a different data type to it.

MULTIPLE ASSIGNMENT IN PYTHON

Multiple assignment in python refers to the ability to assign multiple values to multiple variables in a single line. This is a convenient and concise way to assign values to variables without repeating the assignment statement.

Here are some few examples of multiple assignment –

Assigning values to multiple variables -



```python
# Assigning values to multiple variables
a, b, c = 1, 2, 3
print(a, b, c)  # Output: 1 2 3
```

DATA TYPES

Python data types are used to define the type of a variable. It defines what type of data we are going to store in a variable. The data stored in memory can be of many types.

For example, A person's age is stored as a numeric value and his or her address is stored as alphanumeric characters.

Python-built-in data types

Numeric – int, float, complex

String – str

Sequence – list, tuple, range

Binary – bytes, bytearray, memoryview

Mapping – dict

Boolean – bool

Set – set, frozenset

None – NoneType

Python numeric data type

Python numeric data type store numeric values. Number objects are created when you assign a value to them. Python supports four different numerical types –

1. Int
2. Long int
3. Float

4. Complex

Python String data type

Python strings are identified as a contiguous set of characters represented in the quotation marks

Python allows for single, double quotes, and triple 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.

Python List data type

Python lists are the most versatile compound data types.

A python list contains items separated by commas and enclosed within square brackets.

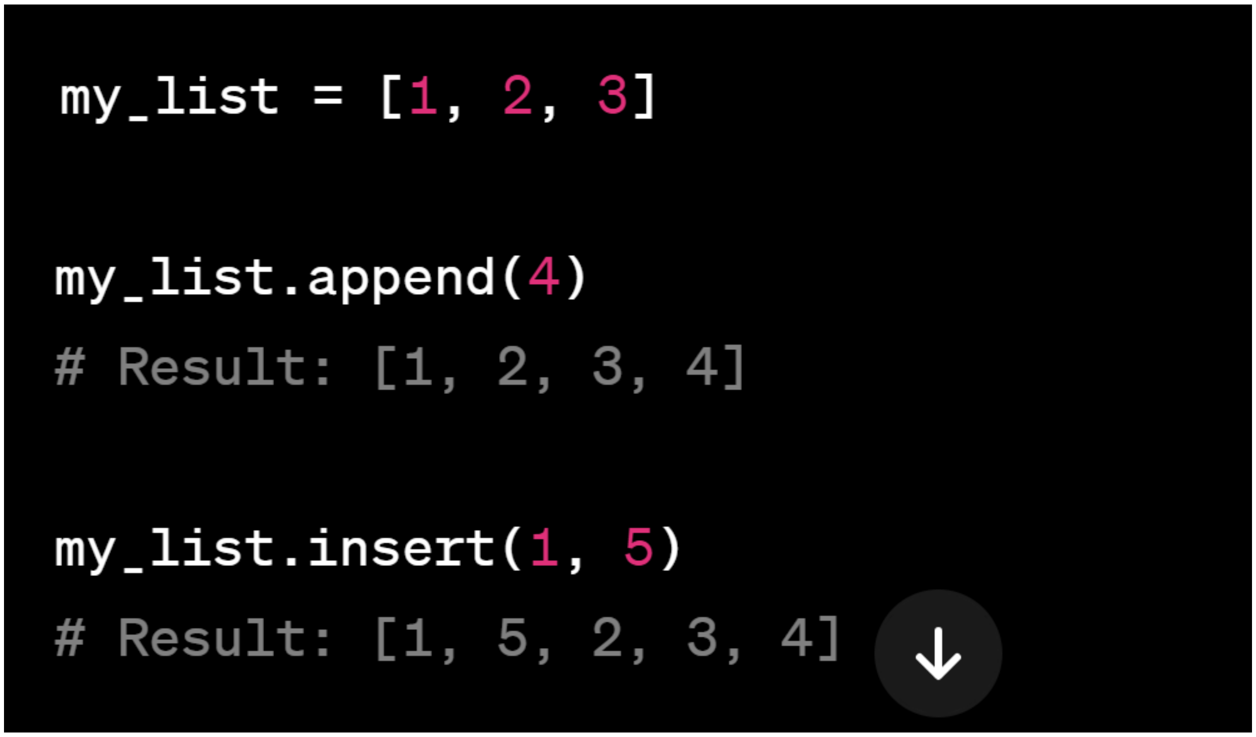
To some extent, Python lists are like arrays in C.

One difference between them is that all the items belonging to a python list can be of different data type whereas in C, array can store elements related to a particular type.

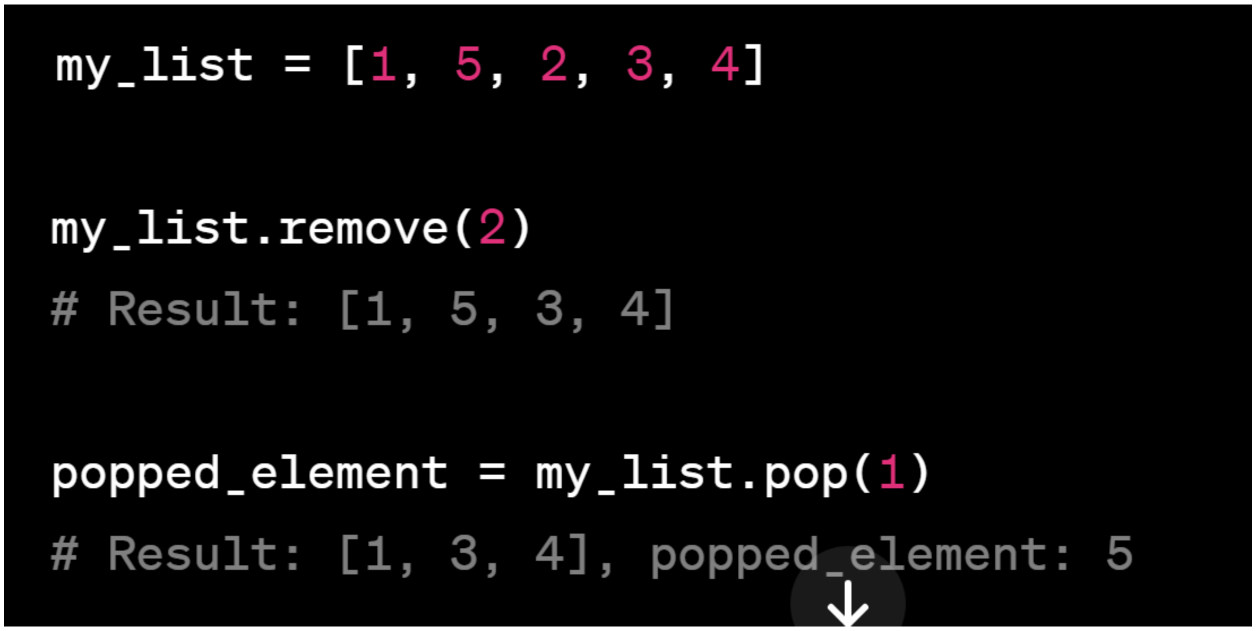
The values stored in a python list can be accessed using the slice operator ([] and [:]) with indexes starting at 0 in the beginning of the list and working their way to end -1.

The plus (+) sign is the list concatenation operator, and the asterisk (*) is the repetition operator.

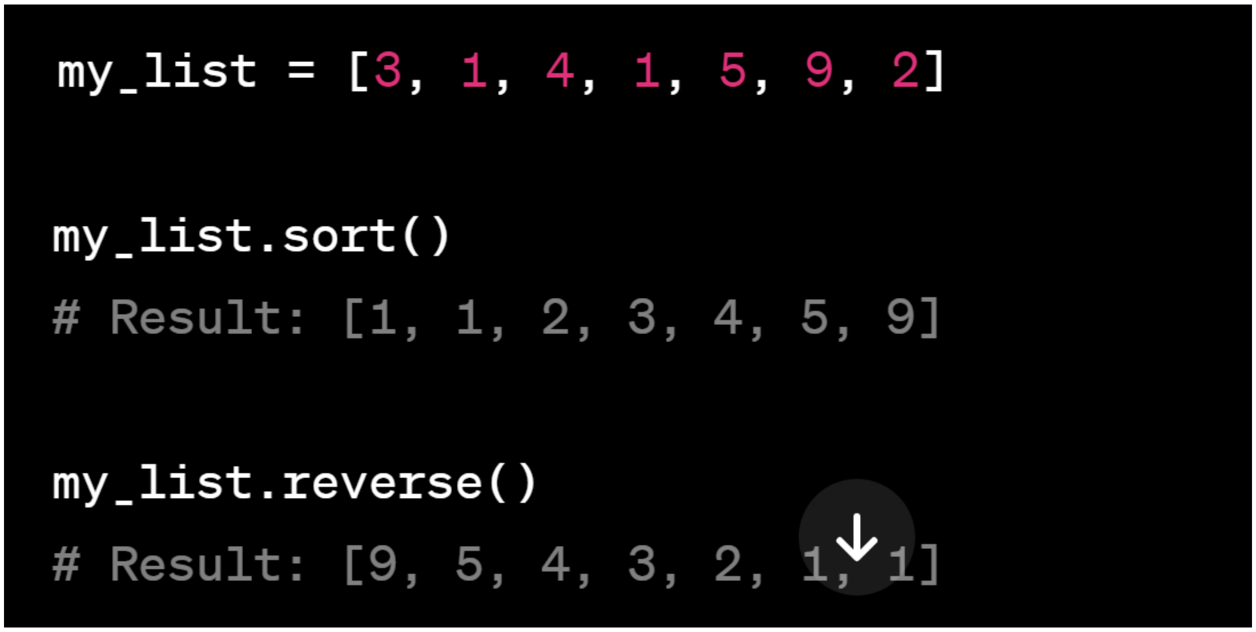
Functions performed on lists –



```
my_list = [1, 2, 3]

my_list.append(4)
# Result: [1, 2, 3, 4]

my_list.insert(1, 5)
# Result: [1, 5, 2, 3, 4]
```



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

my_list.remove(2)
# Result: [1, 5, 3, 4]

popped_element = my_list.pop(1)
# Result: [1, 3, 4], popped_element: 5
```



```python
my_list = [3, 1, 4, 1, 5, 9, 2]

my_list.sort()
# Result: [1, 1, 2, 3, 4, 5, 9]

my_list.reverse()
# Result: [9, 5, 4, 3, 2, 1, 1]
```

Python tuple data type

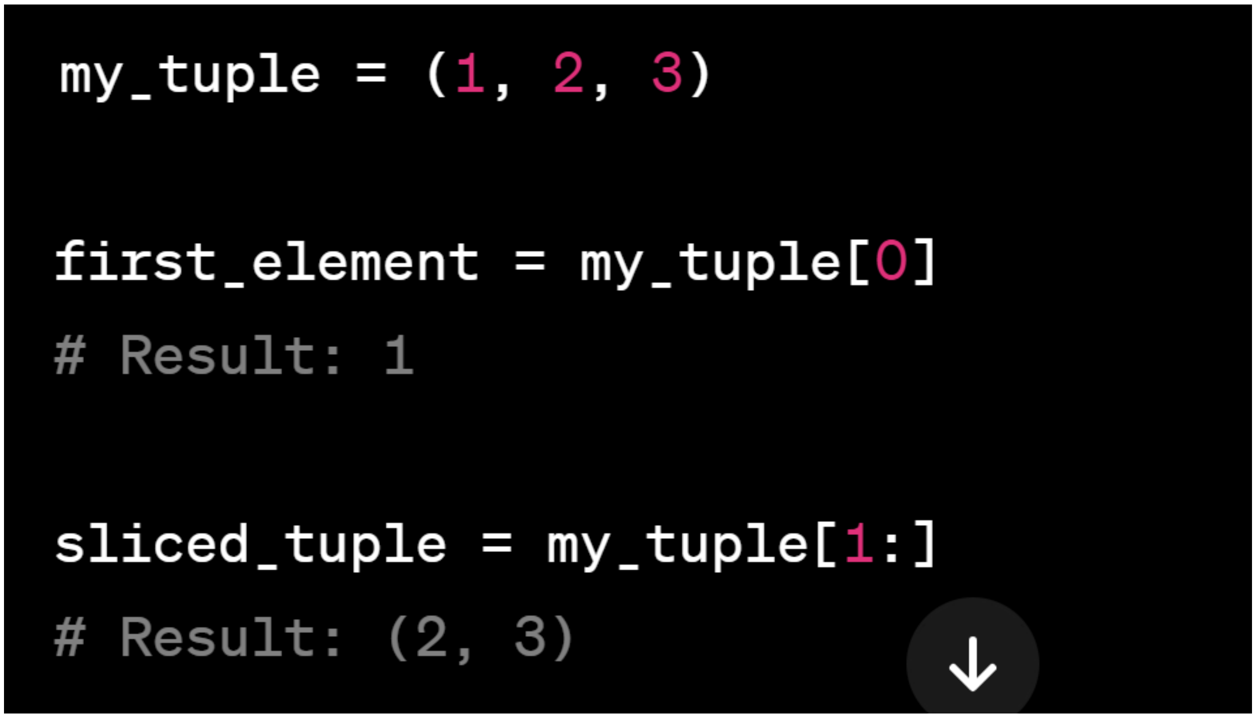
Python tuple is another sequence data type that is like a list.

A python tuple consists of several values separated by commas. Unlike lists, however, tuples are enclosed within parentheses().

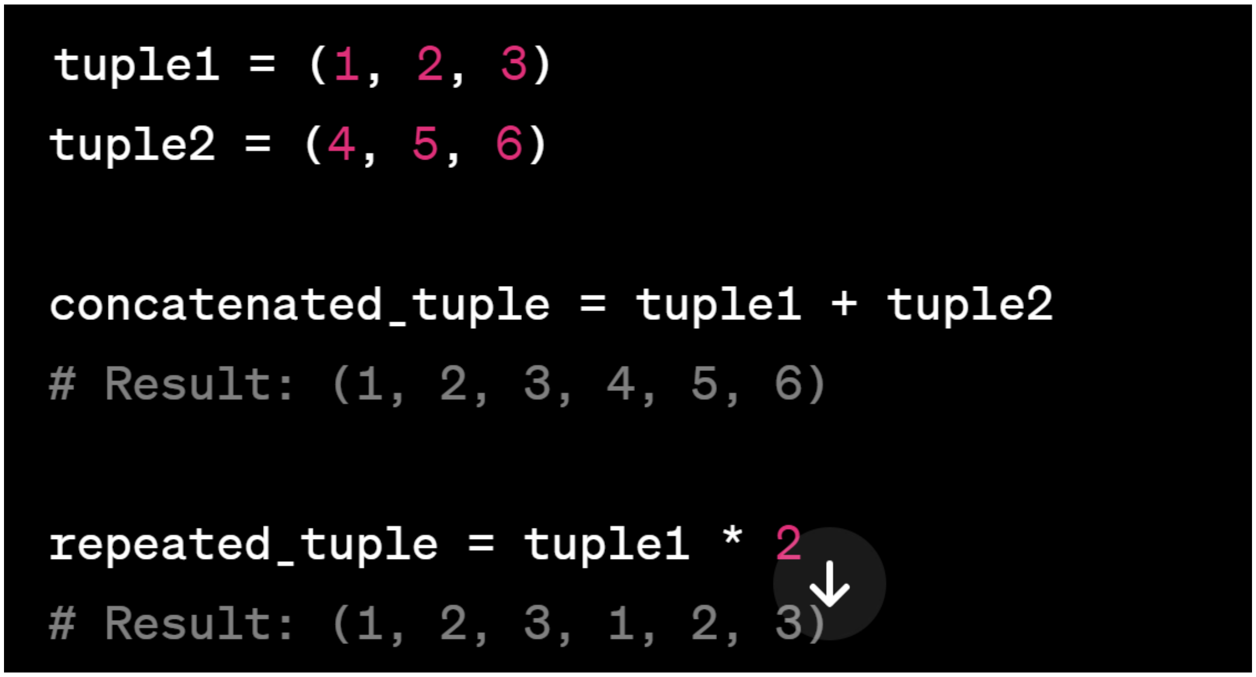
The main difference between lists and tuples are – Lists are enclosed in brackets and their elements and size can be changed (mutable), while tuples are enclosed in parentheses and cannot be updated (immutable).

Tuples can be thought of as read-only lists.

Functions performed on tuples –



```
my_tuple = (1, 2, 3)

first_element = my_tuple[0]
# Result: 1

sliced_tuple = my_tuple[1:]
# Result: (2, 3)
```



```
tuple1 = (1, 2, 3)
tuple2 = (4, 5, 6)

concatenated_tuple = tuple1 + tuple2
# Result: (1, 2, 3, 4, 5, 6)

repeated_tuple = tuple1 * 2
# Result: (1, 2, 3, 1, 2, 3)
```

Python range function

Python range() is an in-built function in Python which returns a sequence of numbers starting from 0 and increments to 1 until it reaches a specified number. We use range() function with for and while loop to generate a sequence of numbers.

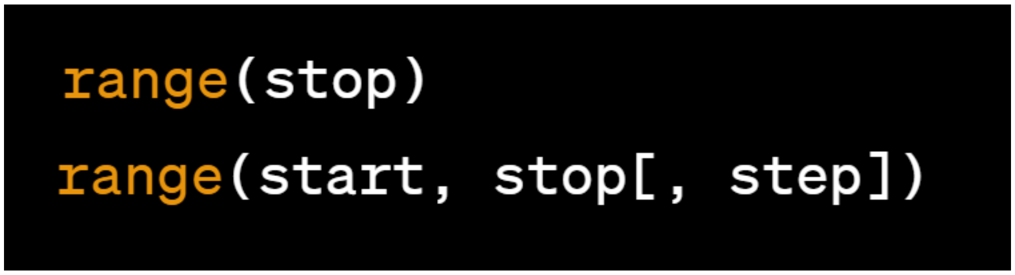
Following is the syntax of the function:

range(start, stop, step)

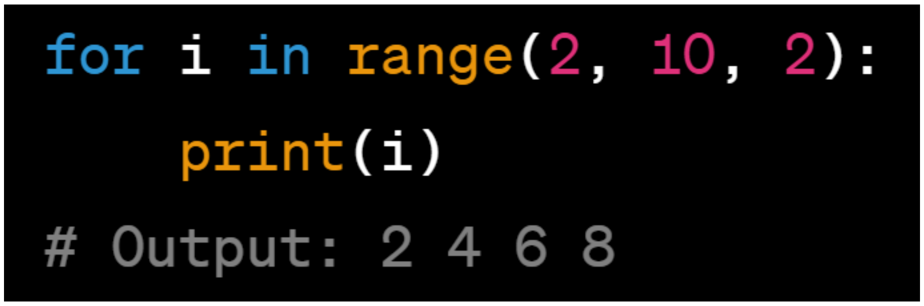
• start: Integer number to specify starting position, (Its optional, Default: 0)

• stop: Integer number to specify starting position (It's mandatory)

• step: Integer number to specify increment, (Its optional, Default: 1)



```
range(stop)
range(start, stop[, step])
```



```
for i in range(2, 10, 2):
    print(i)
# Output: 2 4 6 8
```

Python dictionary

Python dictionaries are kind of hash table type.

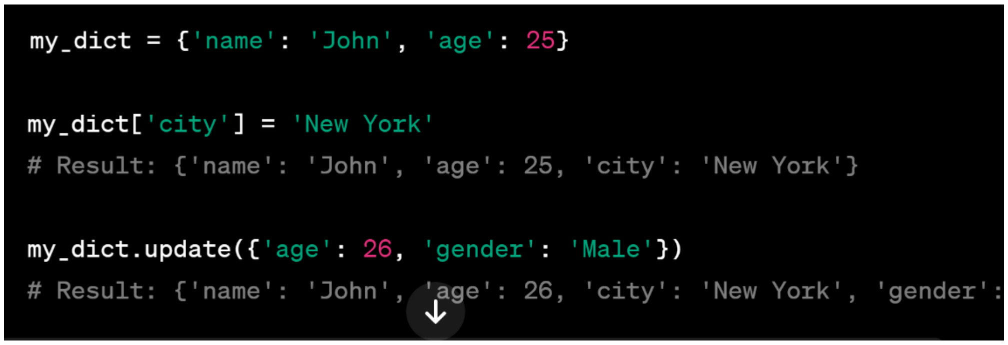
They work like associative arrays or hashes found in Pert and consist of key-value pairs.

A dictionary key can be almost any Python type but are usually numbers or strings.

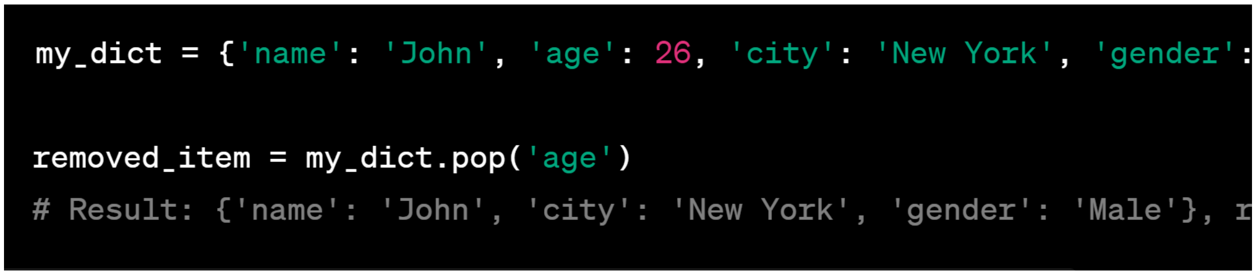
Values, on the other hand, can be any random python object.

Dictionaries are enclosed by curly braces and values can be assigned and accesses using square brackets.

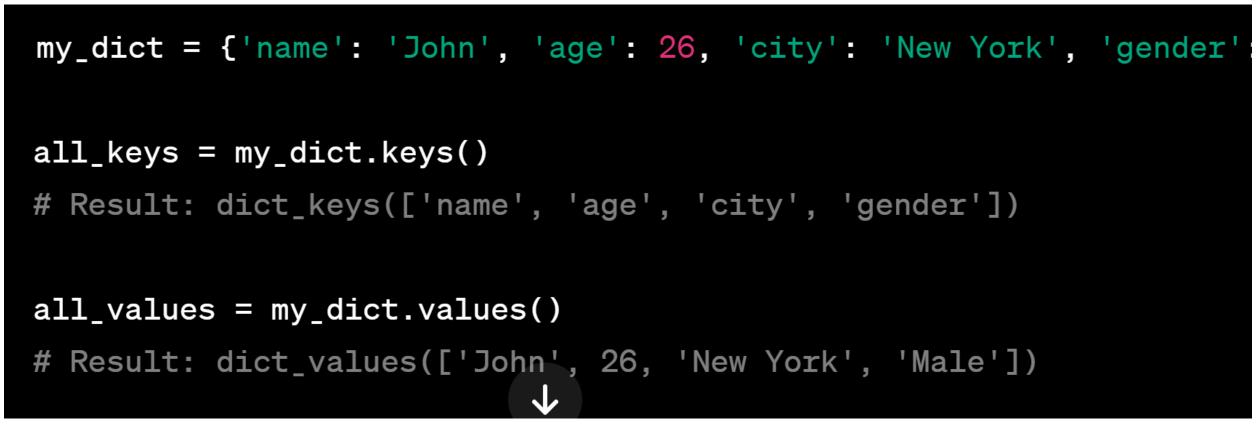
Functions performed on dictionaries –



```
my_dict = {'name': 'John', 'age': 25}

my_dict['city'] = 'New York'
# Result: {'name': 'John', 'age': 25, 'city': 'New York'}

my_dict.update({'age': 26, 'gender': 'Male'})
# Result: {'name': 'John', 'age': 26, 'city': 'New York', 'gender':
```



```
my_dict = {'name': 'John', 'age': 26, 'city': 'New York', 'gender':

removed_item = my_dict.pop('age')
# Result: {'name': 'John', 'city': 'New York', 'gender': 'Male'}, r
```



```
my_dict = {'name': 'John', 'age': 26, 'city': 'New York', 'gender'

all_keys = my_dict.keys()
# Result: dict_keys(['name', 'age', 'city', 'gender'])

all_values = my_dict.values()
# Result: dict_values(['John', 26, 'New York', 'Male'])
```

Python Boolean data type

Python Boolean type is one of built-in data types which represents one of the two values either true or false.

Python bool function allows you to evaluate the value of any expression and returns true or false based on the expression.

OPERATORS IN PYTHON

In Python programming, Operators in general are used to perform operation on values and variables.

These are standard symbols used for the purpose of logical and arithmetic operations.

Operators – These are the special symbols. Example - +, *, /, etc.

Operand – It is the value on which the operator is applied.

Operators in python are symbols or special keywords that are used to perform operations on variables and values. Python supports a wide range of operators, including arithmetic operators, comparison operators, logical operators, assignment operators, bitwise operators, membership operators, and identity operators.

1. Arithmetic Operators:

Arithmetic operators are used to perform mathematical operations on numeric values -

+ (Addition): Adds two operands.

- (Subtraction): Subtracts the right operand from the left operand.

* (Multiplication): Multiplies two operands.

/ (Division): Divides the left operand by the right operand.

// (Floor Division): Divides the left operand by the right operand and rounds down to the nearest whole number.

% (Modulus): Returns the remainder of the division of the left operand by the right operand.

** (Exponentiation): Raises the left operand to the power of the right operand.

2. Comparison Operators:

Comparison operators are used to compare values and return True or False based on the comparison.

== (Equal): Checks if two operands are equal.

!= (Not Equal): Checks if two operands are not equal.

< (Less Than): Checks if the left operand is less than the right operand.

> (Greater Than): Checks if the left operand is greater than the right operand.

<= (Less Than or Equal To): Checks if the left operand is less than or equal to the right operand.

>= (Greater Than or Equal To): Checks if the left operand is greater than or equal to the right operand.

3. Logical Operators:

Logical operators are used to perform logical operations on Boolean values (True or False).

and (Logical AND): Returns True if both operands are True.

or (Logical OR): Returns True if at least one operand is True.

not (Logical NOT): Returns the opposite of the operand's value.

4. Assignment Operators:

Assignment operators are used to assign values to variables.

= (Assignment): Assigns the value of the right operand to the left operand.

+= (Addition Assignment): Adds the right operand to the left operand and assigns the result to the left operand.

-= (Subtraction Assignment): Subtracts the right operand from the left operand and assigns the result to the left operand.

*= (Multiplication Assignment): Multiplies the left operand by the right operand and assigns the result to the left operand.

/= (Division Assignment): Divides the left operand by the right operand and assigns the result to the left operand.

5. Bitwise Operators:

Bitwise operators are used to perform operations on individual bits of integer values. These operators are commonly used in low-level programming and in situations where you need to manipulate binary representations of data.

& (Bitwise AND): Performs a bitwise AND operation.

| (Bitwise OR): Performs a bitwise OR operation.

^ (Bitwise XOR): Performs a bitwise XOR (exclusive OR) operation.

~ (Bitwise NOT): Flips the bits of a number, changing 0s to 1s and vice versa.

<< (Left Shift): Shifts the bits to the left by a specified number of positions.

>> (Right Shift): Shifts the bits to the right by a specified number of positions.

6. Membership Operators:

Membership operators are used to test whether a value is a member of a sequence (such as a string, list, or tuple).

in: Returns True if the specified value is found in the sequence.

not in: Returns True if the specified value is not found in the sequence.

7. Identity Operators:

Identity operators are used to compare the memory addresses of two objects to determine if they are the same object.

is: Returns True if the two operands refer to the same object in memory.

is not: Returns True if the two operands do not refer to the same object in memory.

UNIT – 2

DECISION MAKING STATEMENTS

In python, decision-making statements are used to control the flow of a program based on certain conditions. The primary decision-making statements in python are –

If statement

The if statement is used to execute a block of code if a specified condition is true. It has the following syntax –

```
If condition:

        # code to execute if the condition is true
```

Example –

```
X = 10

If x > 5:

        Print(“x is greater than 5”)
```

If else statement

The if else statement allows you to execute one block of code if a condition is true and another block if its false. It has the following syntax –

```
If condition:

        # code to execute if condition is true

Else:

        # code to execute if condition is false
```

Example –

```
X = 3

If x % 2 == 0:

        Print(“x is even”)

Else:

        Print(“x is odd”)
```

if-elif-else Statement:

The if-elif-else statement is used when you have multiple conditions to check, and it allows you to execute different blocks of code based on which condition is true. It has the following syntax:

```
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 is true
```

Example –

```
x = 5


if x > 10:

   print("x is greater than 10")

elif x > 5:

   print("x is greater than 5 but not greater than 10")

else:

   print("x is less than or equal to 5")
```

Nested if statement

You can nest if statements within other if statements to handle more complex decision-making scenarios. This allows you to check multiple conditions within different levels of indentation. Here's an example:

```
x = 10

y = 5


if x > 5:

   if y > 3:

      print("Both x and y are greater than 5 and 3")

   else:

      print("x is greater than 5, but y is not greater than 3")

else:

   print("x is not greater than 5")
```

PYTHON LOOPS

Loops are used in programming to repeat a specific block of code.

Python has two primitive loop commands –

1. While loop
2. For loop

For loop in python

A for loop is used when you know in advance that how many times you want to execute a block of code. It iterates over a sequence (such as a list, tuple or string) or an iterable (such as range) and executes the code for each item in the sequence. Here is the basic syntax –

```
For variable in sequence:

        # code to execute for each item in the sequence
```

Example 1 : Looping through a list

```
Fruits = [“apple”, “banana”, “cherry”]

For fruit in fruits:

        Print(fruit)
```

Example 2 : Looping with the range() function

```
For i in range (1,5):

        Print(i)
```

While loop in python

A while loop is used when you want to repeatedly execute a block of code as long as a specified condition is true. It continues to execute the code until the condition becomes false. Here is the basic syntax –

```
While condition :

        # code to execute as long as the condition is true
```

Example : countdown with a while loop

```
Countdown = 5

While countdown > 0:

        Print(Countdown)

        Countdown = Countdown – 1
```

Loop control statements

Python also provides loop control statements that allow you to modify the behavior of loops –

1. Break – the break statement is used to exit a loop prematurely. It is often used when a certain condition is met, and you want to stop the loop.
   Example –
   ```
   For I in range(1, 10):
       If I == 5:
           Break
       Print(i)
   ```
2. Continue – the continue statement is used to skip the current iteration for a loop and move to the next iteration.
   Example –
   ```
   For I in range (1, 6):
       If I == 3:
           Continue
       Print(i)
   ```
3. Else clause in loops – Python allows you to use an else clause in for and while loops. The code in the else block is executed after the loop completes normally (without entering a break statement).
   Example –
   ```
   For I in range (1, 5):
       Print(i)
   Else:
       Print("Loop finished without a break")
   ```

   These loop control statements add flexibility to your loops and allow you to handle various situations within your code.

STRINGS IN PYTHON

Strings are a fundamental data type in Python that represent sequences of characters. They are used to store and manipulate textual data. Here are some basics of working with strings in Python –

Creating Strings:

Strings in Python can be created using single quotes (**'**), double quotes (**"**), or triple quotes (**'''** or **"""**). The choice of quotation style depends on your preference and the specific requirements of your string.

Examples –

```
single_quoted = 'This is a single-quoted string.'

double_quoted = "This is a double-quoted string."

triple_quoted = '''This is a triple-quoted string.'''
```

Accessing Characters:

You can access individual characters within a string using indexing. Python uses zero-based indexing, meaning the first character is at index 0, the second at index 1, and so on.

Examples –

```
my_string = "Hello, World!"

first_char = my_string[0]  # 'H'
```

String Length:

To find the length (number of characters) of a string, you can use the **len()** function.

Example –

```
my_string = "Hello, World!"

length = len(my_string)  # 13
```

String Concatenation:

You can concatenate (combine) strings using the **+** operator.

Example –

```
first_name = "John"

last_name = "Doe"

full_name = first_name + " " + last_name  # "John Doe"
```

String Methods:

Python provides numerous built-in methods for manipulating strings. Here are some common string methods:

- **str.upper()**: Converts all characters in the string to uppercase.
- **str.lower()**: Converts all characters in the string to lowercase.
- **str.strip()**: Removes leading and trailing whitespace from the string.
- **str.replace(old, new)**: Replaces all occurrences of **old** with **new** in the string.
- **str.split(separator)**: Splits the string into a list of substrings based on the **separator**.
- **St**r.capitalize()
- Str.casefold()
- Str.center()
- Str.count()
- Str.endswith()
- Str.index()
- Str.alphanum()
- Str.numeric()
- Str.replace()
- Str.startswith()

Example –

```
text = "   Python Programming   "

text = text.strip()  # Removes leading and trailing whitespace

text = text.lower()  # Converts to lowercase
```

String Formatting:

Python supports various ways of formatting strings, including:

- F-strings (formatted string literals)
- String interpolation using **.format()**
- String concatenation

Example –

```
name = "Alice"
```

age = 30

formatted_string = f"My name is {name} and I am {age} years old."

String Indexing and Slicing:

You can use indexing to access individual characters and slicing to extract substrings from a string.

Example –

my_string = "Python is fun!"

first_char = my_string[0] # 'P'

substring = my_string[7:9] # 'is'

Escape Characters:

Python uses escape characters, like **\n** for a newline and **\"** for a double quote within a string.

Example –

escaped_string = "This is a line\nThis is another line"

Strings are versatile and widely used in Python for a variety of tasks, including text processing, data manipulation, and more. Understanding these string basics is essential for working with textual data effectively in Python.

MODULES IN PYTHON

In python, a module is a file containing Python definitions and statements. The file name is the module name with suffix '.py' added. Modules allow you to organize your python code logically and reuse it in other programs.

Here's a brief overview of working with modules in Python –

1. Creating a module – To create a module, you simple write your python code in a file with a '.py' extension. For example, if you have a file named 'my_module.py', it can be considered a module.
2. Importing a module – You can use the import statement to include the definitions from a module into your current script or another module.
   Example – import my_module
3. Accessing module elements – Once a module is imported, you can access its functions, variables or classes using dot notation.
   Example – my_module.my_function()
4. Alias for a module – You can use an alias to refer to a module with a different name in your code.
   ```
   Example – import my_module as mm
                mm.my_function()
   ```
5. Importing specific elements – You can import only specific elements from a module using the 'from …. Import …..' statement.
   ```
   Example – from my_module import my_function
             My_function()
   ```
6. Built-in modules – Python comes with a set of built-in modules that provides additional functionality. You can import and use these modules in your programs.
   ```
   Example – import math
             Print(math.sqrt(16))
   ```
7. Creating a standalone script – You can include the following block in your module to create a standalone script that can be executed directly.
   ```
   Example – if __name__ == "__main__":
                # Code to run when the script is executed.
   ```
   This allows you to write code that can be both a reusable module and a standalone script.

   Here's a simple example demonstrating these concepts –

   ```
   #my_module.py
   Def my_function():
           Print("HELLO FROM my_function in my_module!")

   #main_script.py
   Import my_module

   My_module.my_function()

   #Output –HELLO FROM my_function is my_module!
   ```

MATH MODULE

The **math** module in Python provides mathematical functions for various operations. Here are some commonly used functions from the **math** module:

1. **Basic Mathematical Functions:**
    - **math.sqrt(x)**: Returns the square root of x.
    - **math.exp(x)**: Returns the exponential of x.
    - **math.log(x)**: Returns the natural logarithm of x.
    - **math.log10(x)**: Returns the base-10 logarithm of x.
    - **math.pow(x, y)**: Returns x raised to the power y.
    - **math.ceil(x)**: Returns the ceiling value of x (the smallest integer greater than or equal to x).
    - **math.floor(x)**: Returns the floor value of x (the largest integer less than or equal to x).
    - **math.trunc(x)**: Returns the truncated integer value of x.
2. **Trigonometric Functions:**
    - **math.sin(x)**, **math.cos(x)**, **math.tan(x)**: Return the sine, cosine, and tangent of x, respectively.
    - **math.asin(x)**, **math.acos(x)**, **math.atan(x)**: Return the arcsine, arccosine, and arctangent of x, respectively.
3. **Angular Conversion:**
    - **math.degrees(x)**: Converts angle x from radians to degrees.
    - **math.radians(x)**: Converts angle x from degrees to radians.
4. **Constants:**
    - **math.pi**: A constant representing the mathematical constant pi (approximately 3.14159).
    - **math.e**: A constant representing the mathematical constant e (approximately 2.71828).

Here's a simple example demonstrating the use of the **math** module:

```
import math

# Basic mathematical functions

print(math.sqrt(25)) # 5.0

print(math.pow(2, 3)) # 8.0

print(math.ceil(4.3)) # 5
print(math.floor(4.9)) # 4

 # Trigonometric functions

print(math.sin(math.radians(30))) # 0.49999999999999994 (approximately 0.5)

# Constants

print(math.pi) # 3.141592653589793

print(math.e) # 2.718281828459045
```

These are just a few examples of what the **math** module provides. The module is quite extensive and covers a wide range of mathematical operations.

RANDOM MODULE

The **random** module in Python provides functions for generating pseudorandom numbers. Pseudorandom numbers are not truly random, but they are generated in a way that simulates randomness. Here are some commonly used functions from the **random** module:

1. **Generating Random Numbers:**
    - **random.random()**: Returns a random floating-point number in the range [0.0, 1.0).

```
import random

random_number = random.random()

print(random_number)
```

2. **Generating Random Integers:**
    - **random.randint(a, b)**: Returns a random integer in the range [a, b], including both endpoints.

```
import random

random_integer = random.randint(1, 10)

print(random_integer)
```

3. **Choosing Random Items:**
    - **random.choice(seq)**: Returns a randomly selected element from the given sequence (**list**, **tuple**, or **string**).

```
import random

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

random_item = random.choice(my_list)

print(random_item)
```

4. **Shuffling a Sequence:**
    - **random.shuffle(seq)**: Randomly shuffles the elements of the given sequence in place.

```
import random

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

random.shuffle(my_list)

print(my_list)
```

5. **Generating Random Floating-Point Numbers within a Range:**
    - **random.uniform(a, b)**: Returns a random floating-point number in the range [a, b).

```
import random

random_float = random.uniform(1.0, 5.0)

print(random_float)
```

6. **Seed for Reproducibility:**
    - **random.seed(seed)**: Sets the seed for the random number generator. This is useful for reproducibility, as the same seed will produce the same sequence of random numbers.

```
import random

random.seed(42)

random_number = random.random()

print(random_number)
```

The **random** module is often used in scenarios where randomness is needed, such as simulations, games, and statistical applications. Keep in mind that the numbers generated by the **random** module are pseudorandom and are determined by an initial seed value. If you need truly random numbers, you may want to explore the **secrets** module for cryptographic purposes.

COMPOSITION AND DISTRIBUTION UTILITY

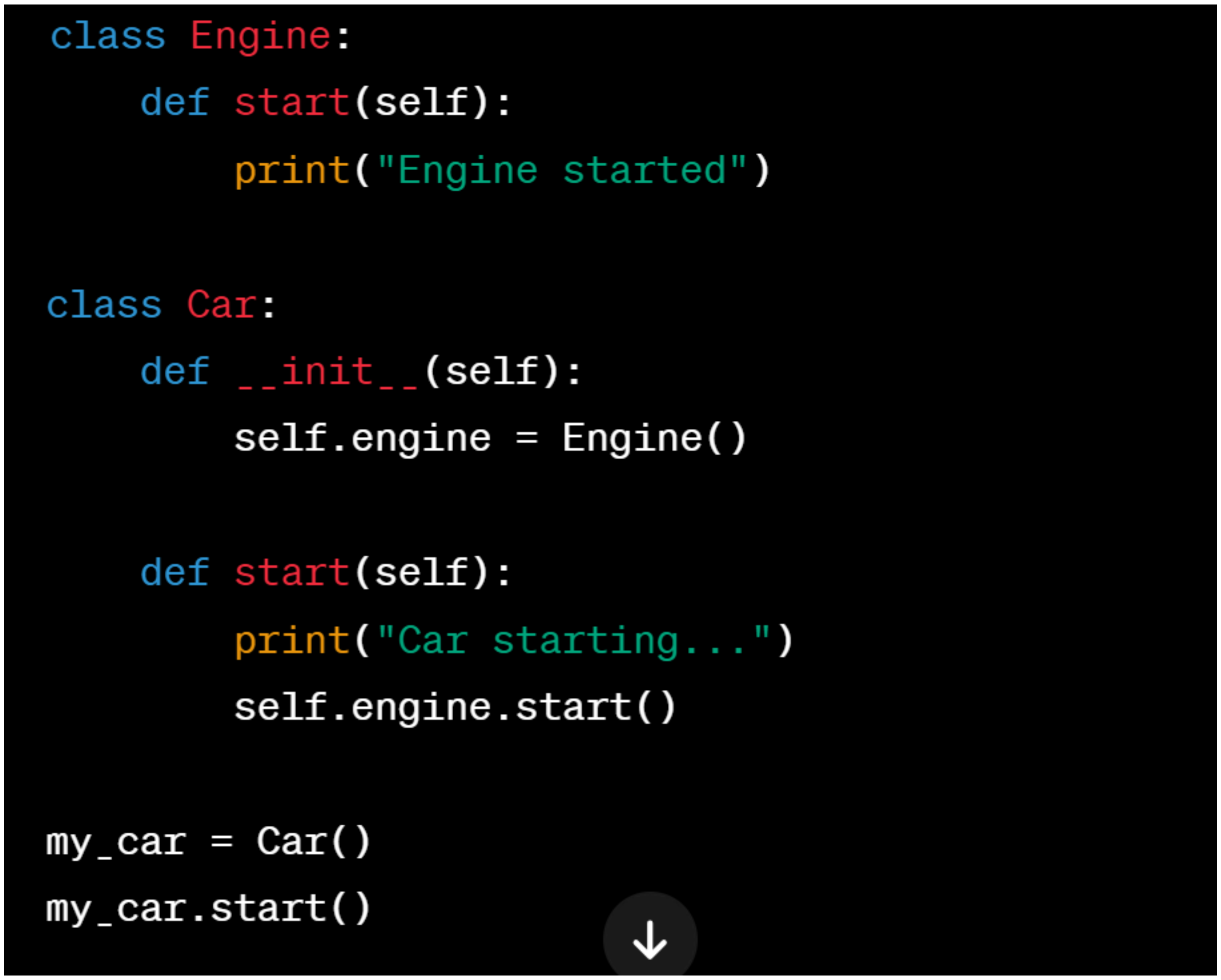
Composition:

Composition is a concept in object-oriented programming (OOP) where a class can be composed of other classes as parts, rather than inheriting from them. It promotes code reuse and a more modular approach to building software.

1. **Composition vs. Inheritance:**
    - Inheritance involves creating a new class that inherits properties and behaviors from an existing class.

- Composition involves creating instances of other classes within your class and using them to implement your class's functionality.

2. **Example of Composition:**



```python
class Engine:
    def start(self):
        print("Engine started")

class Car:
    def __init__(self):
        self.engine = Engine()

    def start(self):
        print("Car starting...")
        self.engine.start()

my_car = Car()
my_car.start()
```

In this example, the **Car** class has a composition relationship with the **Engine** class. An instance of **Engine** is created inside **Car**, and **Car** delegates the **start** functionality to its **Engine**.

Distribution Utility:

If by "The Distribution Utility" you are referring to a specific Python library, tool, or concept introduced after January 2022, I might not be aware of it. Please provide more context or check the latest documentation for any updates.

If "distribution utility" refers to a more general concept, it might involve tools or practices related to packaging, distributing, and deploying Python applications. Common tools for managing Python package distribution include **pip**, **setuptools**, and **virtualenv** or **venv** for creating isolated environments.

UNIT - 3

FUNCTIONS

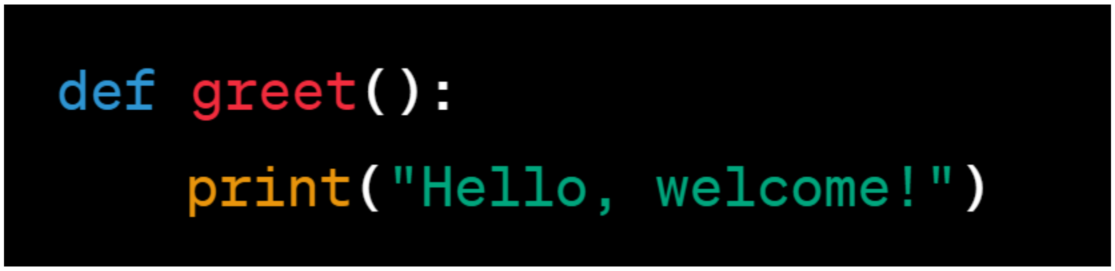
A function is a block of code which only runs when it is called.

You can pass data, known as parameters, into a function.

A function can return data as a result.

1. Defining a Function:

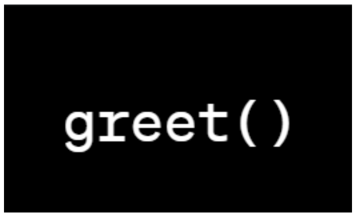
- In Python, you define a function using the **def** keyword followed by the function name and parentheses.
- Example:



```python
def greet():
    print("Hello, welcome!")
```

2. Calling a Function:

- To call a function, you use its name followed by parentheses.
- Example:

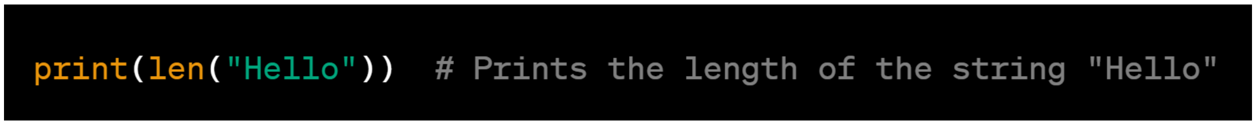


```python
greet()
```

# Calls the greet function and prints "Hello, welcome!"

3. Types of Functions:

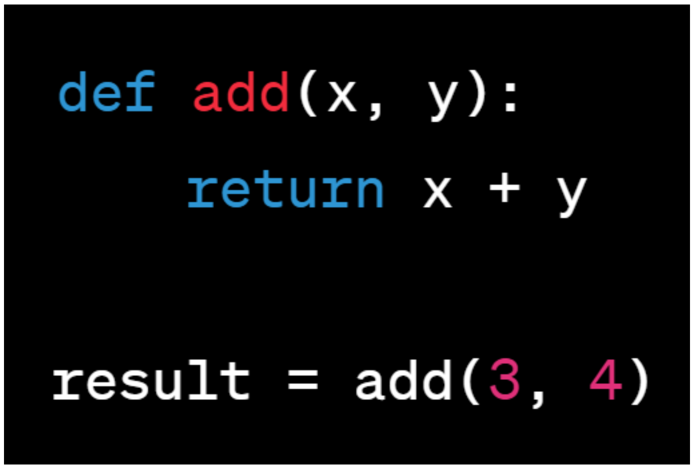
- **Built-in Functions:**
  - These are functions that are part of the Python language, such as **print()**, **len()**, etc.



```python
print(len("Hello"))  # Prints the length of the string "Hello"
```

User-Defined Functions:

- These are functions that you define in your code using the **def** keyword.

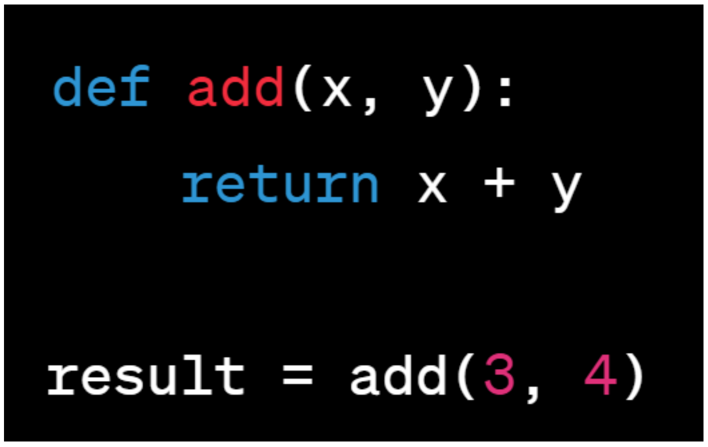


```python
def add(x, y):
    return x + y

result = add(3, 4)
```

# Calls the add function and assigns the result to the variable "result"

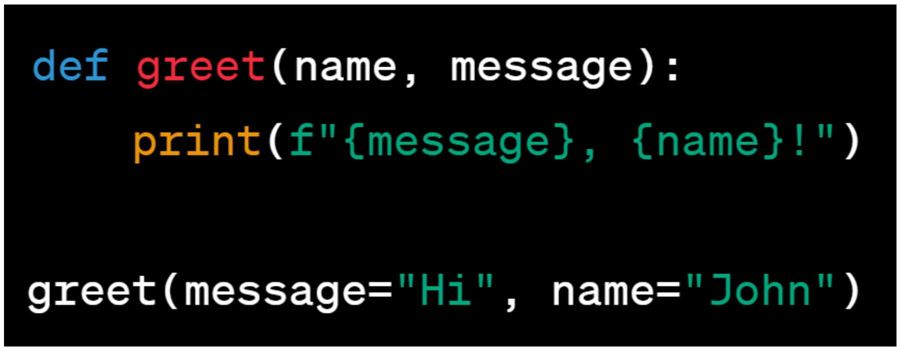
4. Function Arguments:

- **Positional Arguments:**
  - The most common type of arguments. The values are assigned based on their position.



```python
def add(x, y):
    return x + y

result = add(3, 4)
```

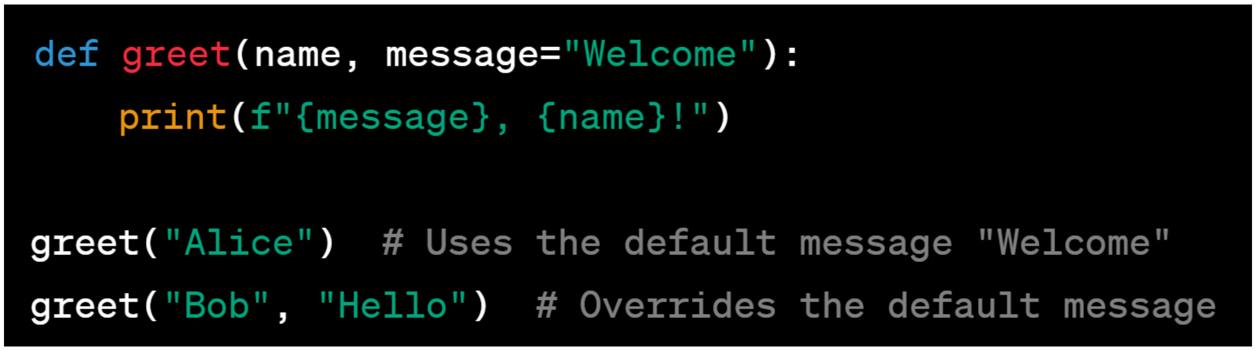
- **Keyword Arguments:**
  - Values are assigned using the parameter names.



```python
def greet(name, message):
    print(f"{message}, {name}!")

greet(message="Hi", name="John")
```

# Order doesn't matter because of keyword arguments

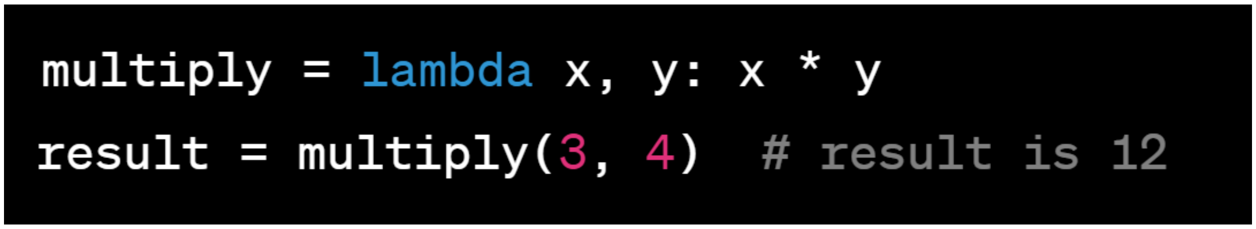
- **Default Values:**
    - You can provide default values for parameters.



```python
def greet(name, message="Welcome"):
    print(f"{message}, {name}!")

greet("Alice")  # Uses the default message "Welcome"
greet("Bob", "Hello")  # Overrides the default message
```

5. Anonymous Functions (Lambda Functions):

- A lambda function is a small, anonymous function defined with the **lambda** keyword.
- Example:



```python
multiply = lambda x, y: x * y
result = multiply(3, 4)  # result is 12
```

- Lambda functions are often used for short-term operations where a full function definition is not necessary.

These are the basics of defining, calling, and using different types of functions in Python. Functions are essential for structuring your code, promoting reusability, and making your code more modular.

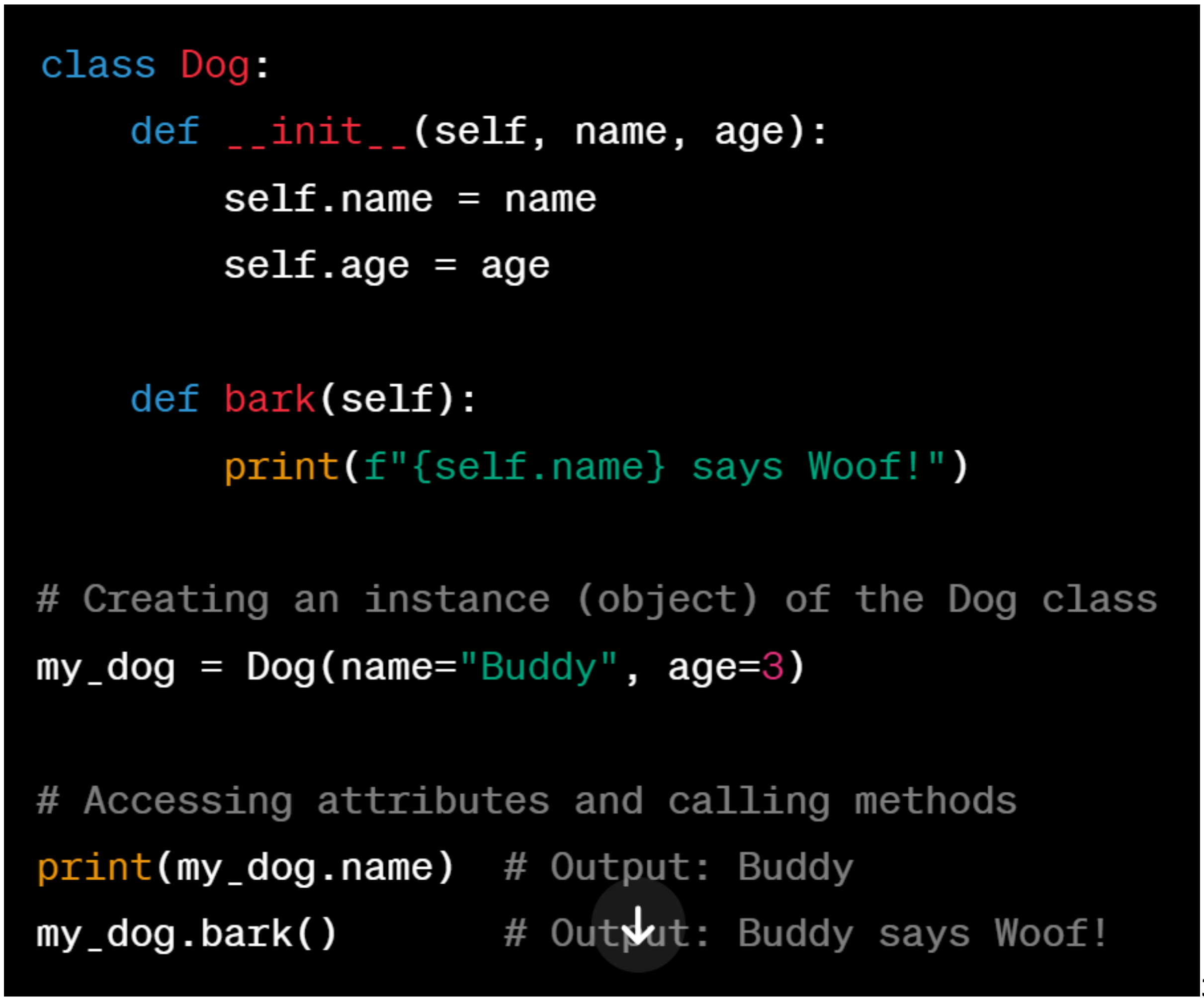
CLASSES AND OBJECTS

1. Introduction to Object-Oriented Programming (OOP):

- **Object-Oriented Programming (OOP)** is a programming paradigm that uses objects and classes to organize code. It revolves around the concept of objects, which encapsulate data and behaviors.

2. Class Coding Basics:

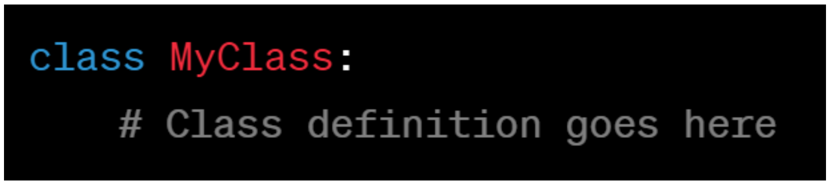
- **Class:** A class is a blueprint for creating objects. It defines a set of attributes (data) and methods (functions) that operates on those attributes.
- **Object:** An object is an instance of a class. It represents a concrete occurrence based on the class definition.
- **Example:**



```python
class Dog:
    def __init__(self, name, age):
        self.name = name
        self.age = age

    def bark(self):
        print(f"{self.name} says Woof!")

# Creating an instance (object) of the Dog class
my_dog = Dog(name="Buddy", age=3)

# Accessing attributes and calling methods
print(my_dog.name)  # Output: Buddy
my_dog.bark()       # Output: Buddy says Woof!
```

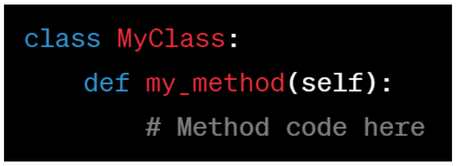
3. Class Coding Details:

- **Class Statement:**
  - The **class** keyword is used to define a class.



```python
class MyClass:
    # Class definition goes here
```

- **Methods:**
  - Methods are functions defined within a class. They operate on the attributes of the class.



```python
class MyClass:
    def my_method(self):
        # Method code here
```

- **Attributes:**
    - Attributes are variables that store data for the class.

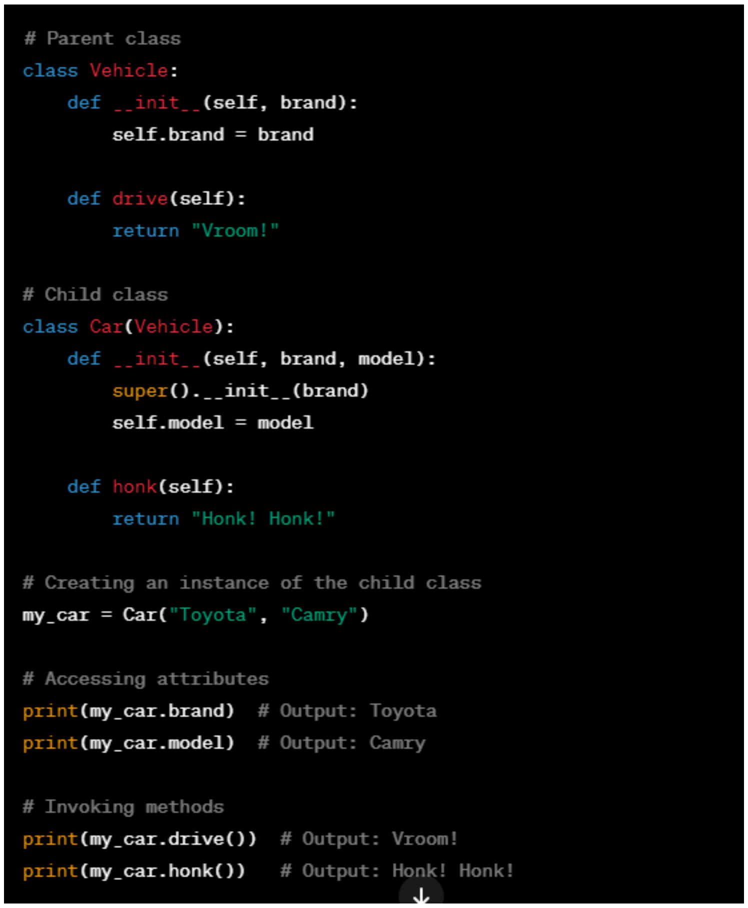
```python
class MyClass:
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
```

- **Constructor (__init__):**
    - The **__init__** method initializes the object's attributes when an object is created.

```python
class MyClass:
    def __init__(self, attribute1, attribute2):
        self.attribute1 = attribute1
        self.attribute2 = attribute2
```

4. Inheritance:

- **Inheritance** allows a new class (subclass or derived class) to inherit the attributes and methods of an existing class (base class or superclass).
- Example:



```python
# Parent class
class Vehicle:
    def __init__(self, brand):
        self.brand = brand

    def drive(self):
        return "Vroom!"

# Child class
class Car(Vehicle):
    def __init__(self, brand, model):
        super().__init__(brand)
        self.model = model

    def honk(self):
        return "Honk! Honk!"

# Creating an instance of the child class
my_car = Car("Toyota", "Camry")

# Accessing attributes
print(my_car.brand)  # Output: Toyota
print(my_car.model)  # Output: Camry

# Invoking methods
print(my_car.drive())  # Output: Vroom!
print(my_car.honk())   # Output: Honk! Honk!
```

- In this example, **Dog** and **Cat** inherit from the **Animal** class. They override the **speak** method to provide their own implementation.

OOP provides a way to structure code, improve code organization, and promote code reuse. Understanding classes, objects, methods, and inheritance is fundamental to effectively using OOP in Python.

DESIGNING WITH CLASSES

Designing with classes in Python and leveraging object-oriented programming (OOP) principles involves structuring your code in a way that reflects the relationships and behaviors of the real-world entities your program models. Let's explore how to design with classes and specifically look at OOP inheritance.

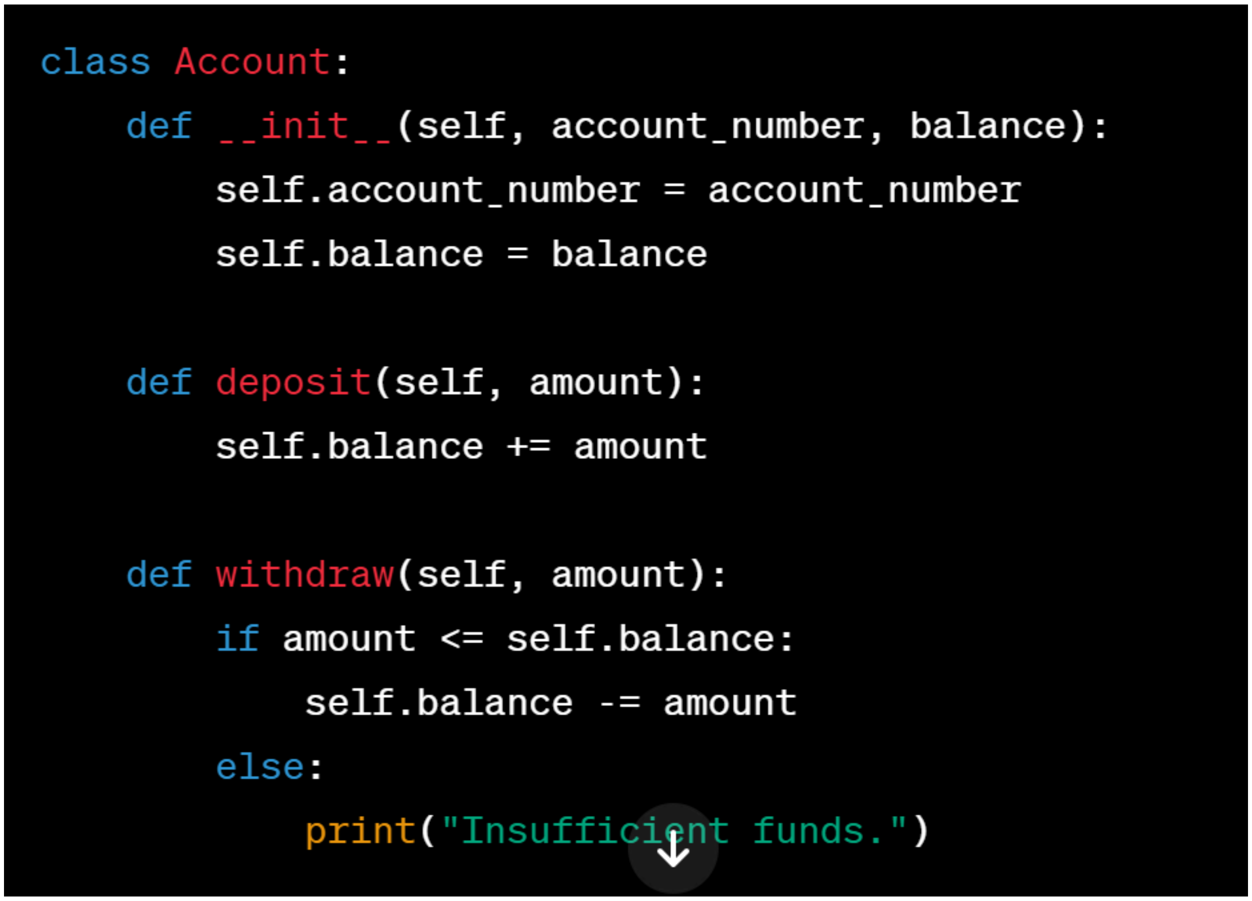
Designing with Classes in Python:

1. **Identify Objects:**

- Identify the key entities in your problem domain that can be represented as objects.
- Example: In a banking system, objects could include **Account**, **Customer**, and **Transaction**.

2. **Define Classes:**

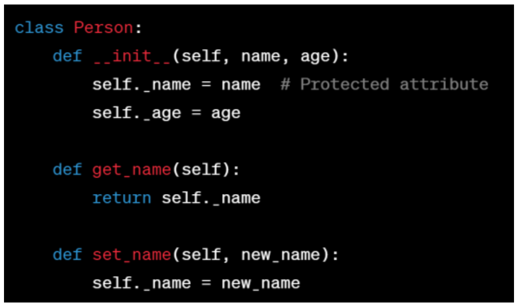
- Define classes based on the identified objects. Include attributes (data) and methods (functions).
- Example:



```python
class Account:
    def __init__(self, account_number, balance):
        self.account_number = account_number
        self.balance = balance

    def deposit(self, amount):
        self.balance += amount

    def withdraw(self, amount):
        if amount <= self.balance:
            self.balance -= amount
        else:
            print("Insufficient funds.")
```

3. **Encapsulation and Information Hiding:**

- Encapsulate data within classes, and control access to it. Use access modifiers for information hiding.
- Example:



```python
class Person:
    def __init__(self, name, age):
        self._name = name  # Protected attribute
        self._age = age

    def get_name(self):
        return self._name

    def set_name(self, new_name):
        self._name = new_name
```

4. **Composition:**

- Use composition to create more complex objects by combining simpler ones.
- Example:

```python
class Address:
    def __init__(self, street, city, zip_code):
        self.street = street
        self.city = city
        self.zip_code = zip_code

class Person:
    def __init__(self, name, age, address):
        self.name = name
        self.age = age
        self.address = address
```

OOP Inheritance:

1. **Identify Commonalities:**

- Identify common attributes and behaviors among classes. Commonalities suggest the possibility of using inheritance.

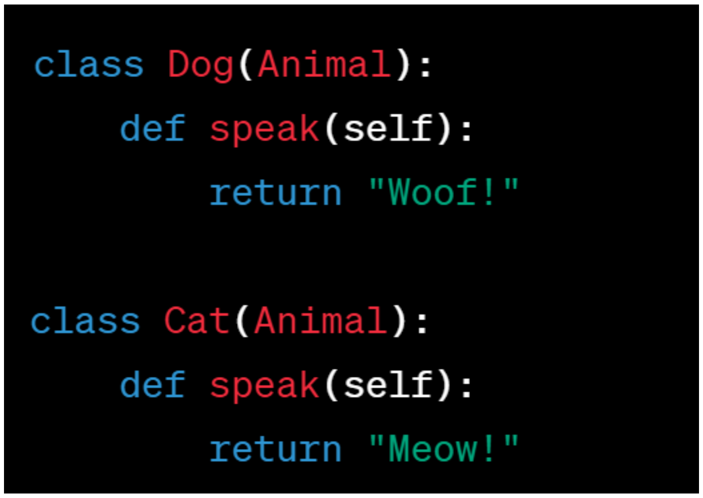
2. **Base (Super) Class:**

- Create a base class (superclass) with shared attributes and methods.
- Example:

```python
class Animal:
    def __init__(self, name):
        self.name = name

    def speak(self):
        pass  # Abstract method, to be overridden by subclasses
```

3. **Derived (Sub) Classes:**

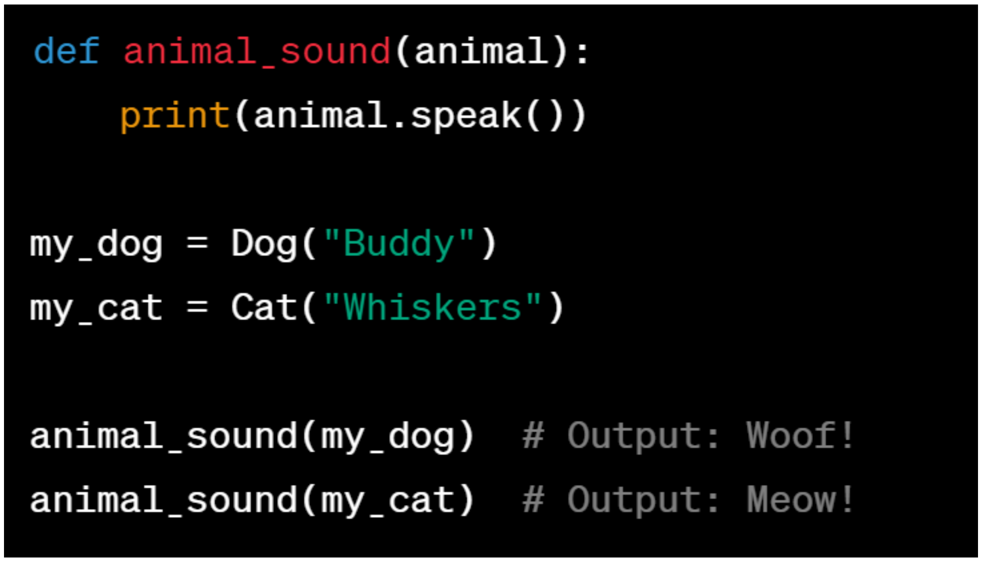
- Create derived classes (subclasses) that inherit from the base class. Add specific attributes and methods.
- Example:



```python
class Dog(Animal):
    def speak(self):
        return "Woof!"

class Cat(Animal):
    def speak(self):
        return "Meow!"
```

4. **Polymorphism:**

- Objects of different classes can be used interchangeably if they share a common base class. This is known as polymorphism.
- Example:



```python
def animal_sound(animal):
    print(animal.speak())

my_dog = Dog("Buddy")
my_cat = Cat("Whiskers")

animal_sound(my_dog)  # Output: Woof!
animal_sound(my_cat)  # Output: Meow!
```

Additional Tips:

- **Design Patterns:** Consider using design patterns such as Singleton, Factory, or Observer to address common design challenges.

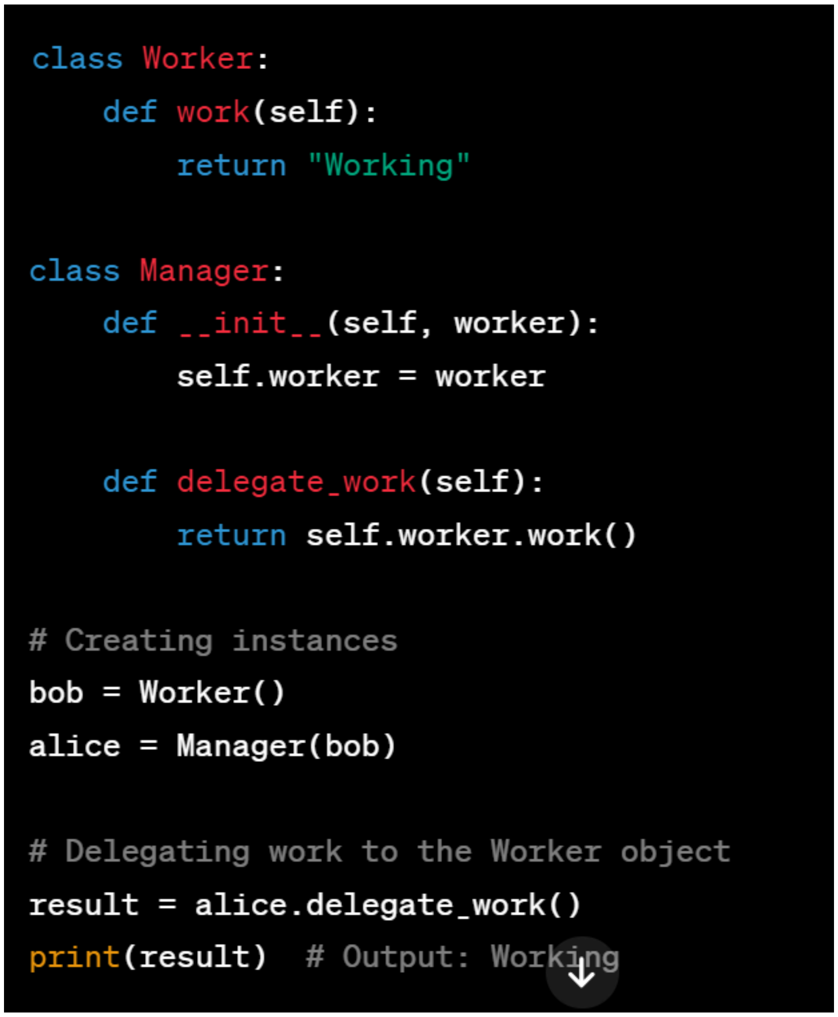
- **UML Diagrams:** Use Unified Modeling Language (UML) diagrams to visualize and plan your class hierarchy and relationships.

Effective class design enhances code readability, reusability, and maintainability. It also facilitates collaborative development by providing a clear structure for the software.

DELEGATION

In Python, delegation is a programming concept where an object passes on a specific task to another object, allowing the second object to handle that task on its behalf. This is typically achieved by calling a method on another object to perform the desired operation. Delegation is a way to achieve code reuse and maintainability by distributing responsibilities among different objects.

Here's an example of delegation in Python:



```python
class Worker:
    def work(self):
        return "Working"

class Manager:
    def __init__(self, worker):
        self.worker = worker

    def delegate_work(self):
        return self.worker.work()

# Creating instances
bob = Worker()
alice = Manager(bob)

# Delegating work to the Worker object
result = alice.delegate_work()
print(result)  # Output: Working
```

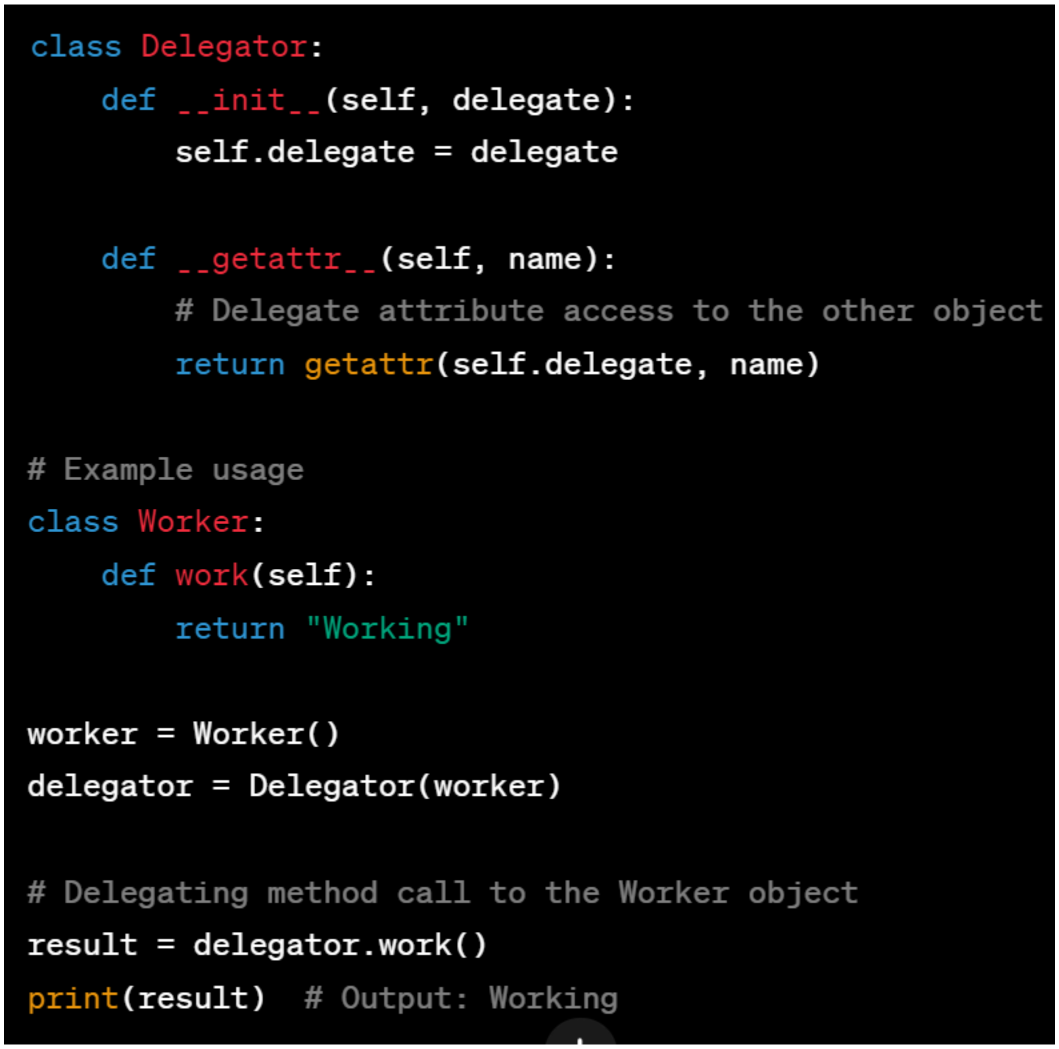
In this example:

- The **Worker** class has a method called **work** that performs a specific task.
- The **Manager** class has an instance variable **worker** of type **Worker**, and it has a method called **delegate_work**.

- When **delegate_work** is called on a **Manager** instance, it delegates the task to the **Worker** instance stored in its **worker** attribute.

This concept is a form of composition, where the **Manager** class delegates the responsibility of performing the actual work to another object (in this case, the **Worker** object). Delegation allows you to separate concerns and create more modular and maintainable code.

Another way to achieve delegation in Python is by using the **__getattr__** method. This allows an object to delegate attribute access to another object. Here's a simple example:



```python
class Delegator:
    def __init__(self, delegate):
        self.delegate = delegate

    def __getattr__(self, name):
        # Delegate attribute access to the other object
        return getattr(self.delegate, name)

# Example usage
class Worker:
    def work(self):
        return "Working"

worker = Worker()
delegator = Delegator(worker)

# Delegating method call to the Worker object
result = delegator.work()
print(result)  # Output: Working
```

In this example, the **Delegator** class forwards attribute access to its **delegate** object using the **__getattr__** method. This allows the **Delegator** object to delegate method calls or attribute access to another object transparently.

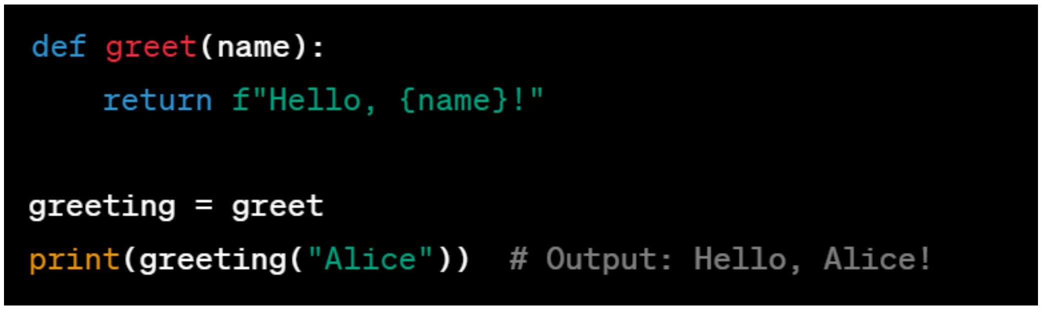
METHODS AND CLASSES ACT AS OBJECTS

In Python, functions and classes are first-class objects, which means they can be treated like any other object, such as integers, strings, or lists. This property allows you to pass functions or classes as arguments to other functions, return them from functions, and store them in data structures.

Understanding that functions and classes are objects in Python is fundamental to many programming paradigms, including functional programming and object-oriented programming.

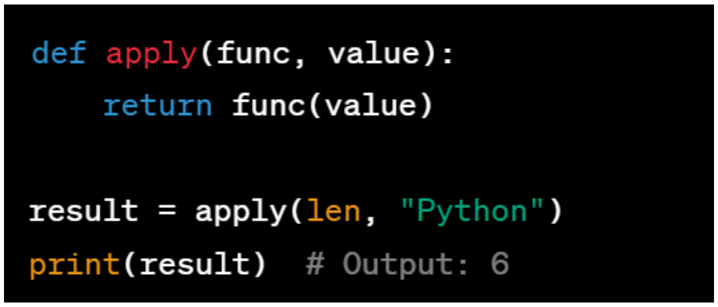
Functions as Objects:

1. **Assigning to Variables:**



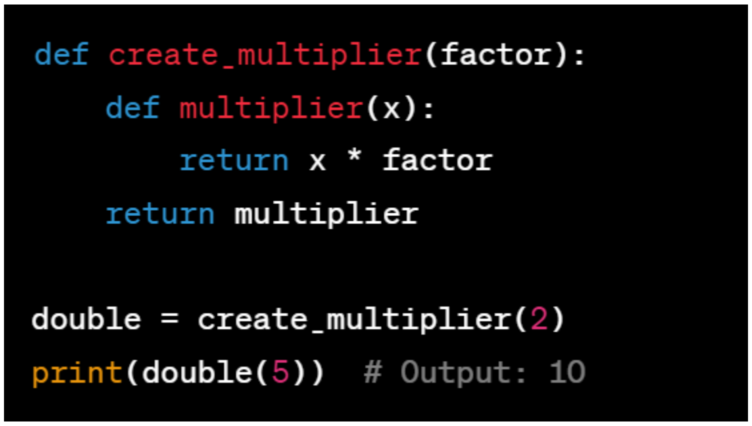
```python
def greet(name):
    return f"Hello, {name}!"

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

2. **Passing as Arguments:**



```python
def apply(func, value):
    return func(value)

result = apply(len, "Python")
print(result)  # Output: 6
```

3. **Returning from Functions:**



```python
def create_multiplier(factor):
    def multiplier(x):
        return x * factor
    return multiplier

double = create_multiplier(2)
print(double(5))  # Output: 10
```

Classes as Objects:

1. **Assigning to Variables:**

```python
class Dog:
    def bark(self):
        return "Woof!"

my_pet = Dog
instance = my_pet()
print(instance.bark())  # Output: Woof!
```

2. **Passing as Arguments:**

```python
def perform_action(obj):
    return obj.bark()

result = perform_action(Dog())
print(result)  # Output: Woof!
```

3. **Returning from Functions:**

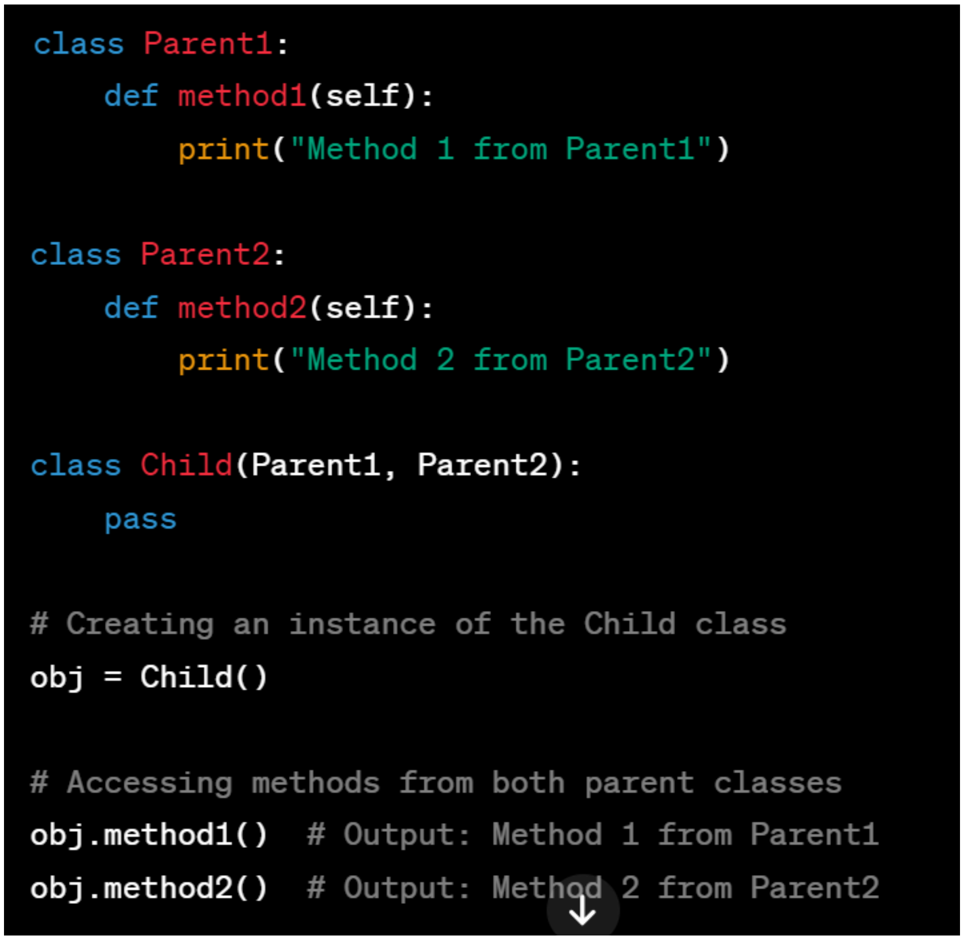
```python
def create_animal(animal_type):
    class Animal:
        def sound(self):
            return f"{animal_type} sound"

    return Animal

cat_class = create_animal("Cat")
cat_instance = cat_class()
print(cat_instance.sound())  # Output: Cat sound
```

Understanding that functions and classes are objects in Python opens the door to powerful programming techniques. It allows for dynamic code, such as creating functions or classes on the fly, and enables patterns like decorators and metaclasses in Python. This flexibility contributes to the expressiveness and richness of the language.

MULTIPLE INHERITANCE

Multiple inheritance in Python occurs when a class inherits from more than one parent class. This means that the subclass has access to the attributes and methods of all its parent classes. While Python supports multiple inheritance, it's essential to use it carefully to avoid potential issues such as the diamond problem.

Basic Syntax:



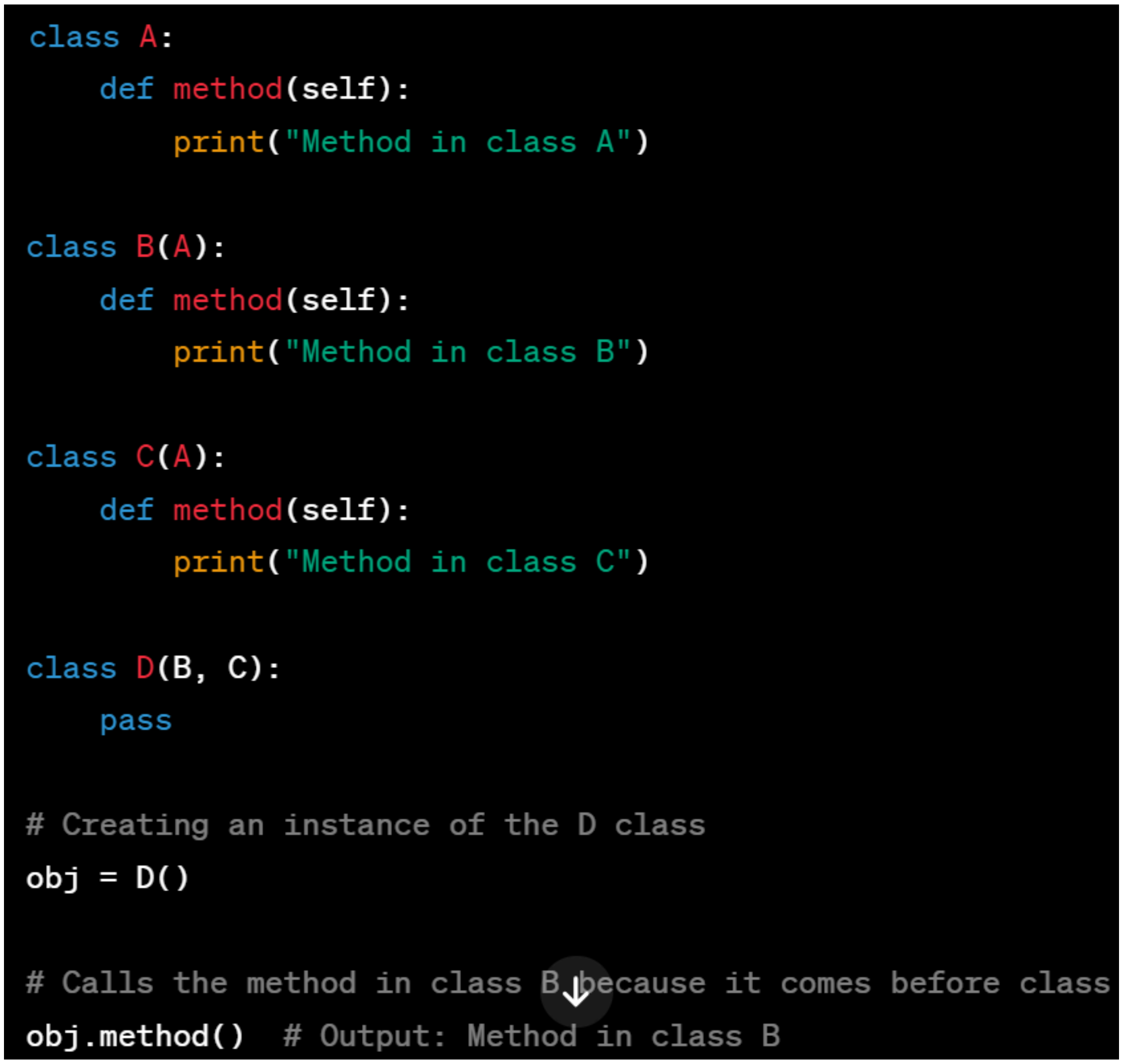
```python
class Parent1:
    def method1(self):
        print("Method 1 from Parent1")

class Parent2:
    def method2(self):
        print("Method 2 from Parent2")

class Child(Parent1, Parent2):
    pass

# Creating an instance of the Child class
obj = Child()

# Accessing methods from both parent classes
obj.method1()  # Output: Method 1 from Parent1
obj.method2()  # Output: Method 2 from Parent2
```

In the example above, the **Child** class inherits from both **Parent1** and **Parent2**. Instances of the **Child** class have access to the methods defined in both parent classes.

Diamond Problem:

The diamond problem is a potential issue in multiple inheritance when a subclass inherits from two classes that have a common ancestor. It can lead to ambiguity if there are conflicting method or attribute names in the common ancestor. Python resolves the diamond problem by using the C3 linearization algorithm to maintain a consistent and predictable MRO.

Example with the Diamond Problem:



```python
class A:
    def method(self):
        print("Method in class A")

class B(A):
    def method(self):
        print("Method in class B")

class C(A):
    def method(self):
        print("Method in class C")

class D(B, C):
    pass

# Creating an instance of the D class
obj = D()

# Calls the method in class B because it comes before class
obj.method()  # Output: Method in class B
```

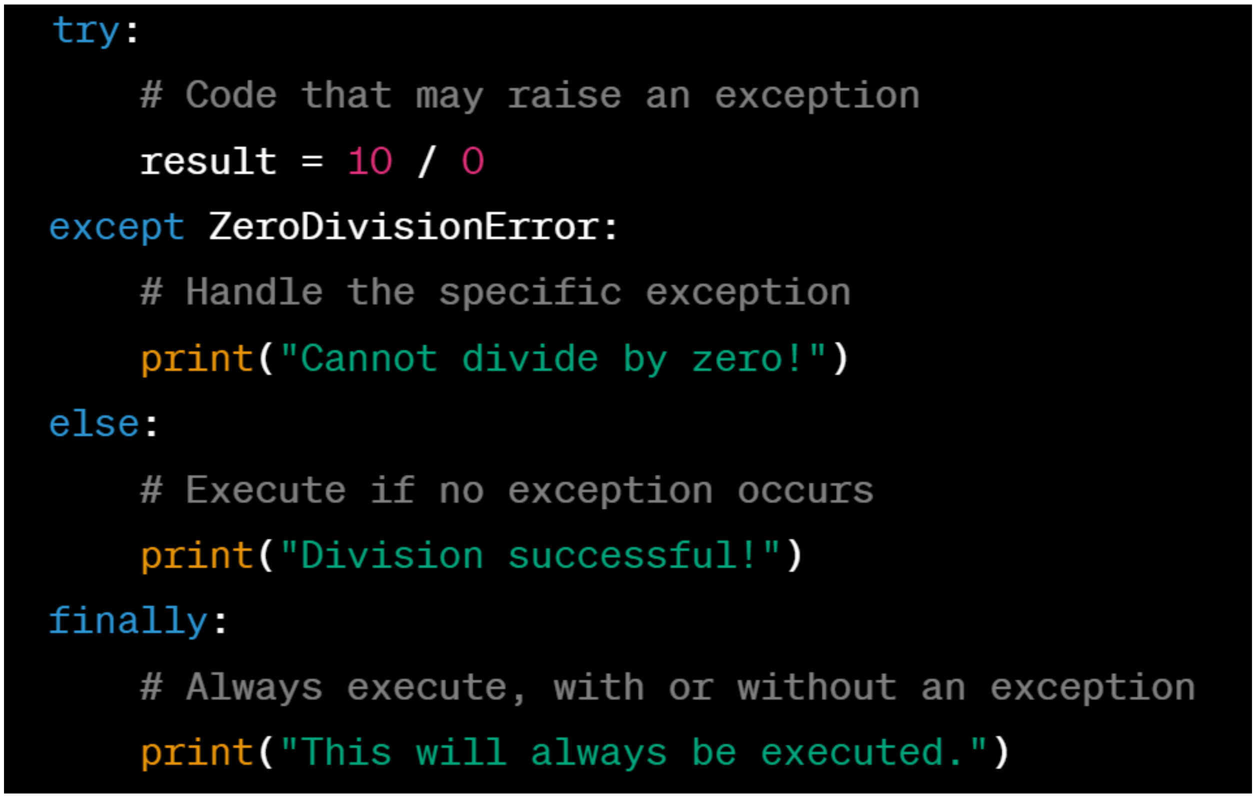
It's important to be aware of potential issues and design class hierarchies carefully when using multiple inheritance. Proper use of interfaces, abstract classes, and well-defined hierarchies can help mitigate problems associated with the diamond problem.

UNIT – 4

EXCEPTIONS

In Python, exceptions are events or errors that occur during the execution of a program. When an exceptional situation arises, and exception is raised, and the normal flow of the program is disrupted. To handle the exceptions, Python provides a mechanism using the ‘try’, ‘except’, ‘else’, and ‘finally’ blocks.

Basic exceptional handling



```
try:
    # Code that may raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Handle the specific exception
    print("Cannot divide by zero!")
else:
    # Execute if no exception occurs
    print("Division successful!")
finally:
    # Always execute, with or without an exception
    print("This will always be executed.")
```

In the above example –

1. The Try block contains the code that may raise an exception.
2. The except block catches a specific exception (‘ZeroDivisionError’ in this case) and handles it.
3. The else block is executed if no exception occurs in the try block.
4. The finally block always executes, whether an exception occurred or not.

BUILT-IN EXCEPTIONS

In python, there are several built-in exceptions that represent different types of errors or exceptional situations that can occur during the execution of a program. Understanding these exceptions can help you handle errors effectively.

Here are some important built-in exceptions that are commonly used –

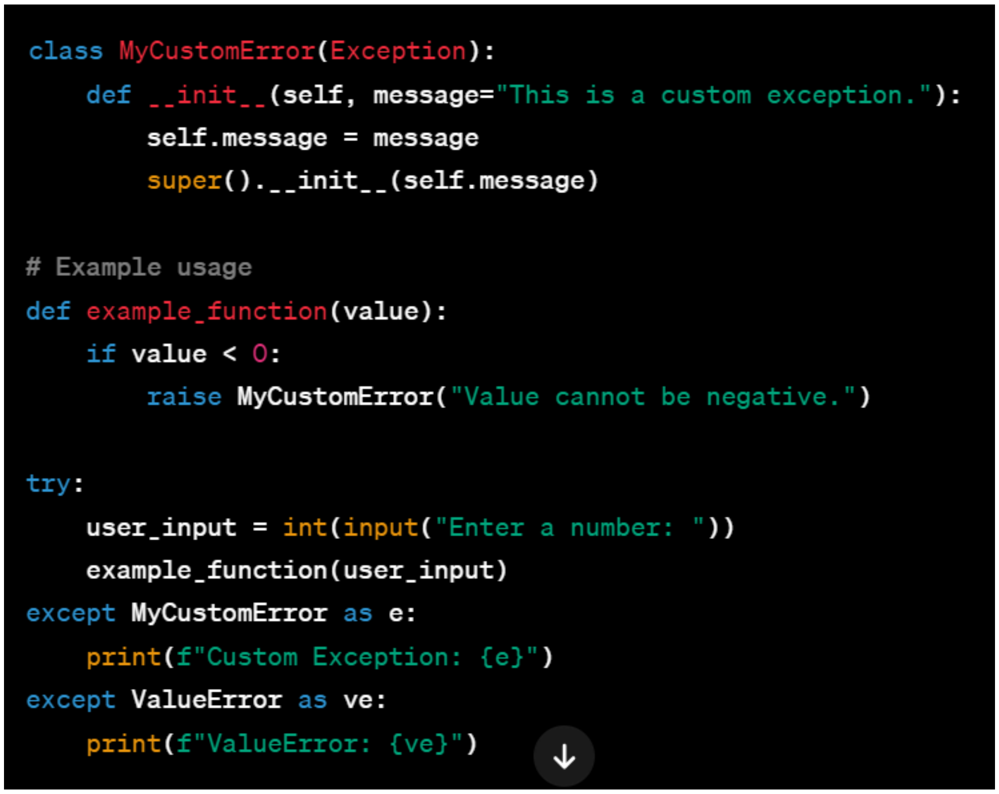
1. SyntaxError – Raised when the python interpreter encounters a syntax error.
2. IndentationError – Raised when there is an incorrect indentation.
3. NameError – Raised when a local or global name is not found.
4. TypeError – Raised when an operation or function is applied to an object of an appropriate type.
5. ValueError – Raised when an operation or function receives an argument of the correct type but an inappropriate value.
6. ZeroDivisionError – Raised when division or module operation is performed with zero as the divisor.
7. KeyError – Raised when a dictionary key is not found.
8. IndexError – Raised when a sequence subscript is out of range.
9. FileNotFoundError – Raised when a file or directory is requested but cannot be found.
10. IOError – Raised when an input/output operation fails.
11. AttributeError – Raised when an attribute reference or assignment fails.
12. ImportError – Raised when an import statement fails to find the module or name being imported.
13. RuntimeError – Raised when an error occurs that doesn't belong to any specific built-in exception category.
14. OSError – The base class for all I/O-related errors.
15. KeyError – Raised when a dictionary key is not found.
16. AssertionError – Raised when an assert statement fails.
17. TypeError – Raised when an operation or function is applied to an object of an inappropriate type.
18. NotImplementedError – Raised when an abstract method that should be implemented in a subclass is not actually implemented.
19. KeyboardInterrupt – Raised when the user interrupts the execution of the program, typically by pressing Ctrl + C.
20. MemoryError – Raised when an operation runs out of memory.

These are just few examples of the many built-in exceptions provided by Python. Properly handling exceptions in your code can make it more robust and resilient to errors. You can catch exceptions using the try, except, else, and finally blocks, as demonstrated in the previous response about exception handling in python.

USER-DEFINED EXCEPTIONS

In Python, you can define your own custom exceptions by creating a new class that inherits from the built-in exception class or one of its subclasses. Custom exceptions allow you to handle specific errors or exceptional situations in your code in a way that makes sense for your application.

Here's an example –



```python
class MyCustomError(Exception):
    def __init__(self, message="This is a custom exception."):
        self.message = message
        super().__init__(self.message)

# Example usage
def example_function(value):
    if value < 0:
        raise MyCustomError("Value cannot be negative.")

try:
    user_input = int(input("Enter a number: "))
    example_function(user_input)
except MyCustomError as e:
    print(f"Custom Exception: {e}")
except ValueError as ve:
    print(f"ValueError: {ve}")
```

In this example, we have created a custom exception class called **MyCustomError**, which inherits from the **Exception** class. The **__init__** method is used to initialize the exception with an optional custom message.

The **example_function** function checks if the input value is negative, and if so, it raises the **MyCustomError** exception with a specific message.

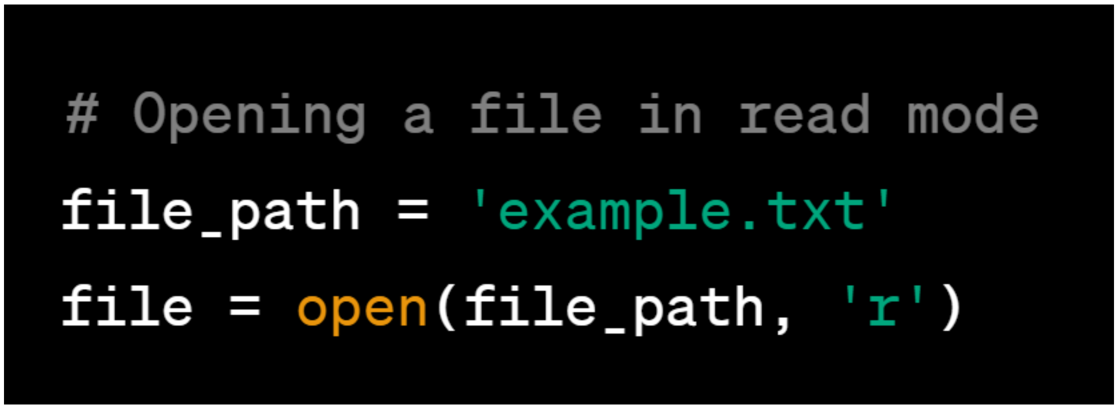
In the **try** block, we take user input, convert it to an integer, and call the **example_function**. If the input is negative, the custom exception is caught in the **except MyCustomError** block, and the custom message is printed. If the input is not a valid integer, a **ValueError** may occur, and it is caught in the **except ValueError** block.

FILE MANAGEMENT IN PYTHON

File management in python involves various operations on files, including opening, reading, writing, and closing files. Let’s cover the basic file operations, including opening files, specifying modes, working with file attributes, specifying encoding, and closing files.

Opening files –

To open a file in python, you can use the built-in open() function. It takes two arguments – The file path and the mode.



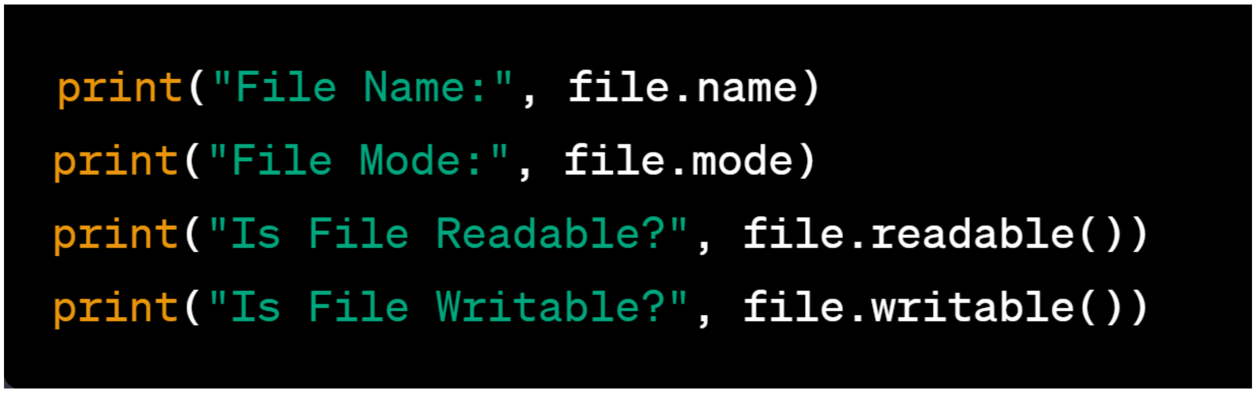
```
# Opening a file in read mode
file_path = 'example.txt'
file = open(file_path, 'r')
```

File Modes –

- ‘r’ – Read (default mode). Opens the file for reading.
- ‘w’ – Write. Opens the file for writing. Creates a new file or truncates the existing file to zero length.
- ‘a’ – Append. Opens the file for writing. Creates a new file or appends to the existing file.
- ‘b’ – Binary mode. Opens the file in binary mode (e.g., ‘rb’, or ‘wb’).
- ‘x’ – Exclusive creation. Creates a new file but raises an error if the file already exists.

File attributes –

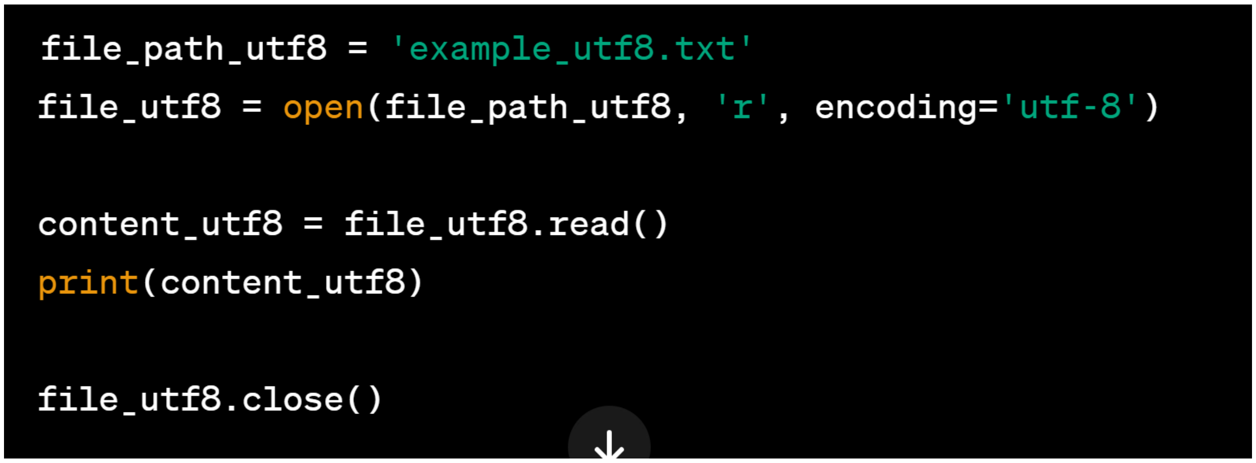
After opening a file, you can access various attributes, such as name, mode, and encoding.



```
print("File Name:", file.name)
print("File Mode:", file.mode)
print("Is File Readable?", file.readable())
print("Is File Writable?", file.writable())
```

Specifying encoding –

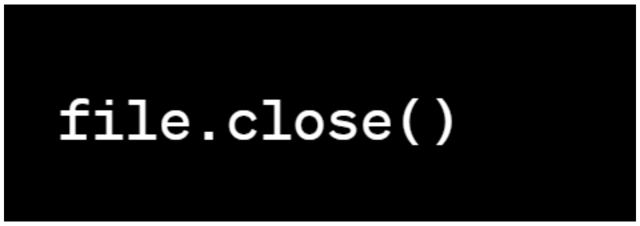
When working with text files, it's good practice to specify the encoding, especially when dealing with non-ASCII characters.



```python
file_path_utf8 = 'example_utf8.txt'
file_utf8 = open(file_path_utf8, 'r', encoding='utf-8')

content_utf8 = file_utf8.read()
print(content_utf8)

file_utf8.close()
```

Closing files –

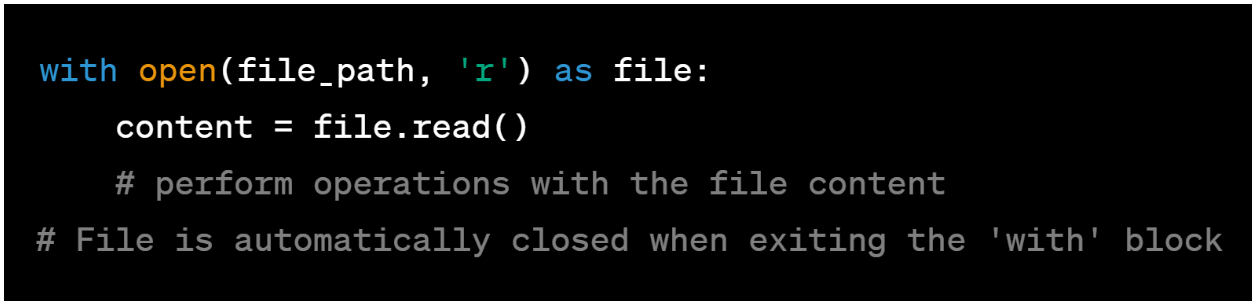
It's essential to close the files after you finish working with them to free up system resources. The 'close()' method is used for this purpose.



```python
file.close()
```

Always remember to close the files properly to avoid potential issues, especially when writing to files, as changes may not be saved if the file is not closed.

A more convenient way to work with files is using the 'with' statement. It automatically takes care of closing the file.



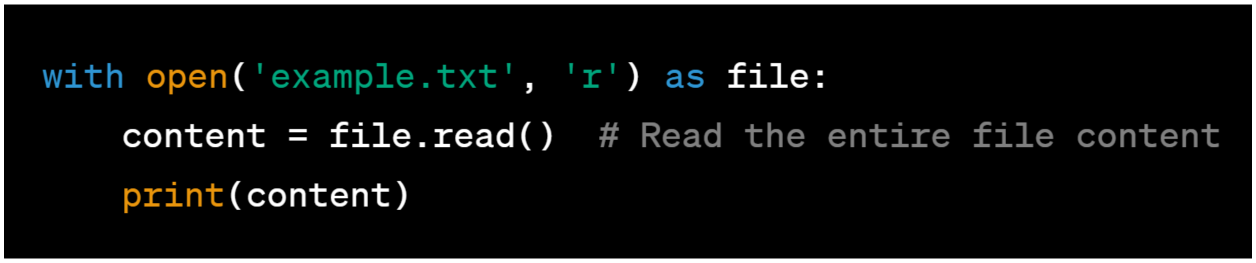
```python
with open(file_path, 'r') as file:
    content = file.read()
    # perform operations with the file content
# File is automatically closed when exiting the 'with' block
```

READ() AND WRITE () METHODS

In python, the read() and write() methods are used for reading and writing to files, respectively. Additionally, the tell() and seek() methods are used for file positioning, allowing you to determine and set the current position within a file.

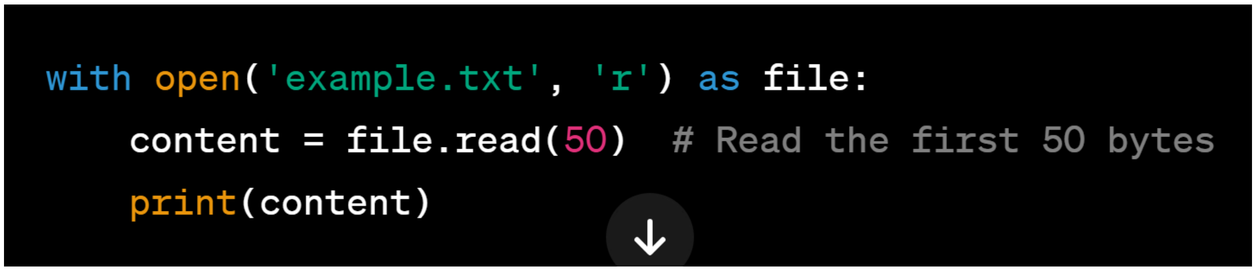
Read() method –

The read() method is used to read a specified number of bytes from the file or the entire file if no size is specified. It returns the content as a string.



```python
with open('example.txt', 'r') as file:
    content = file.read()  # Read the entire file content
    print(content)
```

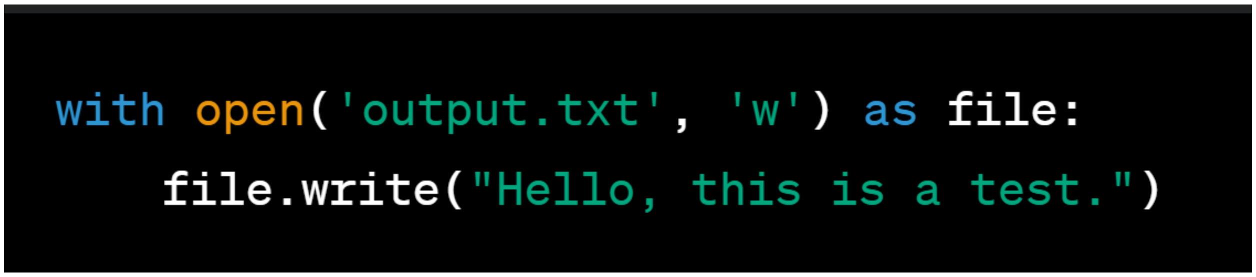
You can also specify the number of bytes to read –



```python
with open('example.txt', 'r') as file:
    content = file.read(50)  # Read the first 50 bytes
    print(content)
```

Write() method –

The write() method is used to write data to a file. It can be used to write strings or binary data, depending on the mode in which the file is opened.



```python
with open('output.txt', 'w') as file:
    file.write("Hello, this is a test.")
```

Tell() method –

The tell() method returns the current position (in bytes) of the file cursor. After reading or writing, the file cursor moves to the end of the read or written content.

```python
with open('example.txt', 'r') as file:
    content = file.read(20)
    position = file.tell()  # Get the current position
    print(f"Read content: {content}")
    print(f"Current position: {position}")
```

Seek() method –

The seek(offset, whence) method is used to change the current file cursor position. The offset parameter specifies the number of bytes to move, and the whence parameter determines the reference point.

- Whence = 0 (default) – Seek from the beginning of the file.
- Whence = 1 – Seek from the current position.
- Whence = 2 – Seek from the end of the file.

```python
with open('example.txt', 'r') as file:
    file.seek(10)  # Move 10 bytes from the beginning
    content = file.read(20)
    print(f"Read content: {content}")
```

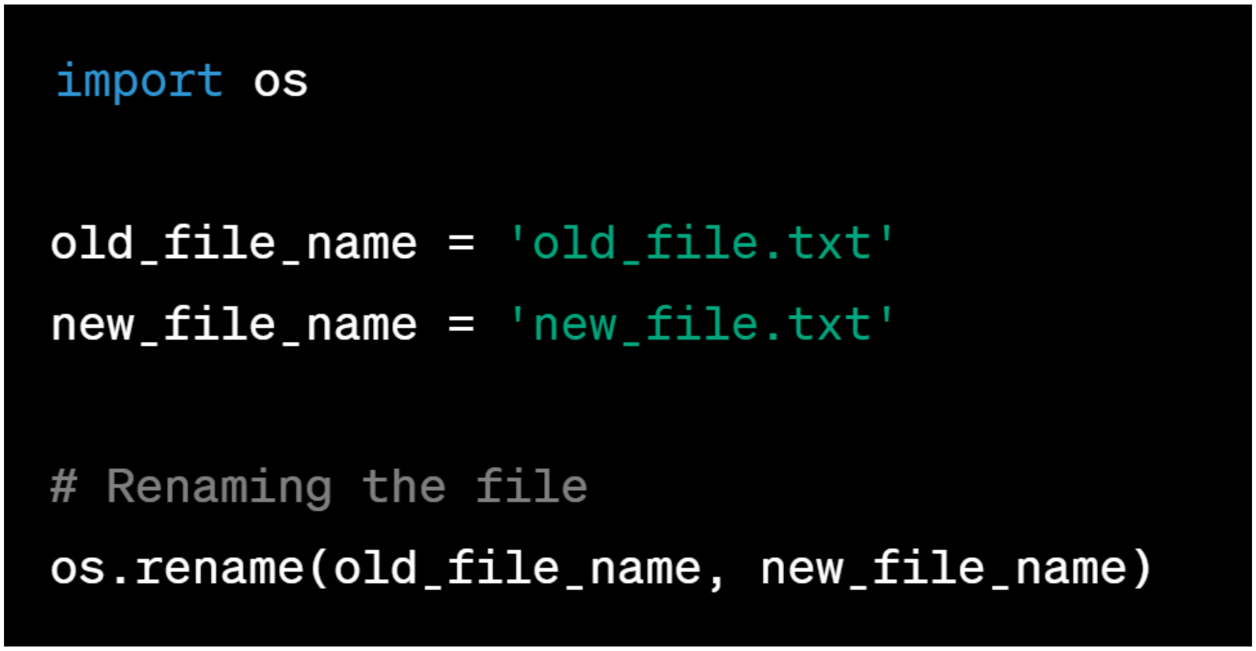
These file methods are essential for working with files in python, allowing you to read and write data and manage the file cursor position.

Renaming, deleting and directories

In Python, you can use the ‘os’ module to rename, delete files, and manipulate directories. Here’s an overview of how you can perform these operations –

Renaming a file –

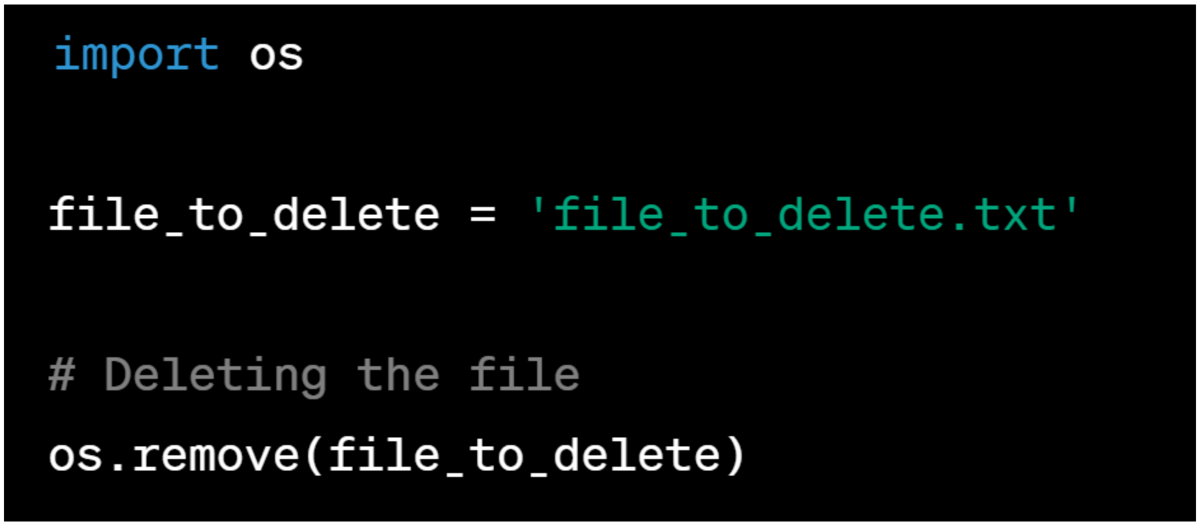
You can use the os.rename() method to rename a file.



```python
import os

old_file_name = 'old_file.txt'
new_file_name = 'new_file.txt'

# Renaming the file
os.rename(old_file_name, new_file_name)
```

Deleting a file –

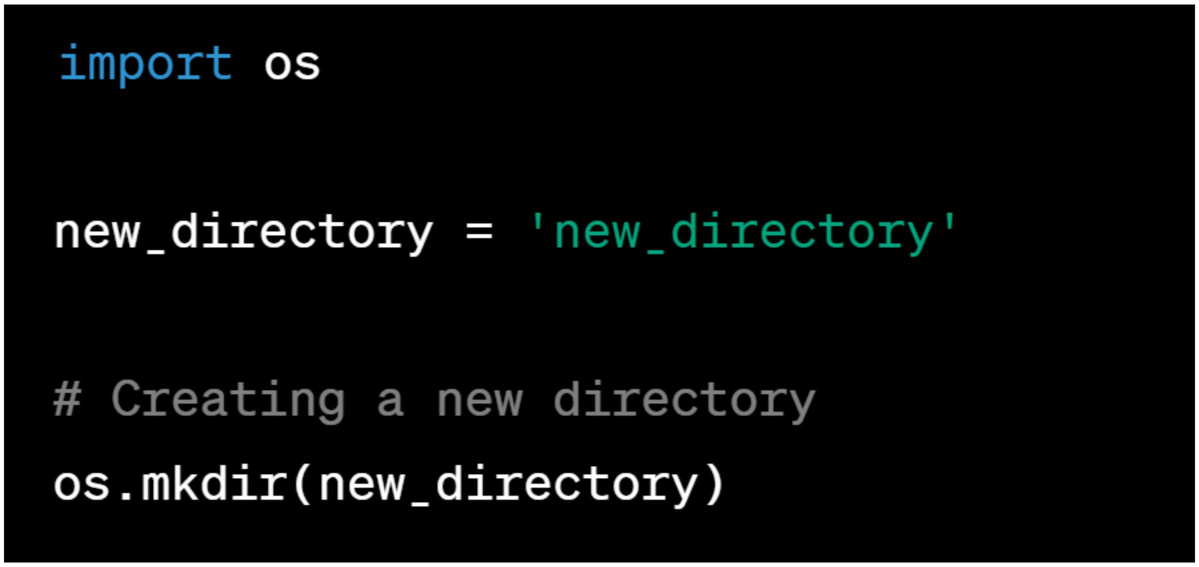
To delete a file, you can use the os.remove() method.



```python
import os

file_to_delete = 'file_to_delete.txt'

# Deleting the file
os.remove(file_to_delete)
```

Creating a directory –

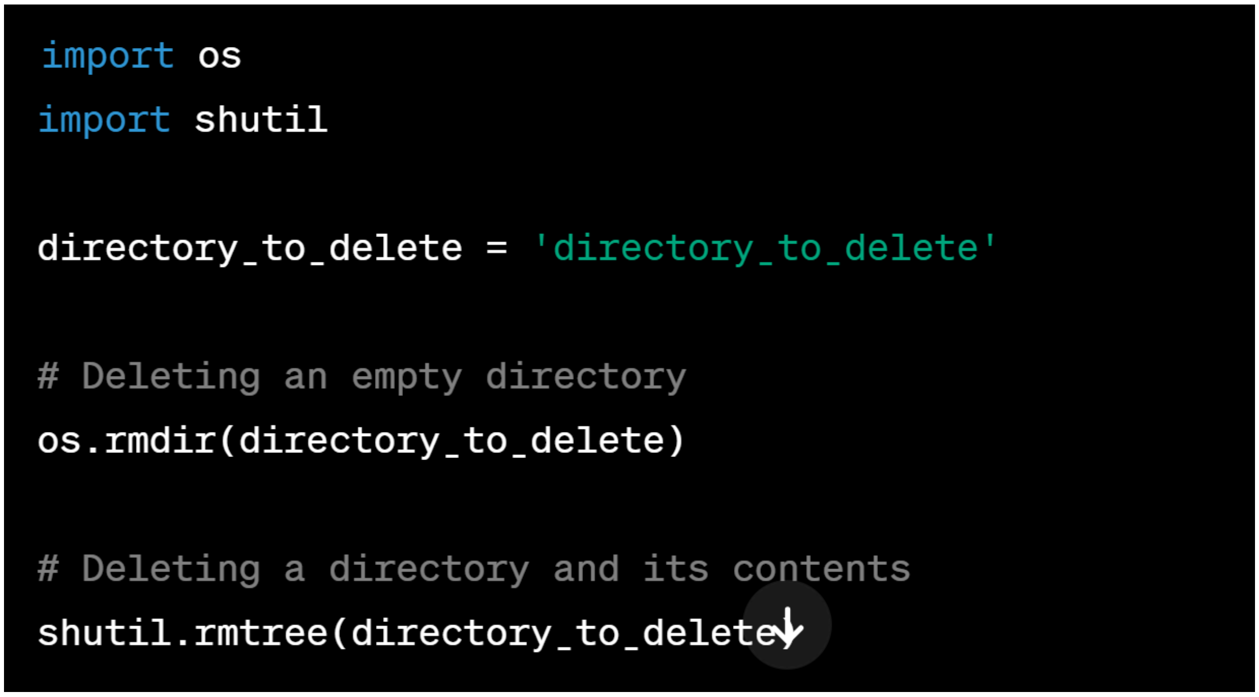
You can use the os.mkdir() method to create a new directory.



```
import os

new_directory = 'new_directory'

# Creating a new directory
os.mkdir(new_directory)
```

Deleting a directory –

To delete a directory, you can use the os.rmdir() method. Note that the directory must be empty for rmdir to succeed. If you want to delete a non-empty directory and its contents, you can use shutil.rmtree() from the shutil module.

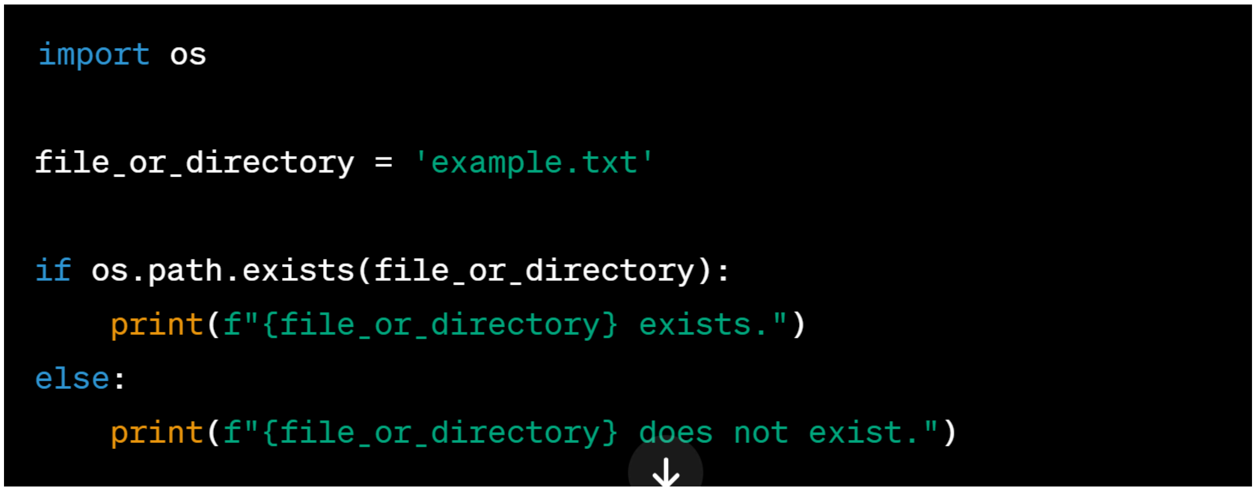


```
import os
import shutil

directory_to_delete = 'directory_to_delete'

# Deleting an empty directory
os.rmdir(directory_to_delete)

# Deleting a directory and its contents
shutil.rmtree(directory_to_delete
```

Checking if a file or directory exists –

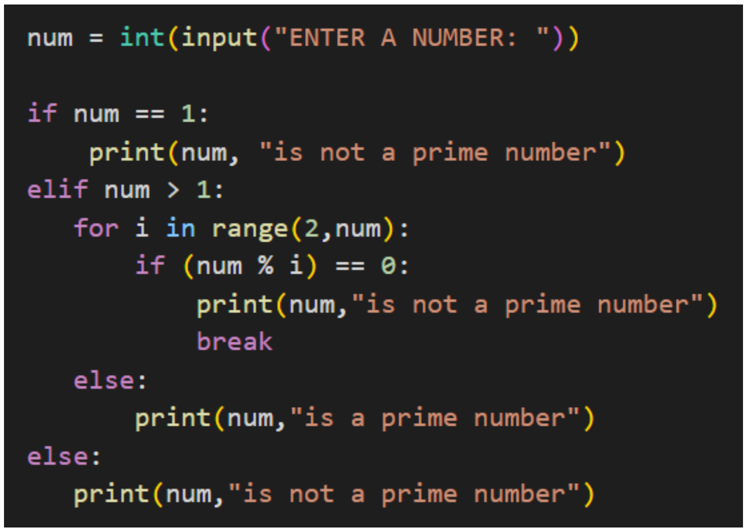
You can use the os.path.exists() method to check if a file or directory exists.

These are basic operations, you should use them carefully, especially when deleting files or directories. Always make sure that the file or directory you are about to delete is intended to be deleted to avoid data loss.

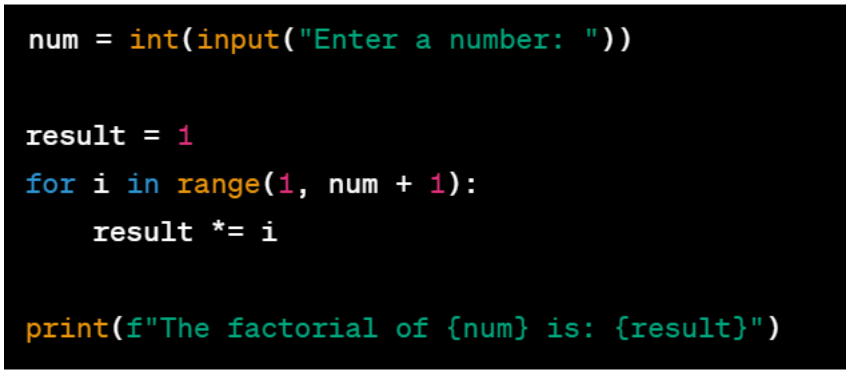


```python
import os

file_or_directory = 'example.txt'

if os.path.exists(file_or_directory):
    print(f"{file_or_directory} exists.")
else:
    print(f"{file_or_directory} does not exist.")
```

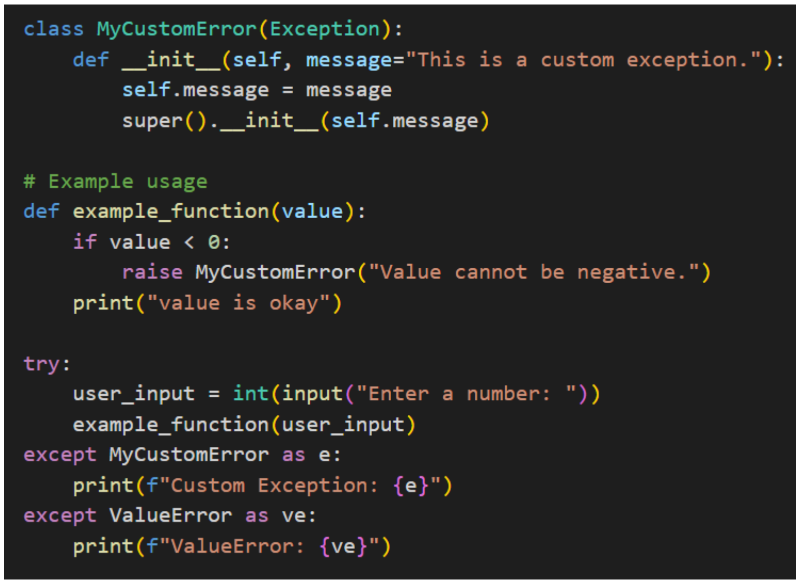
IMPORTANT CODES



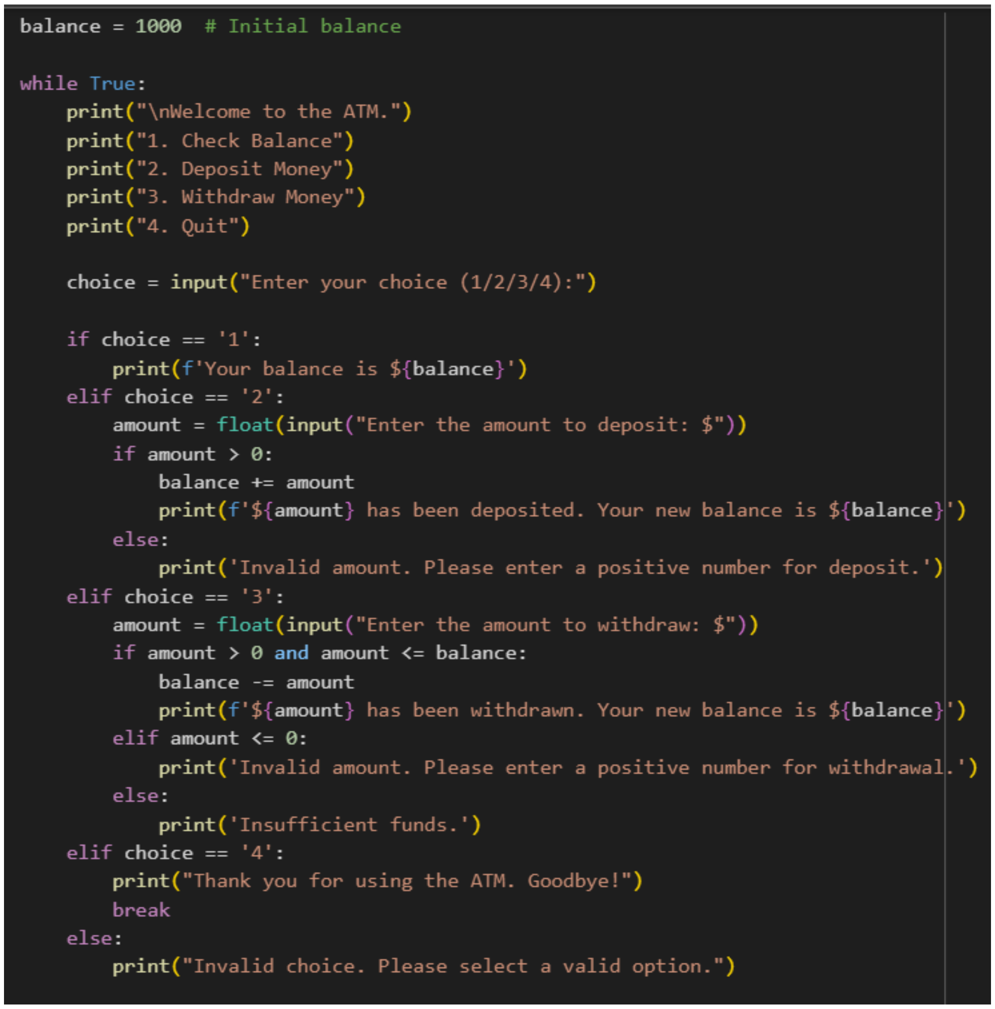
```
num = int(input("ENTER A NUMBER: "))

if num == 1:
    print(num, "is not a prime number")
elif num > 1:
   for i in range(2,num):
       if (num % i) == 0:
           print(num,"is not a prime number")
           break
   else:
       print(num,"is a prime number")
else:
   print(num,"is not a prime number")
```



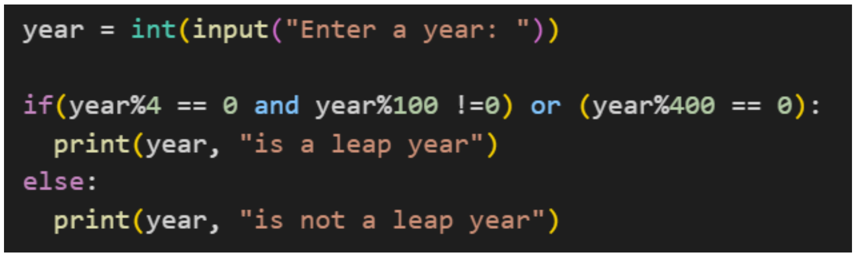
```
num = int(input("Enter a number: "))

result = 1
for i in range(1, num + 1):
    result *= i

print(f"The factorial of {num} is: {result}")
```



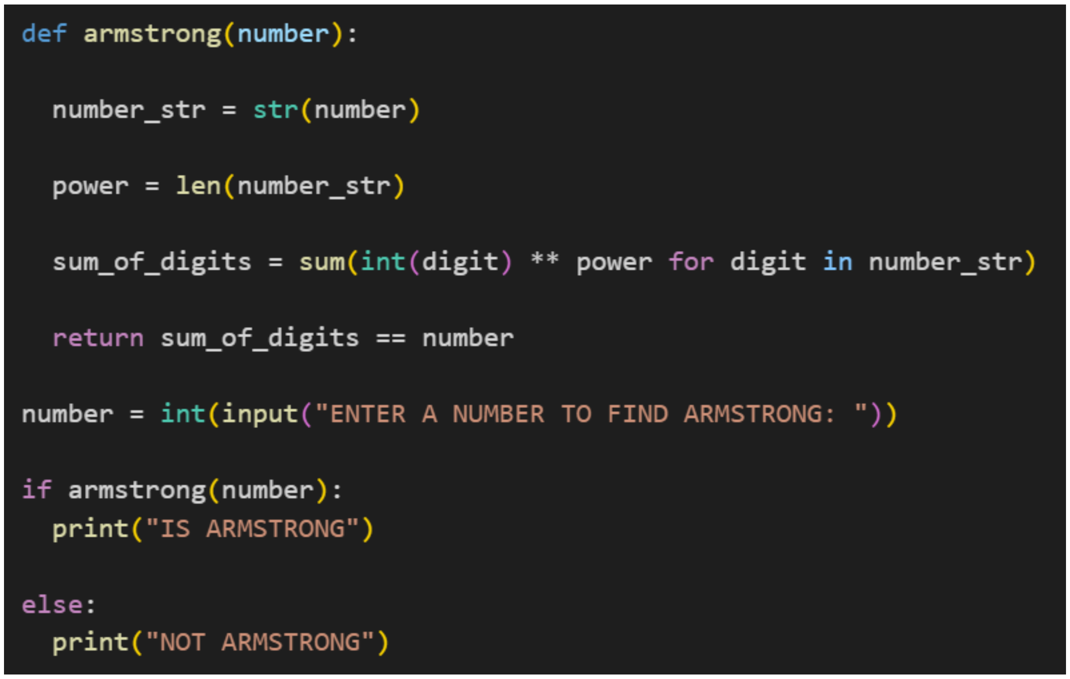
```
class MyCustomError(Exception):
    def __init__(self, message="This is a custom exception."):
        self.message = message
        super().__init__(self.message)

# Example usage
def example_function(value):
    if value < 0:
        raise MyCustomError("Value cannot be negative.")
    print("value is okay")

try:
    user_input = int(input("Enter a number: "))
    example_function(user_input)
except MyCustomError as e:
    print(f"Custom Exception: {e}")
except ValueError as ve:
    print(f"ValueError: {ve}")
```



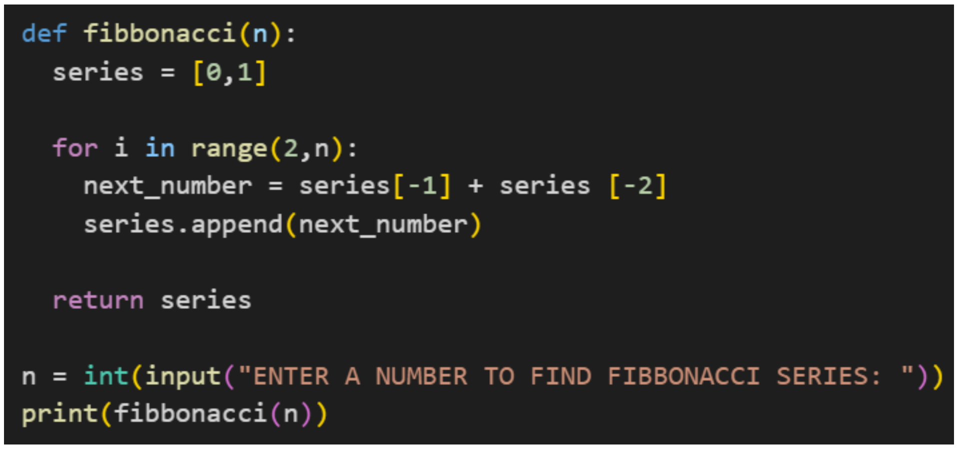
```python
balance = 1000  # Initial balance

while True:
    print("\nWelcome to the ATM.")
    print("1. Check Balance")
    print("2. Deposit Money")
    print("3. Withdraw Money")
    print("4. Quit")

    choice = input("Enter your choice (1/2/3/4):")

    if choice == '1':
        print(f'Your balance is ${balance}')
    elif choice == '2':
        amount = float(input("Enter the amount to deposit: $"))
        if amount > 0:
            balance += amount
            print(f'${amount} has been deposited. Your new balance is ${balance}')
        else:
            print('Invalid amount. Please enter a positive number for deposit.')
    elif choice == '3':
        amount = float(input("Enter the amount to withdraw: $"))
        if amount > 0 and amount <= balance:
            balance -= amount
            print(f'${amount} has been withdrawn. Your new balance is ${balance}')
        elif amount <= 0:
            print('Invalid amount. Please enter a positive number for withdrawal.')
        else:
            print('Insufficient funds.')
    elif choice == '4':
        print("Thank you for using the ATM. Goodbye!")
        break
    else:
        print("Invalid choice. Please select a valid option.")
```



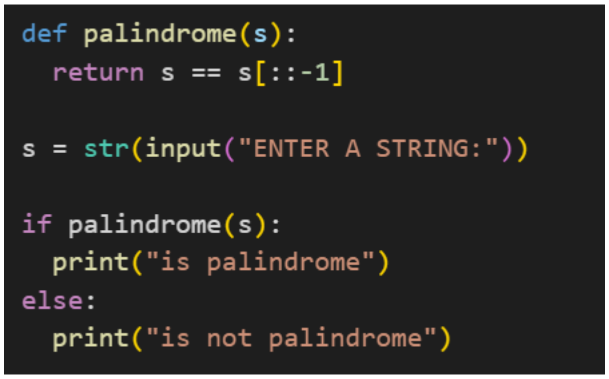
```python
year = int(input("Enter a year: "))

if(year%4 == 0 and year%100 !=0) or (year%400 == 0):
  print(year, "is a leap year")
else:
  print(year, "is not a leap year")
```



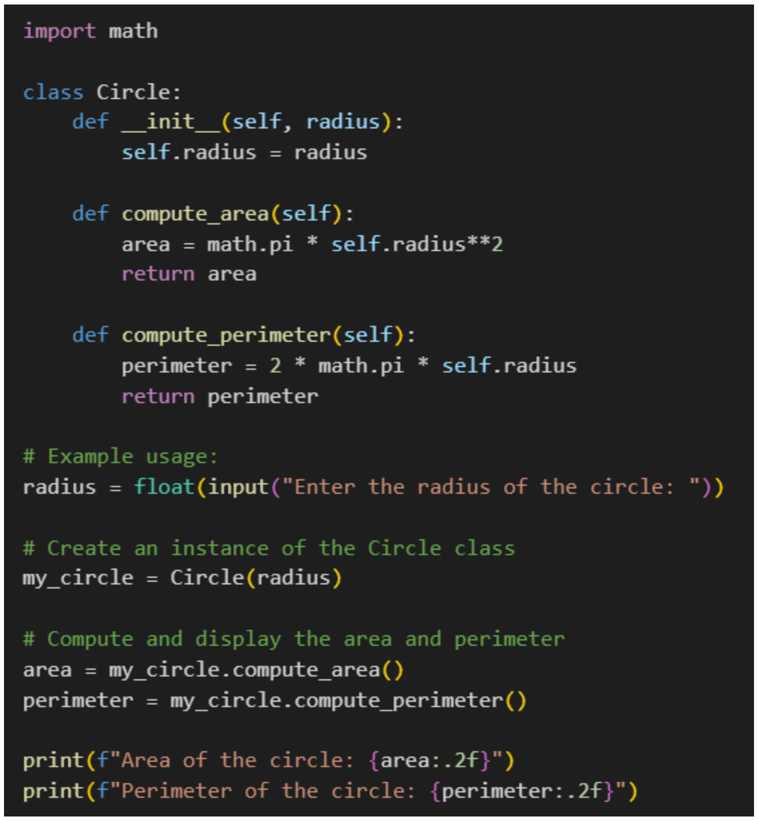
```python
def armstrong(number):

  number_str = str(number)

  power = len(number_str)

  sum_of_digits = sum(int(digit) ** power for digit in number_str)

  return sum_of_digits == number

number = int(input("ENTER A NUMBER TO FIND ARMSTRONG: "))

if armstrong(number):
  print("IS ARMSTRONG")

else:
  print("NOT ARMSTRONG")
```



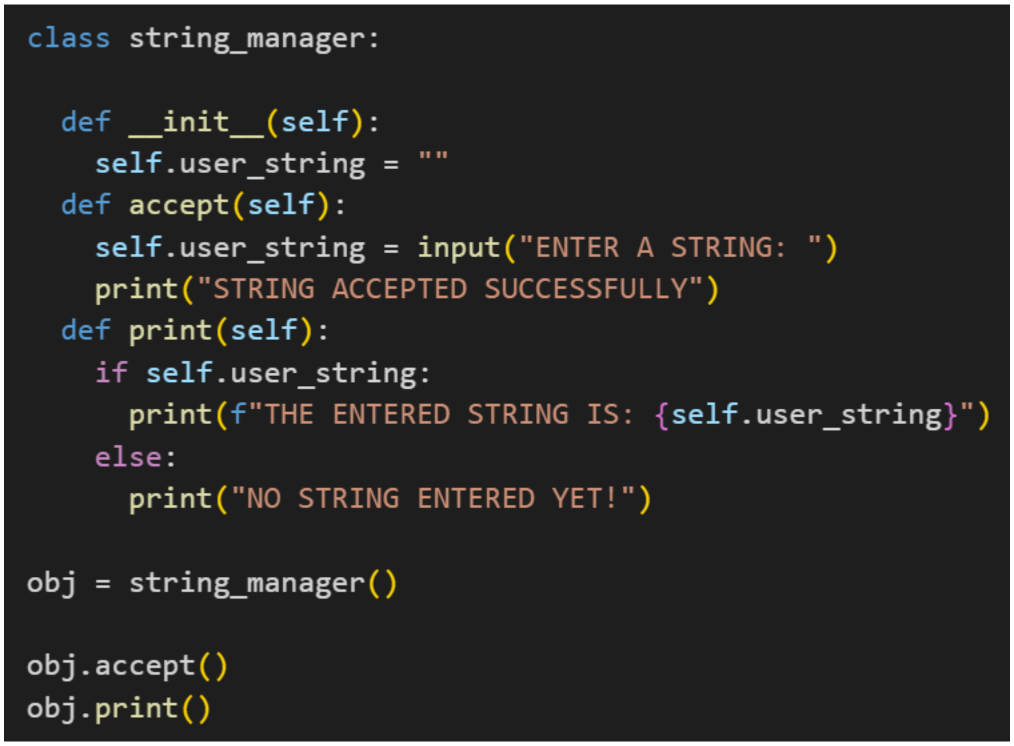
```python
def fibbonacci(n):
  series = [0,1]

  for i in range(2,n):
    next_number = series[-1] + series [-2]
    series.append(next_number)

  return series

n = int(input("ENTER A NUMBER TO FIND FIBBONACCI SERIES: "))
print(fibbonacci(n))
```



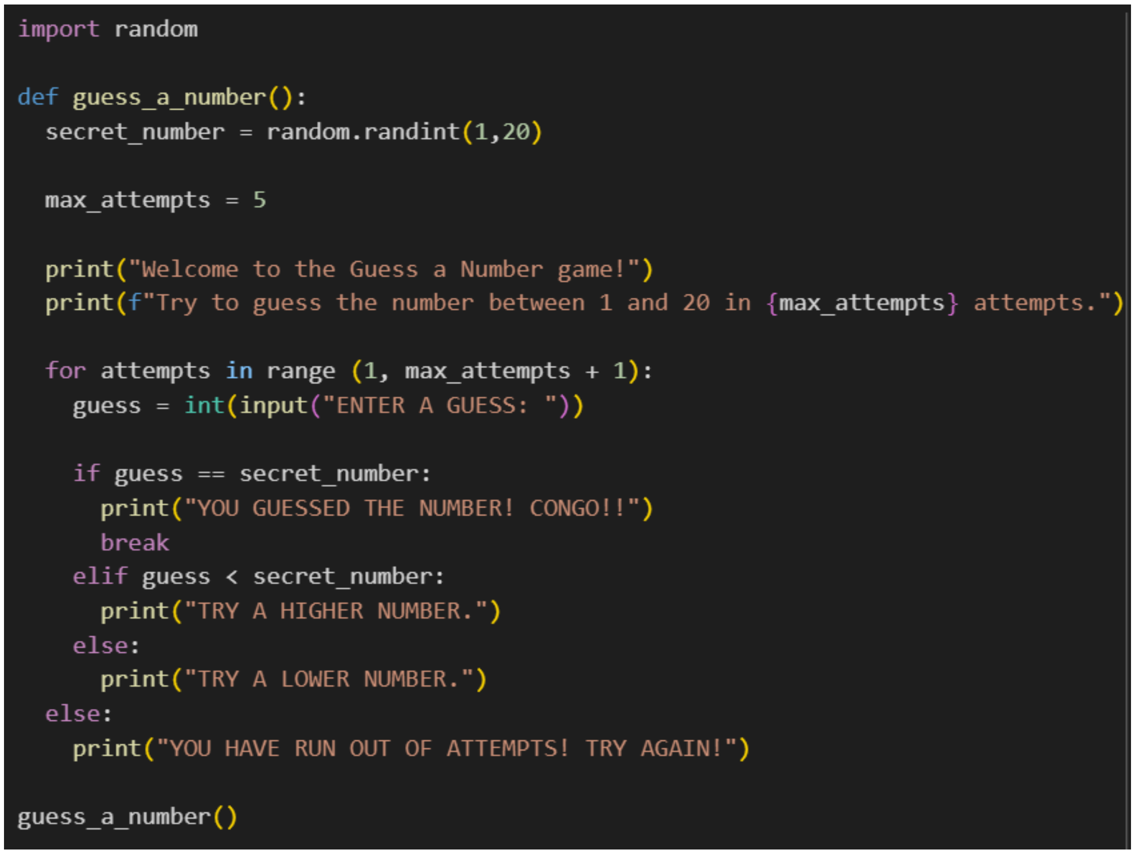
```python
def palindrome(s):
  return s == s[::-1]

s = str(input("ENTER A STRING:"))

if palindrome(s):
  print("is palindrome")
else:
  print("is not palindrome")
```

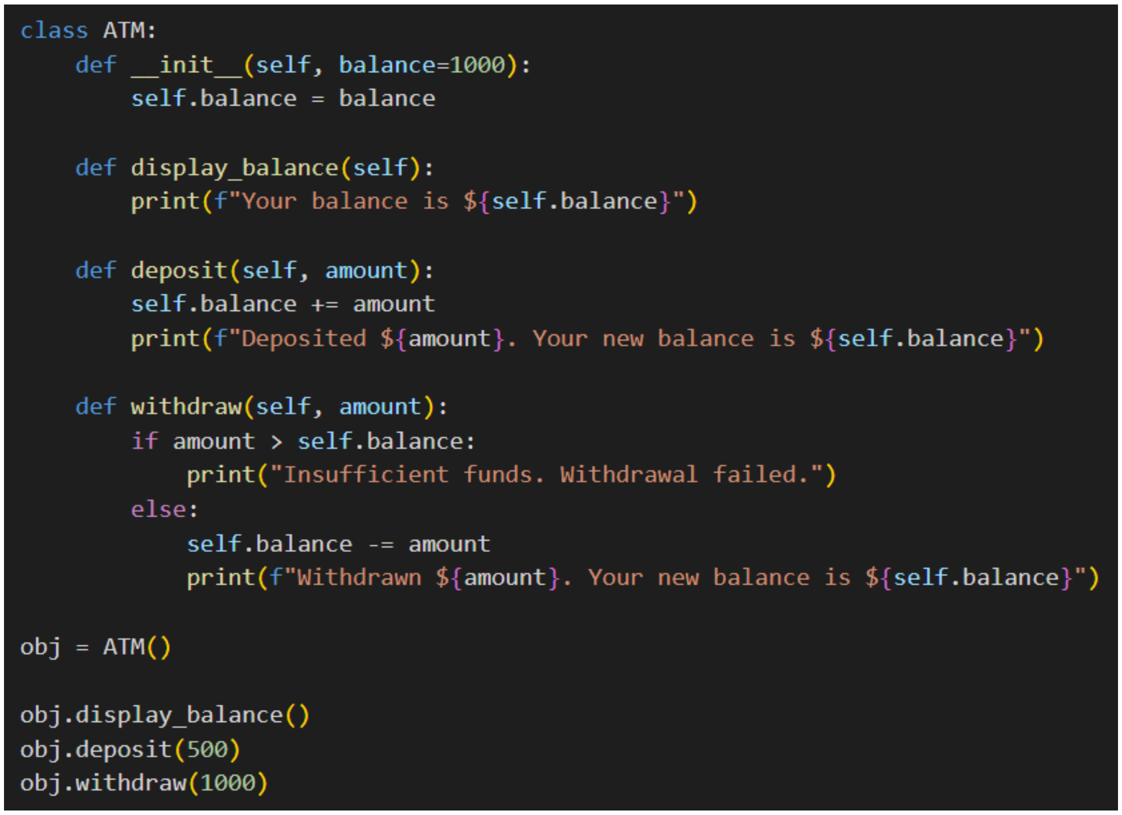


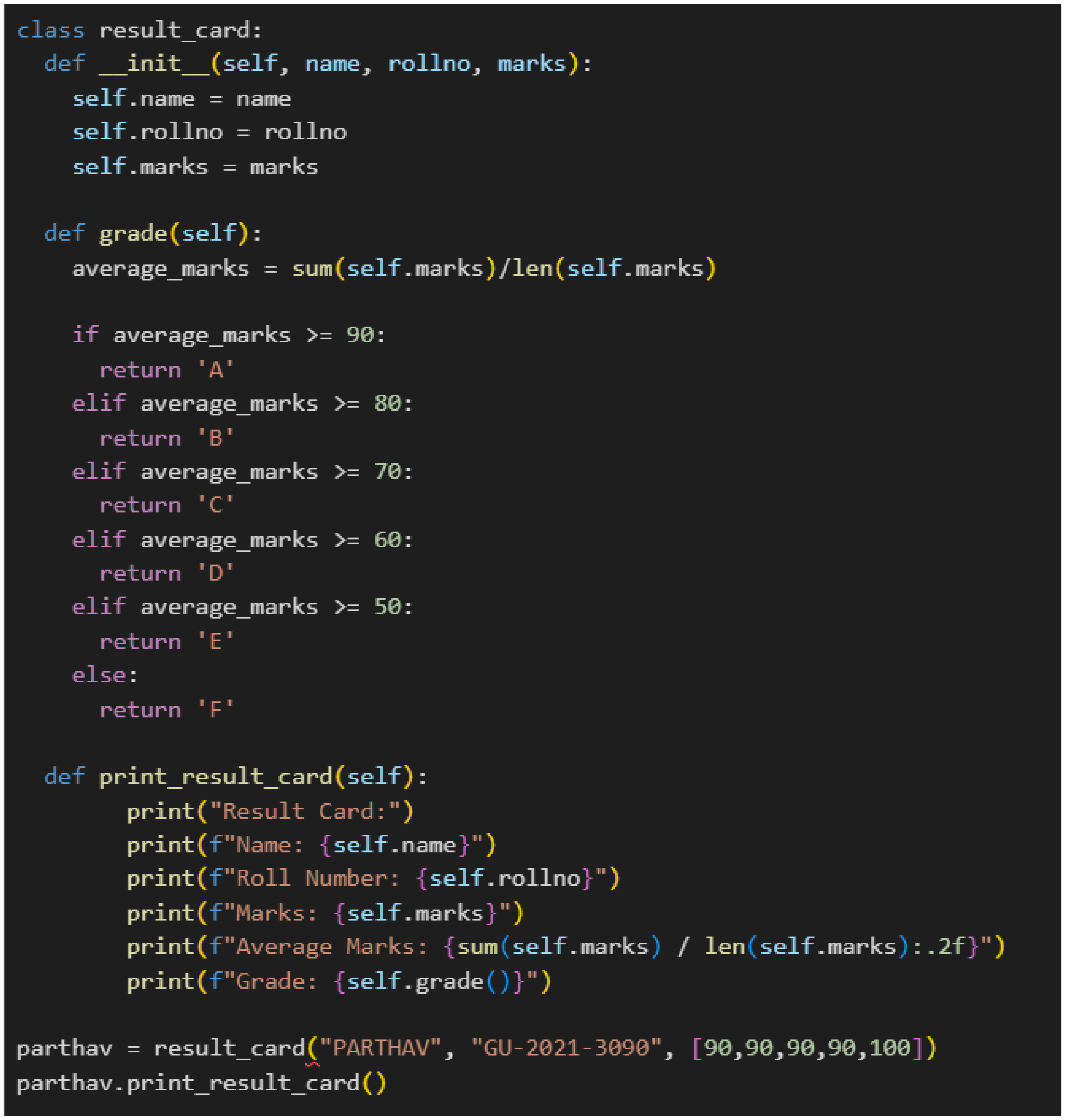
```python
import math

class Circle:
    def __init__(self, radius):
        self.radius = radius

    def compute_area(self):
        area = math.pi * self.radius**2
        return area

    def compute_perimeter(self):
        perimeter = 2 * math.pi * self.radius
        return perimeter

# Example usage:
radius = float(input("Enter the radius of the circle: "))

# Create an instance of the Circle class
my_circle = Circle(radius)

# Compute and display the area and perimeter
area = my_circle.compute_area()
perimeter = my_circle.compute_perimeter()

print(f"Area of the circle: {area:.2f}")
print(f"Perimeter of the circle: {perimeter:.2f}")
```



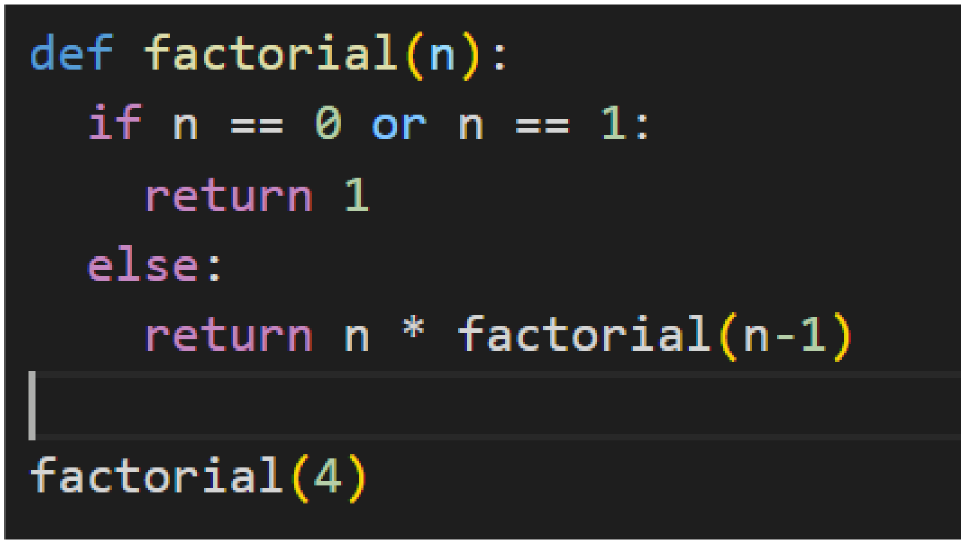
```python
class string_manager:

  def __init__(self):
    self.user_string = ""
  def accept(self):
    self.user_string = input("ENTER A STRING: ")
    print("STRING ACCEPTED SUCCESSFULLY")
  def print(self):
    if self.user_string:
      print(f"THE ENTERED STRING IS: {self.user_string}")
    else:
      print("NO STRING ENTERED YET!")

obj = string_manager()

obj.accept()
obj.print()
```


```python
import random

def guess_a_number():
  secret_number = random.randint(1,20)

  max_attempts = 5

  print("Welcome to the Guess a Number game!")
  print(f"Try to guess the number between 1 and 20 in {max_attempts} attempts.")

  for attempts in range (1, max_attempts + 1):
    guess = int(input("ENTER A GUESS: "))

    if guess == secret_number:
      print("YOU GUESSED THE NUMBER! CONGO!!")
      break
    elif guess < secret_number:
      print("TRY A HIGHER NUMBER.")
    else:
      print("TRY A LOWER NUMBER.")
  else:
    print("YOU HAVE RUN OUT OF ATTEMPTS! TRY AGAIN!")

guess_a_number()
```

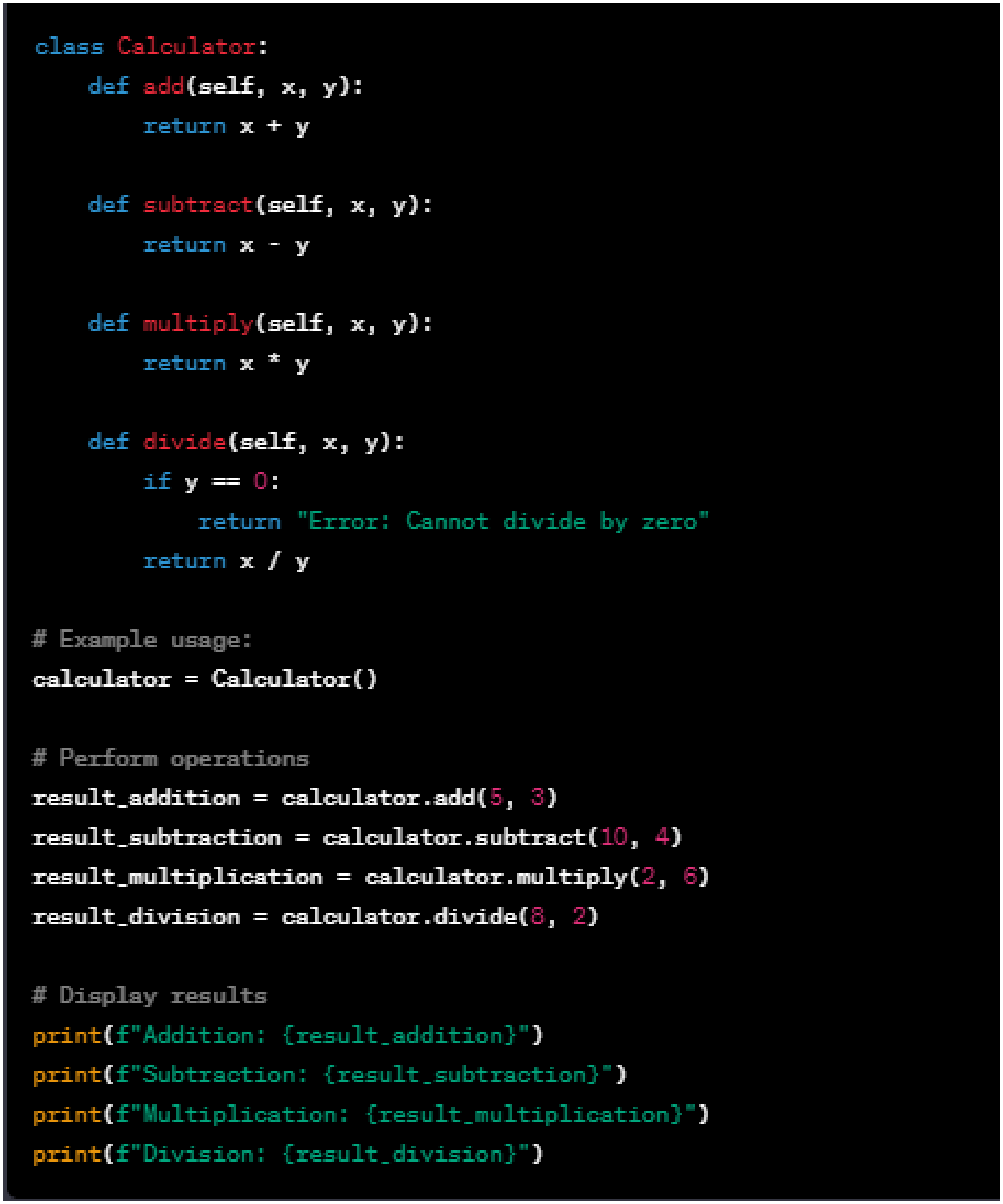

```python
class ATM:
    def __init__(self, balance=1000):
        self.balance = balance

    def display_balance(self):
        print(f"Your balance is ${self.balance}")

    def deposit(self, amount):
        self.balance += amount
        print(f"Deposited ${amount}. Your new balance is ${self.balance}")

    def withdraw(self, amount):
        if amount > self.balance:
            print("Insufficient funds. Withdrawal failed.")
        else:
            self.balance -= amount
            print(f"Withdrawn ${amount}. Your new balance is ${self.balance}")

obj = ATM()

obj.display_balance()
obj.deposit(500)
obj.withdraw(1000)
```



```python
class result_card:
  def __init__(self, name, rollno, marks):
    self.name = name
    self.rollno = rollno
    self.marks = marks

  def grade(self):
    average_marks = sum(self.marks)/len(self.marks)

    if average_marks >= 90:
      return 'A'
    elif average_marks >= 80:
      return 'B'
    elif average_marks >= 70:
      return 'C'
    elif average_marks >= 60:
      return 'D'
    elif average_marks >= 50:
      return 'E'
    else:
      return 'F'

  def print_result_card(self):
        print("Result Card:")
        print(f"Name: {self.name}")
        print(f"Roll Number: {self.rollno}")
        print(f"Marks: {self.marks}")
        print(f"Average Marks: {sum(self.marks) / len(self.marks):.2f}")
        print(f"Grade: {self.grade()}")

parthav = result_card("PARTHAV", "GU-2021-3090", [90,90,90,90,100])
parthav.print_result_card()
```



```python
def factorial(n):
  if n == 0 or n == 1:
    return 1
  else:
    return n * factorial(n-1)

factorial(4)
```



```python
class Calculator:
    def add(self, x, y):
        return x + y

    def subtract(self, x, y):
        return x - y

    def multiply(self, x, y):
        return x * y

    def divide(self, x, y):
        if y == 0:
            return "Error: Cannot divide by zero"
        return x / y

# Example usage:
calculator = Calculator()

# Perform operations
result_addition = calculator.add(5, 3)
result_subtraction = calculator.subtract(10, 4)
result_multiplication = calculator.multiply(2, 6)
result_division = calculator.divide(8, 2)

# Display results
print(f"Addition: {result_addition}")
print(f"Subtraction: {result_subtraction}")
print(f"Multiplication: {result_multiplication}")
print(f"Division: {result_division}")
```