Data Structure List:

A **list** is a built-in data structure used to store a collection of items. Lists are ordered, mutable (can be modified), and allow duplicate elements. They are one of the most commonly used data types in Python and are very versatile.

Key Features of Python Lists

1. **Ordered**: Items in a list have a defined order, and you can access them using their index.
2. **Mutable**: You can add, remove, or modify items after the list is created.
3. **Heterogeneous**: Lists can store elements of different data types (e.g., integers, strings, other lists).
4. **Dynamic**: Lists can grow or shrink in size as needed.
5. **Indexed**: Each item in a list has an index, starting from 0 for the first element.

Creating a List:

You can create a list by enclosing elements in square brackets [ ], separated by commas.

```
# List of integers
numbers = [1, 2, 3, 4, 5]

# List of strings
fruits = ["apple", "banana", "cherry"]

# Mixed data types
mixed_list = [1, "hello", 3.14, True]

# Nested list (list inside a list)
nested_list = [[1, 2, 3], ["a", "b", "c"]]
```

Accessing List Elements:

You can access elements in a list using their index. Python uses **zero-based indexing**, meaning the first element is at index 0.

```
fruits = ["apple", "banana", "cherry"]

# Access the first element
print(fruits[0])  # Output: apple

# Access the last element using negative indexing
print(fruits[-1])  # Output: cherry
```

```
apple
cherry
```

Common List Operations:

1. Adding Elements

- **append()**: Adds an element to the end of the list.
- **insert()**: Inserts an element at a specific index.
- **extend()**: Adds multiple elements (from another list) to the end of the list.

```
fruits = ["apple", "banana"]

# Append an element
fruits.append("cherry")
print(fruits)

# Insert an element at index 1
fruits.insert(1, "blueberry")
print(fruits)

# Extend the list with another list
fruits.extend(["orange", "grape"])
print(fruits)
```

```
['apple', 'banana', 'cherry']
['apple', 'blueberry', 'banana', 'cherry']
['apple', 'blueberry', 'banana', 'cherry', 'orange', 'grape']
```

2. Removing Elements

- **remove()**: Removes the first occurrence of a specific value.
- **pop()**: Removes and returns the element at a specific index (default is the last element).
- **del**: Deletes an element or a slice of the list.
- **clear()**: Removes all elements from the list.

```
fruits = ["apple", "banana", "cherry", "banana"]

# Remove the first occurrence of "banana"
fruits.remove("banana")
print(fruits)

# Remove and return the last element
popped_item = fruits.pop()
print(popped_item)
print(fruits)

# Delete the first element
del fruits[0]
print(fruits)

# Clear the list
fruits.clear()
print(fruits)
```

```
['apple', 'cherry', 'banana']
banana
['apple', 'cherry']
['cherry']
[]
```

3. Slicing Lists

You can extract a portion of a list using slicing. The syntax is list[start:stop:step].

```
numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9]

# Get elements from index 2 to 5 (exclusive)
print(numbers[2:5])

# Get every second element
print(numbers[::2])

# Reverse the list
print(numbers[::-1])
```

```
[2, 3, 4]
[0, 2, 4, 6, 8]
[9, 8, 7, 6, 5, 4, 3, 2, 1, 0]
```

4. List Comprehension

List comprehension is a concise way to create lists. It consists of an expression followed by a for clause.

```
# Create a list of squares
squares = [x ** 2 for x in range(10)]
print(squares)

# Create a list of even numbers
evens = [x for x in range(20) if x % 2 == 0]
print(evens)
```

```
[0, 1, 4, 9, 16, 25, 36, 49, 64, 81]
[0, 2, 4, 6, 8, 10, 12, 14, 16, 18]
```

5. Sorting Lists

- **sort()**: Sorts the list in place (modifies the original list).
- **sorted()**: Returns a new sorted list without modifying the original list.

```
numbers = [3, 1, 4, 1, 5, 9, 2]

# Sort the list in place
numbers.sort()
print(numbers)

# Create a new sorted list
sorted_numbers = sorted(numbers, reverse=True)
print(sorted_numbers)
```

```
[1, 1, 2, 3, 4, 5, 9]
[9, 5, 4, 3, 2, 1, 1]
```

6. Other Useful List Methods

- **len()**: Returns the number of elements in the list.
- **index()**: Returns the index of the first occurrence of a value.
- **count()**: Returns the number of times a value appears in the list.
- **copy()**: Returns a shallow copy of the list.

```
fruits = ["apple", "banana", "cherry", "banana"]

# Get the length of the list
print(len(fruits))

# Find the index of "banana"
print(fruits.index("banana"))

# Count occurrences of "banana"
print(fruits.count("banana"))

# Create a copy of the list
fruits_copy = fruits.copy()
print(fruits_copy)
```

```
4
1
2
['apple', 'banana', 'cherry', 'banana']
```

Nested Lists

Lists can contain other lists, allowing you to create multi-dimensional structures (e.g., matrices).

```
matrix = [
    [1, 2, 3],
    [4, 5, 6],
    [7, 8, 9]
]

# Access the element in the second row, third column
print(matrix[1][2])
```

```
6
```

Iterating Over Lists

You can use a for loop to iterate over the elements of a list.

```
fruits = ["apple", "banana", "cherry"]

# Print each fruit
for fruit in fruits:
    print(fruit)
```

```
apple
banana
cherry
```

Common Use Cases for Lists

1. Storing collections of data (e.g., student names, product prices).
2. Implementing stacks and queues.
3. Storing multi-dimensional data (e.g., matrices).
4. Processing sequences of data (e.g., filtering, mapping).

Tuple:

In Python, a tuple is a built-in data structure used to store a collection of items. Tuples are similar to lists, but they are immutable, meaning once a tuple is created, its elements cannot be changed, added, or removed. Tuples are ordered, allow duplicate elements, and can store elements of different data types.

Key Features of Python Tuples

1. Ordered: Items in a tuple have a defined order, and you can access them using their index.
2. Immutable: Once a tuple is created, it cannot be modified.
3. Heterogeneous: Tuples can store elements of different data types (e.g., integers, strings, other tuples).
4. Indexed: Each item in a tuple has an index, starting from 0 for the first element.
5. Lightweight: Tuples are faster and use less memory compare to lists because of their immutability.

Creating a Tuple

You can create a tuple by enclosing elements in parentheses ( ), separated by commas. If you create a tuple with a single element, you must include a trailing comma.

```python
# Tuple of integers
numbers = (1, 2, 3, 4, 5)

# Tuple of strings
fruits = ("apple", "banana", "cherry")

# Mixed data types
mixed_tuple = (1, "hello", 3.14, True)

# Single-element tuple (note the trailing comma)
single_element_tuple = (42,)

# Nested tuple (tuple inside a tuple)
nested_tuple = ((1, 2, 3), ("a", "b", "c"))
```

Accessing Tuple Elements:

You can access elements in a tuple using their index. Python uses zero-based indexing, meaning the first element is at index 0.

```
fruits = ("apple", "banana", "cherry")

# Access the first element
print(fruits[0])

# Access the last element using negative indexing
print(fruits[-1])
```

Tuples are Immutable:

Once a tuple is created, you cannot modify its elements. Attempting to do so will result in an error.

```
fruits = ("apple", "banana", "cherry")

# Trying to change an element will raise an error
fruits[1] = "blueberry"
```

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[17], line 4
      1 fruits = ("apple", "banana", "cherry")
      3 # Trying to change an element will raise an error
----> 4 fruits[1] = "blueberry"

TypeError: 'tuple' object does not support item assignment
```

Common Tuple Operations

1. Concatenating Tuples

You can combine tuples using the + operator.

```
tuple1 = (1, 2, 3)
tuple2 = ("a", "b", "c")

# Concatenate two tuples
combined_tuple = tuple1 + tuple2
print(combined_tuple)
```

```
(1, 2, 3, 'a', 'b', 'c')
```

2. Repeating Tuples

You can repeat a tuple using the * operator.

```
tuple1 = ("hello",)

# Repeat the tuple 3 times
repeated_tuple = tuple1 * 3
print(repeated_tuple)
```

```
('hello', 'hello', 'hello')
```

3. Slicing Tuples

You can extract a portion of a tuple using slicing. The syntax is tuple[start:stop:step].

```
numbers = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)

# Get elements from index 2 to 5 (exclusive)
print(numbers[2:5])

# Get every second element
print(numbers[::2])

# Reverse the tuple
print(numbers[::-1])
```

```
(2, 3, 4)
(0, 2, 4, 6, 8)
(9, 8, 7, 6, 5, 4, 3, 2, 1, 0)
```

4. Unpacking Tuples

You can unpack the elements of a tuple into variables.

```
fruits = ("apple", "banana", "cherry")

# Unpack the tuple into variables
fruit1, fruit2, fruit3 = fruits
print(fruit1)
print(fruit2)
print(fruit3)
```

```
apple
banana
cherry
```

5. Checking Membership

You can check if an element exists in a tuple using the in keyword.

```
fruits = ("apple", "banana", "cherry")

# Check if "banana" is in the tuple
print("banana" in fruits)
```

```
True
```

6. Tuple Methods

Tuples have only two built-in methods:

- **count()**: Returns the number of times a value appears in the tuple.
- **index()**: Returns the index of the first occurrence of a value.

```
numbers = (1, 2, 3, 2, 4, 2)

# Count occurrences of 2
print(numbers.count(2))

# Find the index of the first occurrence of 2
print(numbers.index(2))
```

```
3
1
```

Advantages of Tuples

1. **Immutable**: Tuples are safer to use when you want to ensure that the data cannot be changed.
2. **Faster**: Tuples are faster than lists because they are immutable and have a fixed size.
3. **Hashable**: Tuples can be used as keys in dictionaries, whereas lists cannot.

Python Set:

In Python, a set is a built-in data structure used to store a collection of unique and unordered elements. Sets are mutable, meaning you can add or remove elements, but the elements themselves must be immutable (e.g., numbers, strings, tuples). Sets are particularly useful for tasks that involve membership testing, removing duplicates, and performing mathematical set operations like union, intersection, and difference.

Key Features of Python Sets

1. **Unordered**: Elements in a set do not have a defined order.
2. **Unique**: Sets cannot contain duplicate elements.
3. **Mutable**: You can add or remove elements from a set.
4. **Heterogeneous**: Sets can store elements of different data types (e.g., integers, strings, tuples).
5. **No Indexing**: Since sets are unordered, you cannot access elements using indices.

Creating a Set

You can create a set by enclosing elements in curly braces {} or by using the set() constructor. If you use curly braces without any elements, it creates an empty dictionary, not a set. To create an empty set, you must use set().

```
# Set of integers
numbers = {1, 2, 3, 4, 5}

# Set of strings
fruits = {"apple", "banana", "cherry"}

# Mixed data types
mixed_set = {1, "hello", 3.14, True}

# Empty set (must use set())
empty_set = set()

# Duplicate elements are automatically removed
duplicate_set = {1, 2, 2, 3, 3, 3}
print(duplicate_set)
```

```
{1, 2, 3}
```

Accessing Set Elements

Since sets are unordered, you cannot access elements using indices. However, you can iterate over the elements using a for loop or check for membership using the in keyword.

```
fruits = {"apple", "banana", "cherry"}

# Iterate over the set
for fruit in fruits:
    print(fruit)

# Check if an element exists in the set
print("banana" in fruits)
```

```
banana
cherry
apple
True
```

Common Set Operations

1. Adding Elements

- **add()**: Adds a single element to the set.
- **update()**: Adds multiple elements (from another iterable) to the set.

```
fruits = {"apple", "banana"}

# Add a single element
fruits.add("cherry")
print(fruits)

# Add multiple elements
fruits.update(["orange", "grape"])
print(fruits)
```

```
{'banana', 'cherry', 'apple'}
{'banana', 'apple', 'orange', 'grape', 'cherry'}
```

2. Removing Elements

- **remove()**: Removes a specific element. Raises an error if the element does not exist.
- **discard()**: Removes a specific element. Does not raise an error if the element does not exist.
- **pop()**: Removes and returns an arbitrary element from the set.
- **clear()**: Removes all elements from the set.

```python
fruits = {"apple", "banana", "cherry"}

# Remove an element
fruits.remove("banana")
print(fruits)

# Discard an element (no error if not found)
fruits.discard("grape")
print(fruits)

# Remove and return an arbitrary element
popped_item = fruits.pop()
print(popped_item)
print(fruits)

# Clear the set
fruits.clear()
print(fruits)
```

```
{'cherry', 'apple'}
{'cherry', 'apple'}
cherry
{'apple'}
set()
```

3. Set Operations

Sets support mathematical set operations like union, intersection, difference, and symmetric difference.

- **Union (|)**: Combines elements from two sets.
- **Intersection (&)**: Returns elements common to both sets.
- **Difference (-)**: Returns elements in the first set but not in the second.
- **Symmetric Difference (^)**: Returns elements in either set but not in both.

```python
set1 = {1, 2, 3, 4}
set2 = {3, 4, 5, 6}

# Union
print(set1 | set2)

# Intersection
print(set1 & set2)

# Difference
print(set1 - set2)

# Symmetric Difference
print(set1 ^ set2)
```

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

4. Set Methods

- **len()**: Returns the number of elements in the set.
- **copy()**: Returns a shallow copy of the set.
- **isdisjoint()**: Returns True if two sets have no common elements.
- **issubset()**: Returns True if all elements of the set are present in another set.
- **issuperset()**: Returns True if the set contains all elements of another set.

```
set1 = {1, 2, 3}
set2 = {3, 4, 5}

# Length of the set
print(len(set1))

# Check if sets are disjoint
print(set1.isdisjoint(set2))

# Check if set1 is a subset of set2
print(set1.issubset(set2))

# Check if set1 is a superset of set2
print(set1.issuperset(set2))
```

```
3
False
False
False
```

Frozen Sets

A **frozen set** is an immutable version of a set. Once created, its elements cannot be changed. Frozen sets are hashable and can be used as keys in dictionaries.

```
# Create a frozen set
frozen_set = frozenset([1, 2, 3])

# Try to add an element (will raise an error)
frozen_set.add(4)
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In[34], line 5
      2 frozen_set = frozenset([1, 2, 3])
      4 # Try to add an element (will raise an error)
----> 5 frozen_set.add(4)

AttributeError: 'frozenset' object has no attribute 'add'
```

Python Dictionary:

In Python, a dictionary (often abbreviated as dict) is a built-in data structure used to store a collection of key-value pairs. Dictionaries are unordered (in Python versions before 3.7), mutable, and indexed by keys, which must be unique and immutable (e.g., strings, numbers, or tuples). Dictionaries are highly optimized for retrieving values based on their keys, making them one of the most efficient data structures for lookups.

Key Features of Python Dictionaries

1. **Key-Value Pairs**: Each element in a dictionary is a pair of a key and its corresponding value.
2. **Unordered**: Before Python 3.7, dictionaries were unordered. Starting from Python 3.7, dictionaries maintain insertion order.
3. **Mutable**: You can add, remove, or modify key-value pairs.
4. **Keys are Unique**: No two keys can be the same in a dictionary.
5. **Keys Must Be Immutable**: Keys can be strings, numbers, or tuples, but not lists or other mutable types.
6. **Values Can Be Anything**: Values can be of any data type, including lists, tuples, sets, or even other dictionaries.

Creating a Dictionary

You can create a dictionary by enclosing key-value pairs in curly braces {}, separated by commas. Each key-value pair is written as key: value.

```
# Dictionary with string keys
student = {"name": "Alice", "age": 25, "grade": "A"}

# Dictionary with integer keys
squares = {1: 1, 2: 4, 3: 9, 4: 16}

# Dictionary with mixed keys
mixed_dict = {"name": "Bob", 1: [1, 2, 3], (1, 2): "tuple key"}

# Empty dictionary
empty_dict = {}
```

Accessing Dictionary Elements

You can access the value associated with a key using square brackets [] or the get() method. If the key does not exist, using [] will raise a KeyError, while get() will return None (or a default value you specify).

```python
student = {"name": "Alice", "age": 25, "grade": "A"}

# Access value using key
print(student["name"])

# Access value using get()
print(student.get("age"))

# Access a non-existent key
print(student.get("address"))
print(student.get("address", "Not Found"))
```

```
Alice
25
None
Not Found
```

Modifying Dictionaries

Dictionaries are mutable, so you can add, update, or remove key-value pairs.

1. Adding or Updating Elements

- Use square brackets [ ] to add or update a key-value pair.

```python
student = {"name": "Alice", "age": 25}

# Add a new key-value pair
student["grade"] = "A"
print(student)

# Update an existing key
student["age"] = 26
print(student)
```

```
{'name': 'Alice', 'age': 25, 'grade': 'A'}
{'name': 'Alice', 'age': 26, 'grade': 'A'}
```

2. Removing Elements

- **pop()**: Removes a key-value pair and returns the value.
- **popitem()**: Removes and returns the last inserted key-value pair (Python 3.7+).
- **del**: Deletes a key-value pair.
- **clear()**: Removes all key-value pairs from the dictionary.

```python
student = {"name": "Alice", "age": 25, "grade": "A"}

# Remove a key-value pair
age = student.pop("age")
print(age)
print(student)

# Remove the last inserted key-value pair
last_item = student.popitem()
print(last_item)
print(student)

# Delete a key-value pair
del student["name"]
print(student)

# Clear the dictionary
student = {"name": "Alice", "age": 25}
student.clear()
print(student)
```

```
25
{'name': 'Alice', 'grade': 'A'}
('grade', 'A')
{'name': 'Alice'}
{}
{}
```

Common Dictionary Operations

1. Checking for Keys

- Use the in keyword to check if a key exists in the dictionary.

```
student = {"name": "Alice", "age": 25}

# Check if a key exists
print("name" in student)
print("grade" in student)
```

```
True
False
```

2. Iterating Over a Dictionary

- You can iterate over keys, values, or key-value pairs using keys(), values(), or items().

```
student = {"name": "Alice", "age": 25, "grade": "A"}

# Iterate over keys
for key in student.keys():
    print(key)

# Iterate over values
for value in student.values():
    print(value)

# Iterate over key-value pairs
for key, value in student.items():
    print(f"{key}: {value}")
```

```
name
age
grade
Alice
25
A
name: Alice
age: 25
grade: A
```

3. Dictionary Methods

- **len()**: Returns the number of key-value pairs.
- **copy()**: Returns a shallow copy of the dictionary.
- **update()**: Merges another dictionary into the current one.

```
student = {"name": "Alice", "age": 25}

# Length of the dictionary
print(len(student))

# Create a copy
student_copy = student.copy()
print(student_copy)

# Update the dictionary
student.update({"grade": "A", "age": 26})
print(student)
```

```
2
{'name': 'Alice', 'age': 25}
{'name': 'Alice', 'age': 26, 'grade': 'A'}
```

Nested Dictionaries

Dictionaries can contain other dictionaries, allowing you to create complex data structures.

```
students = {
    "Alice": {"age": 25, "grade": "A"},
    "Bob": {"age": 22, "grade": "B"},
    "Charlie": {"age": 23, "grade": "C"}
}

# Access nested dictionary
print(students["Alice"]["age"])
```

```
25
```

Dictionary Comprehension

Dictionary comprehension is a concise way to create dictionaries.

```
# Create a dictionary of squares
squares = {x: x ** 2 for x in range(5)}
print(squares)
```

```
{0: 0, 1: 1, 2: 4, 3: 9, 4: 16}
```

Common Use Cases for Dictionaries

Storing Data:

- Use dictionaries to store data in a structured way (e.g., user profiles, configurations).

Counting Occurrences:

- Use dictionaries to count the frequency of elements in a list.

Mapping Relationships:

- Use dictionaries to map relationships between entities (e.g., employee IDs to names).

Configuration Settings:

- Use dictionaries to store configuration settings for an application.

Python String:

In Python, a string is a sequence of characters enclosed in single quotes (' '), double quotes (" "), or triple quotes (''' ''' or """ """). Strings are immutable, meaning once a string is created, it cannot be modified. However, you can create new strings based on existing ones. Python provides a rich set of methods and operations for working with strings.

Key Features of Python Strings

1. **Immutable**: Once a string is created, it cannot be changed.
2. **Ordered**: Strings maintain the order of characters.
3. **Indexed**: You can access individual characters using their index.
4. **Iterable**: You can loop through each character in a string.
5. **Supports Unicode**: Python strings support Unicode, allowing you to work with text in any language.

Creating a String

You can create a string using single quotes, double quotes, or triple quotes (for multi-line strings).

```
# Single quotes
string1 = 'Hello, World!'

# Double quotes
string2 = "Python Programming"

# Triple quotes for multi-line strings
string3 = """This is a
multi-line string."""
```

Accessing String Characters

You can access individual characters in a string using their index. Python uses **zero-based indexing**, meaning the first character is at index 0. You can also use negative indexing to access characters from the end of the string.

```
text = "Python"

# Access the first character
print(text[0])

# Access the last character using negative indexing
print(text[-1])
```

```
P
n
```

String Slicing

You can extract a portion of a string using slicing. The syntax is string[start:stop:step].

```
text = "Python Programming"

# Get the first 6 characters
print(text[0:6])

# Get every second character
print(text[::2])

# Reverse the string
print(text[::-1])
```

```
Python
Pto rgamn
gnimmargorP nohtyP
```

Common String Operations

1. Concatenation

- Use the + operator to concatenate (combine) strings.

```
string1 = "Hello"
string2 = "World"
result = string1 + ", " + string2
print(result)
```

```
Hello, World
```

2. Repetition

- Use the * operator to repeat a string.

```
text = "Python"
repeated_text = text * 3
print(repeated_text)
```

```
PythonPythonPython
```

3. Membership Testing

- Use the in keyword to check if a substring exists in a string.

```
text = "Python Programming"
print("Python" in text)
print("Java" in text)
```

```
True
False
```

4. String Length

- Use the len() function to get the length of a string.

```
text = "Python"
print(len(text))
```

```
6
```

Common String Methods

1. Case Conversion

- **lower()**: Converts the string to lowercase.
- **upper()**: Converts the string to uppercase.
- **title()**: Converts the string to title case (first letter of each word capitalized).

```
text = "Python Programming"

# Convert to lowercase
print(text.lower())

# Convert to uppercase
print(text.upper())

# Convert to title case
print(text.title())
```

```
python programming
PYTHON PROGRAMMING
Python Programming
```

2. Stripping Whitespace

- **strip()**: Removes leading and trailing whitespace.
- **lstrip()**: Removes leading whitespace.
- **rstrip()**: Removes trailing whitespace.

```
text = "   Python Programming   "

# Remove leading and trailing whitespace
print(text.strip())

# Remove leading whitespace
print(text.lstrip())

# Remove trailing whitespace
print(text.rstrip())
```

```
Python Programming
Python Programming
   Python Programming
```

3. Splitting and Joining

- **split()**: Splits a string into a list of substrings based on a delimiter.
- **join()**: Joins a list of strings into a single string using a delimiter.

```
text = "Python,Java,C++"

# Split the string into a list
languages = text.split(",")
print(languages)

# Join the list into a single string
result = " | ".join(languages)
print(result)
```

```
['Python', 'Java', 'C++']
Python | Java | C++
```

4. Replacing Substrings

- **replace()**: Replaces occurrences of a substring with another substring.

```
text = "Python Programming"

# Replace "Python" with "Java"
new_text = text.replace("Python", "Java")
print(new_text)
```

```
Java Programming
```

5. Finding Substrings

- **find()**: Returns the index of the first occurrence of a substring (or -1 if not found).
- **index()**: Similar to find(), but raises an error if the substring is not found.

```
text = "Python Programming"

# Find the index of "Programming"
print(text.find("Programming"))

# Find the index of "Java" (not found)
print(text.find("Java"))
```

```
7
-1
```

6. Checking String Properties

- **isalpha()**: Returns True if all characters are alphabetic.
- **isdigit()**: Returns True if all characters are digits.
- **isalnum()**: Returns True if all characters are alphanumeric.
- **isspace()**: Returns True if all characters are whitespace.

```
text1 = "Python"
text2 = "123"
text3 = "Python123"
text4 = "    "

# Check if all characters are alphabetic
print(text1.isalpha())

# Check if all characters are digits
print(text2.isdigit())

# Check if all characters are alphanumeric
print(text3.isalnum())

# Check if all characters are whitespace
print(text4.isspace())
```

```
True
True
True
True
```

String Formatting

Python provides several ways to format strings:

1. **f-strings** (Python 3.6+): Embed expressions inside string literals.
2. **format() method**: Use placeholders {} to format strings.
3. **% operator**: Old-style string formatting.

```
name = "Alice"
age = 25

# f-strings
print(f"My name is {name} and I am {age} years old.")

# format() method
print("My name is {} and I am {} years old.".format(name, age))

# % operator
print("My name is %s and I am %d years old." % (name, age))
```

```
My name is Alice and I am 25 years old.
My name is Alice and I am 25 years old.
My name is Alice and I am 25 years old.
```

Escape Sequences

Escape sequences are used to include special characters in strings.

Escape Sequence	Description
\n	**Newline**
\t	**Tab**
****	**Backslash**
\"	**Double quote**
\'	**Single quote**

```
text = "Hello,\nWorld!"
print(text)
```

```
Hello,
World!
```

Common Use Cases for Strings:

Text Processing:

- Use strings for tasks like searching, replacing, and formatting text.

Data Validation:

- Use string methods to validate user input.

File Handling:

- Use strings to read from and write to files.

String Manipulation:

- Use slicing and methods to manipulate strings.

Python Functions:

In Python, a **function** is a block of reusable code that performs a specific task. Functions help in organizing code, making it more modular, and avoiding repetition. They are defined using the def keyword, followed by the function name, parameters (if any), and a block of code. Functions can return values using the return statement.

Key Concepts of Python Functions

1. **Function Definition**:
    - Use the def keyword to define a function.

Syntax:

def function_name(parameters):

  # Function body

  return value # Optional

2. **Function Call**:

- After defining a function, you can call it by using its name followed by parentheses.
    - function_name(arguments)

3. **Parameters and Arguments**:
    - **Parameters**: Variables listed inside the parentheses in the function definition.
    - **Arguments**: Values passed to the function when it is called.
4. **Return Statement**:
    - The return statement is used to send a value back to the caller. If no return statement is used, the function returns None.
5. **Scope**:
    - Variables defined inside a function are local to that function and cannot be accessed outside it.

Examples of Python Functions

1. Simple Function

A function that prints a message:

```
def greet():
    print("Hello, World!")

# Call the function
greet()
```

```
Hello, World!
```

2. Function with Parameters

A function that takes two numbers as input and prints their sum:

```
def add_numbers(a, b):
    print(f"The sum of {a} and {b} is {a + b}")

# Call the function
add_numbers(5, 3)
```

```
The sum of 5 and 3 is 8
```

3. Function with Return Value

A function that calculates the square of a number and returns the result:

```
def square(number):
    return number ** 2

# Call the function and store the result
result = square(4)
print(f"The square of 4 is {result}")
```

```
The square of 4 is 16
```

4. Function with Default Arguments

A function with default values for parameters:

```
def greet_user(name="Guest"):
    print(f"Hello, {name}!")

# Call the function without arguments
greet_user()  # Uses the default value
# Call the function with an argument
greet_user("Alice")
```

```
Hello, Guest!
Hello, Alice!
```

5. Function with Variable-Length Arguments

A function that can accept any number of arguments using *args:

```
def sum_all(*args):
    total = 0
    for num in args:
        total += num
    return total

# Call the function with multiple arguments
result = sum_all(1, 2, 3, 4, 5)
print(f"The sum is {result}")
```

```
The sum is 15
```

6. Function with Keyword Arguments

A function that accepts keyword arguments using **kwargs:

```python
def display_info(**kwargs):
    for key, value in kwargs.items():
        print(f"{key}: {value}")

# Call the function with keyword arguments
display_info(name="Alice", age=30, city="New York")
```

```
name: Alice
age: 30
city: New York
```

Best Practices for Writing Functions

1. **Descriptive Names**:
    - Use meaningful names for functions and parameters.
    - Example: calculate_area() is better than func1().
2. **Single Responsibility**:
    - Each function should perform a single task.
3. **Docstrings**:
    - Add a docstring to describe what the function does.
    - Example:

def add(a, b):

"""

Adds two numbers and returns the result.

"""

return a + b

4. **Avoid Global Variables**:
    - Use parameters and return values instead of modifying global variables.
5. **Keep Functions Short**:
    - If a function is too long, consider breaking it into smaller functions.

Types of Functions:

1. Built-in Functions

These are pre-defined functions provided by Python. You can use them directly without needing to define them.

```
# len() - Returns the length of an object
print(len("Hello"))

# type() - Returns the type of an object
print(type(42))

# sum() - Returns the sum of all elements in an iterable
print(sum([1, 2, 3, 4]))

# max() - Returns the largest item in an iterable
print(max([10, 20, 30]))
```

```
5
<class 'int'>
10
30
```

2. User-Defined Functions

These are functions defined by the user to perform specific tasks. They are created using the def keyword.

```
# Define a function to add two numbers
def add(a, b):
    return a + b

# Call the function
result = add(5, 3)
print(result)
```

```
8
```

Python Lambda:

In Python, a **lambda function** is a small, anonymous function defined using the lambda keyword. Lambda functions are also known as **anonymous functions** because they are not declared with the standard def keyword. They are typically used for short, simple operations and can have any number of arguments but only one expression.

Key Features of Lambda Functions

1. **Anonymous**: Lambda functions do not have a name.
2. **Single Expression**: They can only contain one expression, which is evaluated and returned.
3. **Concise**: Lambda functions are often used for short, simple operations.
4. **Inline**: They can be used inline, meaning they can be defined and used in the same place.

Syntax of Lambda Functions

lambda arguments: expression

- **arguments**: The input parameters (can be zero or more).
- **expression**: A single expression that is evaluated and returned.

Examples of Lambda Functions

1. Simple Lambda Function

A lambda function that adds two numbers:

```
add = lambda x, y: x + y
print(add(5, 3))
```

```
8
```

2. Lambda Function with No Arguments

A lambda function that always returns the same value:

```
greet = lambda: "Hello, World!"
print(greet())
```

```
Hello, World!
```

3. Lambda Function with One Argument

A lambda function that calculates the square of a number:

```
square = lambda x: x ** 2
print(square(5))
```

```
25
```

4. Lambda Function with Multiple Arguments

A lambda function that multiplies three numbers:

```
multiply = lambda x, y, z: x * y * z
print(multiply(2, 3, 4))
```

```
24
```

5. Lambda Function with Conditional Logic

A lambda function that checks if a number is even:

```
is_even = lambda x: True if x % 2 == 0 else False
print(is_even(4))
print(is_even(5))
```

```
True
False
```

6. Lambda Function with Default Arguments

A lambda function with default arguments:

```
greet = lambda name="Guest": f"Hello, {name}!"
print(greet())
print(greet("Alice"))
```

```
Hello, Guest!
Hello, Alice!
```

Common Use Cases for Lambda Functions

1. Sorting with Lambda

Lambda functions are often used with sorting functions like sorted() or list.sort() to define custom sorting logic.

Example: Sort a list of tuples by the second element

```
data = [(1, 3), (4, 1), (2, 2)]
sorted_data = sorted(data, key=lambda x: x[1])
print(sorted_data)
```

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

2. Filtering with Lambda

Lambda functions are commonly used with the filter() function to filter elements from a list.

Example: Filter even numbers from a list

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

```
[2, 4, 6]
```

3. Mapping with Lambda

Lambda functions are often used with the map() function to apply a transformation to each element in a list.

Example: Square each number in a list

```
numbers = [1, 2, 3, 4]
squared_numbers = list(map(lambda x: x ** 2, numbers))
print(squared_numbers)
```

```
[1, 4, 9, 16]
```

4. Inline Use

Lambda functions can be used inline, meaning they are defined and used in the same place.

Example: Use a lambda function inline in a function call

```
result = (lambda x, y: x + y)(5, 3)
print(result)
```

```
8
```

Advantages of Lambda Functions

1. **Conciseness**: Lambda functions are short and concise, making them ideal for simple operations.
2. **Readability**: When used appropriately, lambda functions can make code more readable by keeping it compact.
3. **Inline Use**: They can be defined and used in the same place, reducing the need for separate function definitions.

Limitations of Lambda Functions

1. **Single Expression**: Lambda functions can only contain one expression, making them unsuitable for complex logic.
2. **No Statements**: They cannot include statements like if, for, or while directly (though conditional expressions like x if condition else y are allowed).
3. **Readability**: Overusing lambda functions can make code harder to read and understand.

When to Use Lambda Functions

- Use lambda functions for short, simple operations that are used only once or in a limited scope.
- Avoid using lambda functions for complex logic or when the function needs to be reused multiple times.

Example: Combining Lambda Functions

Here's an example of using lambda functions with map() and filter():

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

# Filter even numbers and square them
result = list(map(lambda x: x ** 2, filter(lambda x: x % 2 == 0, numbers)))
print(result)
```

```
[4, 16, 36]
```

Python Module:

In Python, a **module** is a file containing Python code (e.g., functions, classes, variables) that can be imported and used in other Python programs. Modules help in organizing code, making it reusable, and improving maintainability. Python comes with a large standard library of modules, and you can also create your own custom modules.

Key Features of Python Modules

1. **Reusability**: Modules allow you to reuse code across multiple programs.
2. **Namespace Management**: Modules help avoid naming conflicts by organizing code into separate namespaces.
3. **Standard Library**: Python provides a rich set of built-in modules for common tasks.
4. **Custom Modules**: You can create your own modules to organize your code.

Types of Modules

1. **Built-in Modules**: Pre-installed modules that come with Python (e.g., math, os, random).
2. **Custom Modules**: Modules created by the user.

Using Built-in Modules

To use a built-in module, you need to import it using the import statement.

Example: Using the math Module

```
# Import the math module
import math

# Use functions from the math module
print(math.sqrt(16))
print(math.pi)
```

```
4.0
3.141592653589793
```

Creating Custom Modules

You can create your own module by saving Python code in a .py file. The filename (without the .py extension) becomes the module name.

Example: Creating a Custom Module

1. Create a file named mymodule.py:

mymodule.py - C:/Users/Amol Thakare/Downloads/mymodule.py (3.13.1)

File Edit Format Run Options Window Help

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

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

2. Import and use the module in another Python file:

main.py - C:/Users/Amol Thakare/Downloads/main.py (3.13.1)

File Edit Format Run Options Window Help

```
import mymodule

# Use functions from mymodule
print(mymodule.greet("Alice"))
print(mymodule.add(5, 3))
```

Importing Specific Functions from a Module

You can import specific functions or variables from a module using the from ... import ... syntax.

Example:

main.py - C:/Users/Amol Thakare/Downloads/main.py (3.13.1)

File Edit Format Run Options Window Help

```
# Import only the greet function from mymodule
from mymodule import greet

# Use the greet function
print(greet("Bob"))
```

IDLE Shell 3.13.1

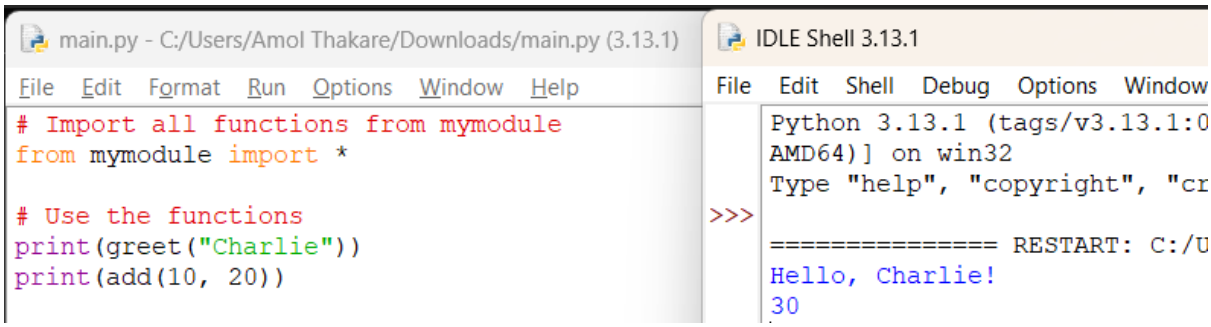
File Edit Shell Debug Options Window Help

```
    Python 3.13.1 (tags/v3.13.1:0671451, Dec
    AMD64)] on win32
    Type "help", "copyright", "credits" or "
>>>
    ========== RESTART: C:/Users/Amol T
    Hello, Bob!
>>>
```

Importing All Functions from a Module

You can import all functions and variables from a module using the from ... import * syntax. However, this is generally discouraged because it can lead to naming conflicts.

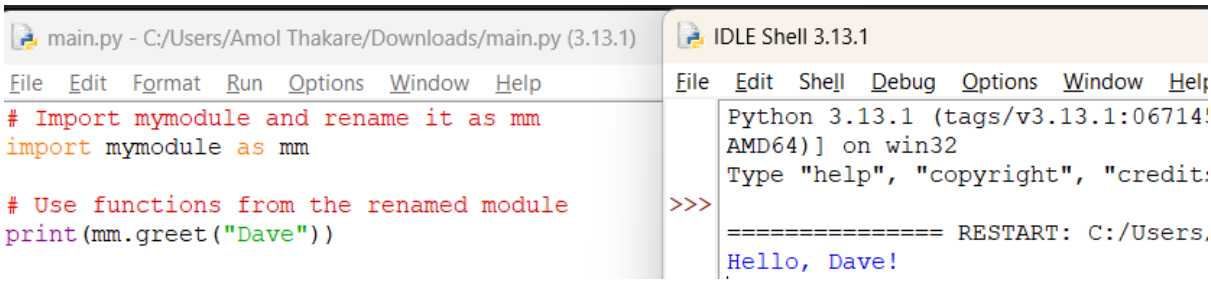
Example:



Renaming Modules or Functions

You can rename a module or function when importing it using the as keyword.

Example:



Standard Library Modules

Python's standard library includes many useful modules. Here are a few examples:

1. random Module

Used for generating random numbers and making random choices.

Example:

```python
import random

# Generate a random integer between 1 and 10
print(random.randint(1, 10))

# Choose a random element from a list
fruits = ["apple", "banana", "cherry"]
print(random.choice(fruits))
```

```
1
banana
```

2. os Module

Used for interacting with the operating system.

Example:

```
import os

# Get the current working directory
print(os.getcwd())

# List files in the current directory
print(os.listdir())
```

```
C:\Users\Amol Thakare
['.anaconda', '.conda', '.condarc', '.continuum', '.idlerc', '.ipynb_checkpoints', '.ipython', '.jupyter', '.vscode', 'anaconda3', 'AppData', 'Application Data', 'Contacts', 'Cookies', 'Documents', 'Downloads', 'Favorites', 'Links', 'Local Settings', 'Music', 'My Documents', 'mymodule.py', 'NetHood', 'NTUSER.DAT', 'ntuser.dat.LOG1', 'ntuser.dat.LOG2', 'NTUSER.DAT{d5f681f3-837d-11ef-8b4a-96bb43965969}.TM.blf', 'NTUSER.DAT{d5f681f3-837d-11ef-8b4a-96bb43965969}.TMContainer00000000000000000001.regtrans-ms', 'NTUSER.DAT{d5f681f3-837d-11ef-8b4a-96bb43965969}.TMContainer00000000000000000002.regtrans-ms', 'ntuser.ini', 'OneDrive', 'Pictures', 'PrintHood', 'Python Week 1.ipynb', 'Python Week 2.ipynb', 'Recent', 'Saved Games', 'Searches', 'SendTo', 'Start Menu', 'Templates', 'Untitled.ipynb', 'Videos', '__pycache__']
```

3. datetime Module

Used for working with dates and times.

Example:

```
from datetime import datetime

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

```
2025-03-09 12:24:59.695764
```

4. math Module

Used for mathematical operations.

Example:

```
import math

# Calculate the square root of a number
print(math.sqrt(25))

# Calculate the factorial of a number
print(math.factorial(5))
```

```
5.0
120
```

Module Search Path

When you import a module, Python searches for it in the following locations:

1. The current directory.
2. Directories listed in the PYTHONPATH environment variable.
3. The installation-dependent default directory (e.g., Python's standard library).

You can view the search path using the sys module:

```
import sys
print(sys.path)
```

```
['C:\\Users\\Amol Thakare', 'C:\\Users\\Amol Thakare\\anaconda3\\python312.zip', 'C:\\Users\\Amol Thakare\\anaconda3\\DLLs', 'C:\\Users\\Amol Thakare\\an
aconda3\\Lib', 'C:\\Users\\Amol Thakare\\anaconda3', '', 'C:\\Users\\Amol Thakare\\anaconda3\\Lib\\site-packages', 'C:\\Users\\Amol Thakare\\anaconda3\\L
ib\\site-packages\\win32', 'C:\\Users\\Amol Thakare\\anaconda3\\Lib\\site-packages\\win32\\lib', 'C:\\Users\\Amol Thakare\\anaconda3\\Lib\\site-packages
\\Pythonwin', 'C:\\Users\\Amol Thakare\\anaconda3\\Lib\\site-packages\\setuptools\\_vendor']
```

Python File Handling:

In Python, **file handling** refers to the process of working with files on your computer. This includes creating, reading, writing, and modifying files. Python provides built-in functions and methods to handle files easily. Files can be of different types, such as text files, binary files, CSV files, etc.

Key Concepts of File Handling

1. **Opening a File**: Use the open() function to open a file.
2. **Reading from a File**: Use methods like read(), readline(), or readlines() to read data from a file.
3. **Writing to a File**: Use methods like write() or writelines() to write data to a file.
4. **Closing a File**: Use the close() method to close a file after you're done with it.
5. **File Modes**: Specify the mode in which you want to open the file (e.g., read, write, append).

File Modes

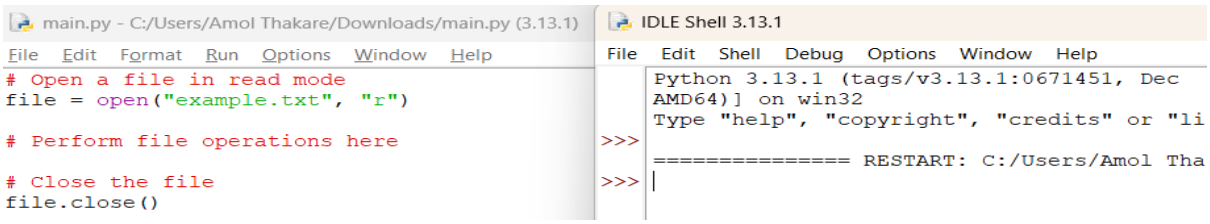
When opening a file, you need to specify the mode. Here are the most common modes:

Mode	Description
'r'	**Read mode**: Opens the file for reading (default mode).
'w'	**Write mode**: Opens the file for writing. Creates a new file if it doesn't exist, or overwrites the existing file.
'a'	**Append mode**: Opens the file for appending. Creates a new file if it doesn't exist.
'b'	**Binary mode**: Opens the file in binary mode (e.g., 'rb', 'wb').
'x'	**Exclusive creation mode**: Opens the file for writing only if it doesn't exist.
'+'	**Update mode**: Opens the file for both reading and writing (e.g., 'r+').

Opening and Closing a File

To open a file, use the open() function. Always close the file using the close() method when you're done.

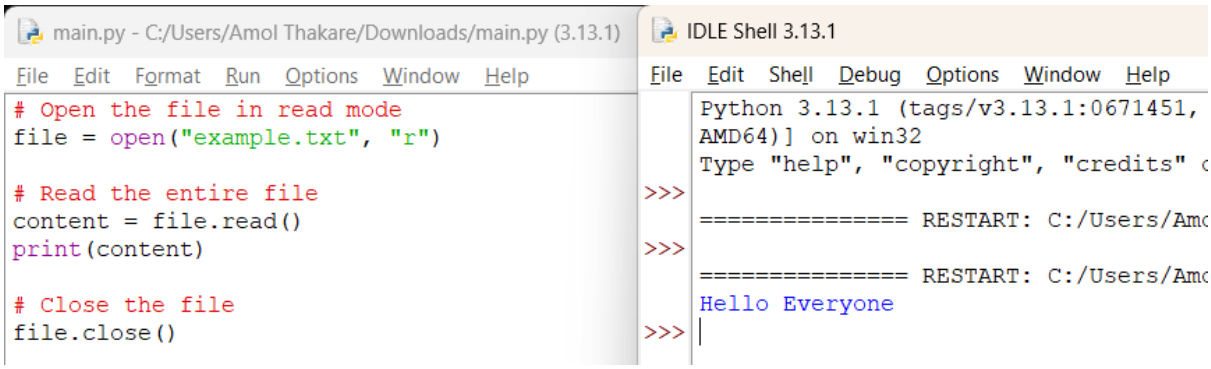
Example:



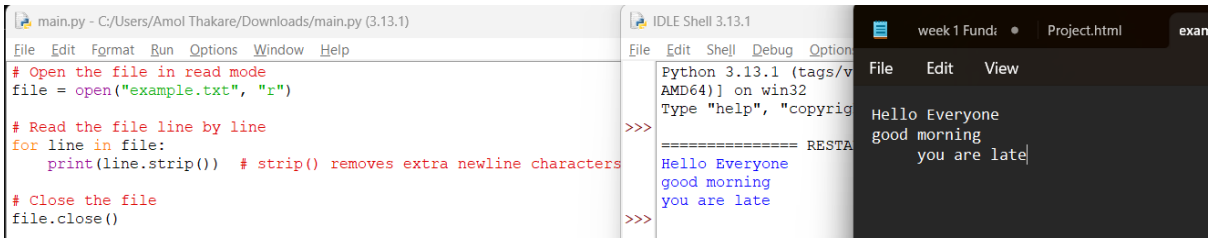
Reading from a File

You can read the contents of a file using methods like read(), readline(), or readlines().

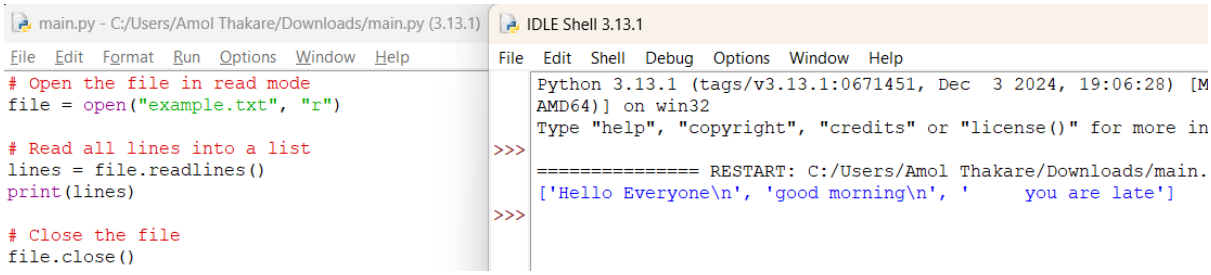
Example 1: Reading the Entire File



Example 2: Reading Line by Line



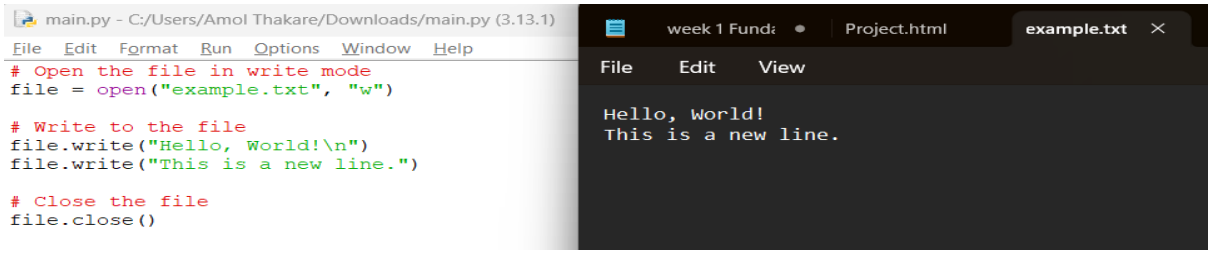
Example 3: Reading All Lines into a List



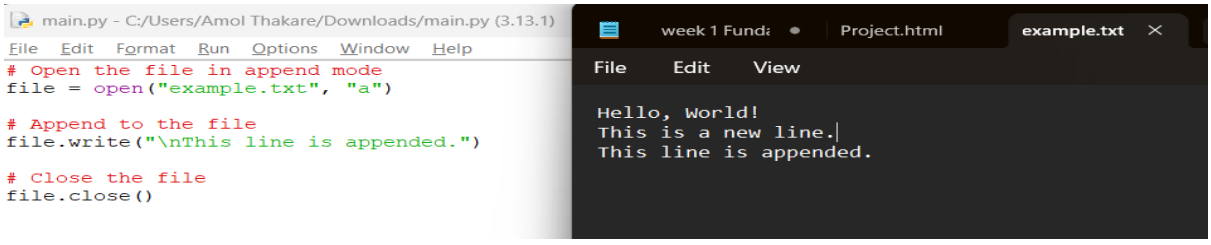
Writing to a File

You can write to a file using the write() or writelines() methods.

Example 1: Writing to a File



Example 2: Appending to a File



Python Exception Handling:

In Python, **exception handling** is a mechanism to handle runtime errors gracefully. When an error occurs during the execution of a program, Python raises an **exception**. If the exception is not handled, the program will crash. Exception handling allows you to catch and handle these errors, ensuring that your program can continue running or terminate gracefully.

Key Concepts of Exception Handling

1. **Try-Except Block**: The primary way to handle exceptions in Python.
2. **Finally Block**: Used to execute code regardless of whether an exception occurs.
3. **Else Block**: Executes if no exception occurs in the try block.
4. **Raising Exceptions**: You can manually raise exceptions using the raise keyword.
5. **Custom Exceptions**: You can define your own exception classes.

Basic Exception Handling with try-except

The try block contains the code that might raise an exception. The except block contains the code to handle the exception.

Example:

```
try:
    # Code that might raise an exception
    result = 10 / 0
except ZeroDivisionError:
    # Handle the exception
    print("Error: Division by zero is not allowed.")
```

```
Error: Division by zero is not allowed.
```

Handling Multiple Exceptions

You can handle multiple exceptions by specifying multiple except blocks.

Example:

```
try:
    # Code that might raise an exception
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
```

```
Enter a number:  0
Error: Division by zero is not allowed.
```

```
try:
    # Code that might raise an exception
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
```

```
Enter a number:  abc
Error: Invalid input. Please enter a valid number.
```

Using else Block

The else block is executed if no exception occurs in the try block.

Example:

```
try:
    num = int(input("Enter a number: "))
    result = 10 / num
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
except ValueError:
    print("Error: Invalid input. Please enter a valid number.")
else:
    print(f"The result is {result}")
```

```
Enter a number:  6
The result is 1.6666666666666667
```

Using finally Block

The finally block is executed regardless of whether an exception occurs. It is typically used for cleanup actions (e.g., closing files).

Example:

```
try:
    file = open("example.txt", "r")
    content = file.read()
    print(content)
except FileNotFoundError:
    print("Error: File not found.")
finally:
    # This block will always execute
    print("Closing the file.")
    file.close()
```

```
Hello, World!
This is a new line.
Closing the file.
```

Raising Exceptions

You can manually raise exceptions using the raise keyword. This is useful for enforcing certain conditions in your code.

Example:

```
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative.")
    elif age < 18:
        print("You are a minor.")
    else:
        print("You are an adult.")

try:
    check_age(-5)
except ValueError as e:
    print(f"Error: {e}")
```

```
Error: Age cannot be negative.
```

Custom Exceptions

You can define your own exception classes by inheriting from Python's built-in Exception class.

Example:

```
# Define a custom exception
class NegativeAgeError(Exception):
    pass

def check_age(age):
    if age < 0:
        raise NegativeAgeError("Age cannot be negative.")
    elif age < 18:
        print("You are a minor.")
    else:
        print("You are an adult.")

try:
    check_age(-5)
except NegativeAgeError as e:
    print(f"Error: {e}")
```

```
Error: Age cannot be negative.
```

Handling All Exceptions

You can use a generic except block to catch all exceptions. However, this is generally discouraged because it can hide unexpected errors.

Example:

```
try:
    result = 10 / 0
except Exception as e:
    print(f"An error occurred: {e}")
```

```
An error occurred: division by zero
```