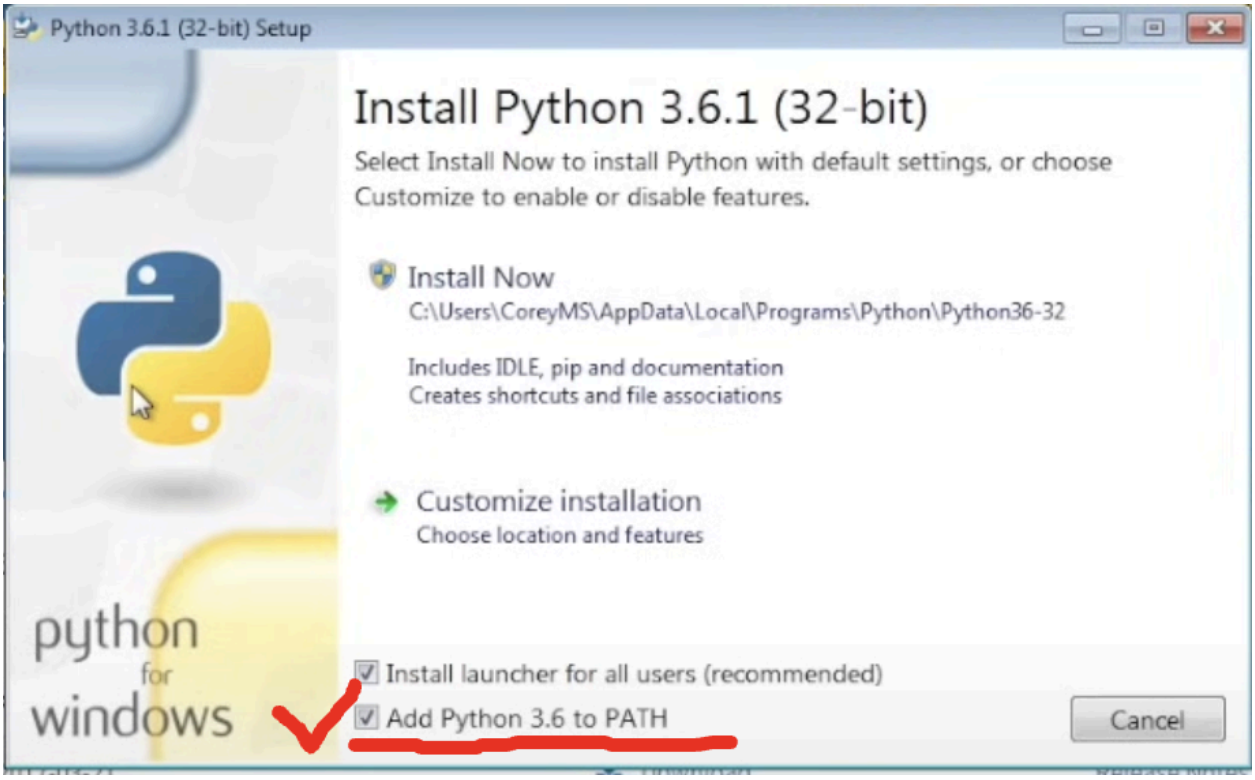
Welcome to the Python World

Master Class 1: Python Fundamentals

Download and Insatll Python:

https://www.python.org/downloads/

Windows Users (be careful to check this during installation):



Check Python Version:

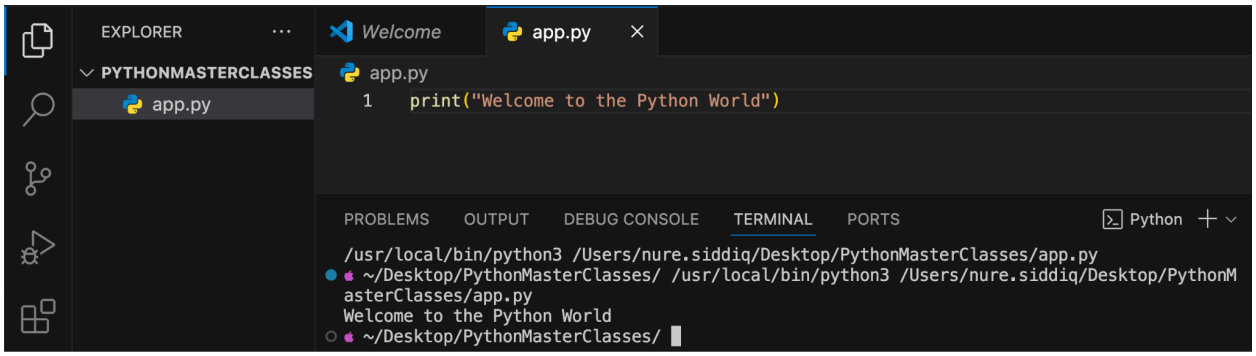
python --version or python3 --version or python -V

bongoDev Python Play List:

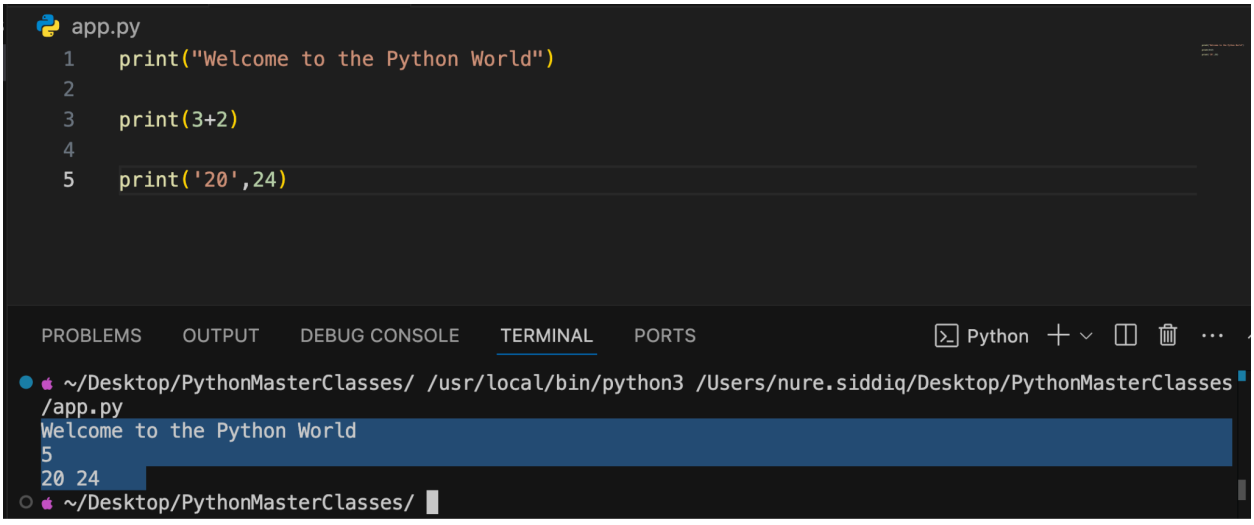
https://www.youtube.com/watch?v=mjYXQxDSQds&list=PL4VsqV8BmJN-hfUap967qOtfIDM3gdufY

Enable Python in VS Code:

First Python Program:



Python RUNS Line by Line (Top to bottom)



Data Types:

```
app.py
1   print(type("Welcome to the Python World"))
2
3   print(type(3))
4
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

~/Desktop/PythonMasterClasses/ /usr/local/bin/python3 /Users/
/app.py
<class 'str'>
<class 'int'>
~/Desktop/PythonMasterClasses/
```

Variable:

Is a holder of data that holds/store a value and we can use multiple times
name = 'Nure Siddiq' (Is a Python statement)
name is variable name
= is an assignment operator which assigns value 'Nure Siddiq' to variable name
'Nure Siddiq' is the value

```
app.py > ...
1   name = 'Nure Siddiq'
2
3   print(name)
4
5   age = 30
6   print(age)
7
PROBLEMS  OUTPUT  DEBUG CONSOLE  T

~/Desktop/PythonMasterClasses/ /usr/lo
/app.py
Nure Siddiq
30
~/Desktop/PythonMasterClasses/
```

Important Features in Python:

- [ ] High level programming language (Like plain English)
- [ ] Dynamically typed language (No need to define data type)
- [ ] Huge popular in task automation (Scripting language)
- [ ] Python is both interpreted and compiled
- [ ] Case sensitive language (Nure and nure are different)

Operators In Python:

1. Arithmetic Operators

- `+` : Addition
- `-` : Subtraction
- `*` : Multiplication
- `/` : Division (float division)
- `//` : Floor Division
- `%` : Modulus
- `**` : Exponentiation

2. Comparison Operators

- `==` : Equal to
- `!=` : Not equal to
- `>` : Greater than
- `<` : Less than
- `>=` : Greater than or equal to
- `<=` : Less than or equal to

```
# Variables for comparison
a = 10
b = 5
c = 10

# 1. Equal to (==)
print(a == c)   # True, because 10 is equal to 10

# 2. Not equal to (!=)
print(a != b)   # True, because 10 is not equal to 5

# 3. Greater than (>)
print(a > b)    # True, because 10 is greater than 5

# 4. Less than (<)
print(b < a)    # True, because 5 is less than 10

# 5. Greater than or equal to (>=)
print(a >= c)   # True, because 10 is equal to 10

# 6. Less than or equal to (<=)
print(b <= c)   # True, because 5 is less than or equal to 10
```

3. Logical Operators

- `and` : Returns `True` if both operands are true
- `or` : Returns `True` if at least one operand is true
- `not` : Returns `True` if operand is false

```
# Variables for logical operations
x = True
y = False
a = 10
b = 5

# 1. 'and' operator
# Returns True only if both conditions are True
print(x and (a > b))   # True, because both x is True and 10 > 5

print(y and (a > b))   # False, because y is False

# 2. 'or' operator
# Returns True if at least one condition is True
print(x or y)          # True, because x is True

print(y or (b > a))    # False, because both y is False and 5 is not greater than 10

# 3. 'not' operator
# Returns True if operand is False
print(not x)           # False, because x is True

print(not y)           # True, because y is False
```

4. Membership Operators

- `in` : Returns `True` if a value is found in a sequence
- `not in` : Returns `True` if a value is not found in a sequence

```
# Example sequence (list)

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

# 1. 'in' operator
# Returns True if the element is in the list
print("banana" in fruits)      # True, because "banana" is in the list

print("grape" in fruits)       # False, because "grape" is not in the list

# 2. 'not in' operator
# Returns True if the element is not in the list
print("grape" not in fruits)   # True, because "grape" is not in the list

print("apple" not in fruits)   # False, because "apple" is in the list
```

5. Identity Operators

- `is` : Returns `True` if both variables point to the same object
- `is not` : Returns `True` if both variables do not point to the same object

```
# Variables for identity operations
a = [1, 2, 3]
b = a          # 'b' is assigned to the same object as 'a'
c = [1, 2, 3]  # 'c' is a different object with the same content as 'a'

# 1. 'is' operator
# Returns True if both variables point to the same object
print(a is b)       # True, because 'b' is assigned to the same object as 'a'

print(a is c)       # False, because 'a' and 'c' have the same content but are different objects

# 2. 'is not' operator
# Returns True if both variables do not point to the same object
print(a is not c)   # True, because 'a' and 'c' are different objects

print(a is not b)   # False, because 'a' and 'b' point to the same object
```

User Input: (Always STRING - str data type)

```python
num_1 = input("Type first number: ") #user input is always STRING (str) data type

num_2 = input("Type second number: ") #user input is always STRING (str) data type

print(num_1 + num_2)    #sum
print(num_1 - num_2)    #Subtract
print(num_1 * num_2)    #Multiply
print(num_1 / num_2)    #Devide
print(num_1 % num_2)    #Reminder (mod / modulus)
print(num_1 ** num_2)   #Power
```

```
~/Desktop/PythonMasterClasses/ /usr/local/bin/python3 /Users/nure.siddiq/Desktop/PythonMasterClasses
/app.py
Type first number: 5
Type second number: 3
53
Traceback (most recent call last):
  File "/Users/nure.siddiq/Desktop/PythonMasterClasses/app.py", line 6, in <module>
    print(num_1 - num_2)    #Subtract
          ~~~~~~^~~~~~
TypeError: unsupported operand type(s) for -: 'str' and 'str'
~/Desktop/PythonMasterClasses/
```

Type Casting: (Convert one data type to another)

Needed to convert user input **str** to **int**

```python
num_2 = int(input("Type second number: ")) #Converted string to integer

print(num_1 + num_2)    #sum
print(num_1 - num_2)    #Subtract
print(num_1 * num_2)    #Multiply
print(num_1 / num_2)    #Devide
print(num_1 % num_2)    #Reminder (mod / modulus)
print(num_1 ** num_2)   #Power
```

```
~/Desktop/PythonMasterClasses/ /usr/local/bin/python3 /Users/nure.siddiq/Desktop/PythonMasterC
/app.py
Type first number: 5
Type second number: 3
8
2
15
1.6666666666666667
2
125
~/Desktop/PythonMasterClasses/
```

Data Types In Detail:

```
13  print(type('Hello Nure'))
14  print(type(25))
15  print(type(12.07))
16  print(type(True))
17
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

~/Desktop/PythonMasterClasses/ /usr/local/bin/python3 /Us
/app.py
~/Desktop/PythonMasterClasses/ /usr/local/bin/python3 /Us
/app.py
<class 'str'>
<class 'int'>
<class 'float'>
<class 'bool'>
~/Desktop/PythonMasterClasses/
```

- [ ] String (str): Textual data

```
'Hello'or "Hello" or """Hello"""

"""
Line 1
line 2
line 3

"""
```

- [ ] Integer (int) - number 3, 4, 5

- [ ] Float - decimal number

```
>>> type(13.123)
<class 'float'>
```

- [ ] Boolean (True, False)

```
>>> type(True)
<class 'bool'>
>>> type(False)
<class 'bool'>
```

```
>>> 5 > 3
True
>>> 13 < 5
False
>>>
```

Conditions / logics in Python:

In Python, conditions are used to perform different actions based on whether a condition is `True` or `False`. Python commonly uses `if`, `elif`, and `else` statements for conditional checks.

Example: if and else:

```
temperature = 30

if temperature > 25:
    print("It's a warm day.")
else:
    print("It's a cool day.")
```

Output: It's a warm day.

Example: if, elif, and else:

```
score = 85

if score >= 90:
    print("Excellent!")

elif score >= 70:
    print("Good job!")

else:
    print("You can improve.")
```

Solve the Problems:

Problem1:

Write a Python program that takes a string input from the user and checks if Enjoy available in the input string:

"Hello, world! Hello everyone. Welcome to the world of Python. Enjoy coding in Python."

Problem2:

Write a Python program that takes an input of average marks from the user and then categorizes the grade as follows:

- If marks are greater than or equal to 90, the grade is "A+."
- If marks are less than 90 but greater than or equal to 70, the grade is "A-."
- If marks are less than 70 but greater than or equal to 50, the grade is "B."
- If marks are less than 50, the grade is "Fail."

Problem3:

Write a Python program that takes a single integer `n` as input from the user. The program should output:

- `"Fizz"` if `n` is a multiple of 3.
- `"Buzz"` if `n` is a multiple of 5.
- `"FizzBuzz"` if `n` is a multiple of both 3 and 5.
- Otherwise, output not a FizzBuzz number.

Problem4:

Write a calculator program that takes three inputs from the user:

1. **Input1**: A number (float or integer).
2. **Input2**: A number (float or integer).
3. **Operator**: A character representing a mathematical operation. The operator can be one of the following: `+`, `-`, `*`, `/`, or `%`.

The program should perform the following tasks:

- Validate the inputs to ensure that `Input1` and `Input2` are valid numbers and that the `Operator` is one of the specified characters.
- Use conditional statements to determine which operation to perform based on the `Operator` provided.
- If the operator is `+`, return the sum of `Input1` and `Input2`.
- If the operator is `-`, return the difference between `Input1` and `Input2`.
- If the operator is `*`, return the product of `Input1` and `Input2`.
- If the operator is `/`, return the quotient of `Input1` divided by `Input2`. If `Input2` is zero, return an appropriate error message indicating that division by zero is not allowed.
- If the operator is `%`, return the remainder of `Input1` divided by `Input2`.
- Display the result of the operation to the user.

Example Input/Output:

1. Input: `Input1 = 10`, `Input2 = 5`, `Operator = +`
   Output: `The result is 15`
2. Input: `Input1 = 10`, `Input2 = 0`, `Operator = /`
   Output: `Error: Division by zero is not allowed.`

Master Class 2: Data Types Details

Python has several built-in data types, each designed to store different kinds of data. Here's a comprehensive list with examples for each:

1. Numeric Types

- **Integer (`int`)**: Stores whole numbers (positive or negative, without a decimal).

```
x = 10
y = -5
```

- **Float (`float`)**: Stores decimal or floating-point numbers.

```
pi = 3.14159
temperature = -7.5
```

- **Complex (`complex`)**: Represents complex numbers, with a real and imaginary part.

```
z = 2 + 3j
```

2. String (`str`)

- **Definition**: A sequence of characters used to represent text.
- **Syntax**: Enclosed in single, double, or triple quotes.
- **Example**:

```
name = "Alice"
greeting = 'Hello, World!'
multiline = """This is a
multiline string"""
```

3. Boolean (`bool`)

- **Definition**: Represents truth values: `True` or `False`.
- **Example**:

```
is_active = True
is_closed = False
```

4. NoneType (`None`)

- **Definition**: Represents the absence of a value or a null value.
- **Example**:

```
result = None
```

5. Sequence Types

- **List (`list`)**: An ordered, mutable sequence of elements. Allows duplicate elements.

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

- **Tuple (`tuple`)**: An ordered, immutable sequence. Also allows duplicate elements.

```
colors = ("red", "green", "blue")
```

- **Range (`range`)**: Represents a sequence of numbers, typically used in loops.

```
numbers = range(1, 10)  # 1 to 9
```

6. Mapping Type

- **Dictionary (`dict`)**: Stores key-value pairs. Keys must be unique and immutable.

```
student = {"name": "Alice", "age": 21, "grade": "A"}
```

7. Set Types

- **Set (`set`)**: An unordered collection of unique, immutable elements.

```
unique_numbers = {1, 2, 3, 4}
```

- **Frozen Set (`frozenset`)**: An immutable version of a set.

```
vowels = frozenset({"a", "e", "i", "o", "u"})
```

Understand Mutable Vs Immutable:

Mutable vs. Immutable Data Types

In Python, mutability refers to whether or not the content of a data type can be changed after it has been created.

Mutable Data Types

- **Definition:** Mutable objects can be changed after their creation. You can modify, add, or remove elements without creating a new object.
- **Use Cases:** Useful when you need to alter data in place, such as updating a list of items, modifying a dictionary's contents, etc.

Immutable Data Types

- **Definition:** Immutable objects cannot be altered once created. Any modification results in the creation of a new object.
- **Use Cases:** Ideal for fixed data that shouldn't change, ensuring data integrity and facilitating safe usage in various contexts like keys in dictionaries.

Summary Table

Data Type	Mutable	Immutable
List	✅	❌
Dictionary	✅	❌
Set	✅	❌
Integer	❌	✅
Float	❌	✅
String	❌	✅
Tuple	❌	✅
Frozen Set	❌	✅

List, Dictionary and Set are the primary mutable built-in data types in Python.

Mutable Data Types in Python

1. **List**
   - **Description**: An ordered, mutable collection of items.
   - **Example**:

```
fruits = ["apple", "banana", "cherry"]
print(fruits)  # Output: ['apple', 'banana', 'cherry']

# Modifying the list
fruits.append("date")
print(fruits)  # Output: ['apple', 'banana', 'cherry', 'date']
```

2. **Dictionary (`dict`)**
   - **Description**: An unordered collection of key-value pairs.
   - **Example**:

```
student = {"name": "Alice", "age": 25}
print(student)  # Output: {'name': 'Alice', 'age': 25}

# Adding a new key-value pair
student["grade"] = "A"
print(student)  # Output: {'name': 'Alice', 'age': 25, 'grade': 'A'}
```

3. **Set (`set`)**
   - **Description**: An unordered collection of unique elements.
   - **Example**:

```
unique_numbers = {1, 2, 3}
print(unique_numbers)  # Output: {1, 2, 3}

# Adding a new element
unique_numbers.add(4)
print(unique_numbers)  # Output: {1, 2, 3, 4}
```

Immutable Data Types in Python

1. **Integer (`int`)**
   - **Description**: Represents whole numbers.
   - **Example**:

```
x = 10
print(x)  # Output: 10

# Attempting to change the value
x = x + 5
print(x)  # Output: 15
```

   Note: A new integer object is created when `x` is updated.

2. **Float (`float`)**
   - **Description**: Represents floating-point numbers.
   - **Example**:

```
pi = 3.14
print(pi)  # Output: 3.14

# Attempting to change the value
pi = pi + 0.0016
print(pi)  # Output: 3.1416
```

3. **String (`str`)**
   - **Description**: A sequence of characters.
   - **Example**:

```
greeting = "Hello"
print(greeting)  # Output: Hello

# Attempting to change a character (will raise an error)
# greeting[0] = "h"  # TypeError: 'str' object does not support itemassignmer

# Creating a new string
greeting = "h" + greeting[1:]
print(greeting)  # Output: hello
```

4. **Tuple (`tuple`)**
    - **Description**: An ordered, immutable collection of items.
    - **Example**:

```
colors = ("red", "green", "blue")
print(colors)  # Output: ('red', 'green', 'blue')

# Attempting to modify (will raise an error)
# colors[0] = "yellow"  # TypeError: 'tuple' object does not support item a
```

5. **Range (`range`)**
    - **Description**: Represents an immutable sequence of numbers.
    - **Example**:

python Copy code

```
numbers = range(1, 5)
print(list(numbers))  # Output: [1, 2, 3, 4]

# range objects are immutable; you cannot modify them.
```

Tuple, Integer, Float, String, Frozen Set, Bytes, and Range are the primary immutable built-in data types in Python.

Data Structures:

Python offers several built-in data structures to store and manage data effectively. A data structure is a way of organizing, storing, and managing data in a way that makes it efficient to access and modify. Data structures are fundamental for writing efficient programs, enabling you to handle, manipulate, and analyze data logically and systematically.

When to Use Which Data Structure?

- **List:** Use when you need a sequence of elements where duplicates are allowed and order matters.
- **Tuple:** Choose when you need an ordered, immutable collection, like coordinates.
- **Dictionary:** Ideal for key-value pairs where fast lookups by key are necessary.
- **Set:** Best for collections where duplicates need to be removed automatically.
- **Deque:** Useful when you need to add or remove items from both ends quickly.
- **Counter:** Perfect for frequency counting or tallying elements.
- **Heap:** Ideal for priority-based tasks, like scheduling or finding minimum/maximum values.

Data Types and their Operations:

String Data Operations:

Some essential string operations with code examples to demonstrate each one:

1. Length of a String

Use `len()` to get the number of characters in a string.

```
text = "Hello, World!"
length = len(text)
print("Length:", length)
```

2. String Slicing

Access parts of a string by slicing.

```
text = "Hello, World!"
first_five = text[:5]  # "Hello"
last_five = text[-5:]  # "orld!"
print("First five:", first_five)
print("Last five:", last_five)
```

3. Concatenation

Combine multiple strings using the `+` operator.

```
first = "Hello"
second = "World"
result = first + ", " + second + "!"
print("Concatenated:", result)
```

4. Changing Case

Convert a string to upper or lower case.

```
text = "Hello, World!"
print("Uppercase:", text.upper())
print("Lowercase:", text.lower())
```

5. Removing Whitespace

Trim leading and trailing whitespace with `strip()`.

```
text = "   Hello, World!   "
trimmed = text.strip()
print("Trimmed:", trimmed)
```

6. Finding a Substring

Use `find()` to locate a substring's position (returns -1 if not found).

```
text = "Hello, World!"
position = text.find("World")
print("Position of 'World':", position)
```

7. Replacing a Substring

Replace occurrences of a substring with `replace()`.

```
text = "Hello, World!"
new_text = text.replace("World", "Python")
print("Replaced text:", new_text)
```

8. Checking for a Substring

Use the `in` keyword to check if a substring exists.

```
text = "Hello, World!"
exists = "World" in text
print("Contains 'World':", exists)
```

9. Splitting a String

Split a string into a list based on a delimiter.

```
text = "apple,banana,cherry"
fruits = text.split(",")
print("Split list:", fruits)
```

10. Joining a List into a String

Combine a list of strings into one using `join()`.

```
words = ["Hello", "World"]
sentence = " ".join(words)
print("Joined sentence:", sentence)
```

11. String Formatting

Format strings with placeholders.

```
name = "Alice"
age = 25
intro = f"My name is {name} and I'm {age} years old."
print("Formatted string:", intro)
```

12. **String Reverse**

- **Code**:

```
s = "Python"
reversed_s = s[::-1]
print(reversed_s)  # Output: nohtyP
```

- **Explanation**: Using slicing `[::-1]`, you can reverse the string.

Solve these problems:

1. Count Vowels and Consonants

- **Problem**: Write a Python function that takes a string as input and counts the number of vowels and consonants in the string.
- **Example**:

```
Input: "Hello World"
Output: Vowels = 3, Consonants = 7
```

2. Palindrome Check

- **Problem**: Write a function that checks if a given string is a palindrome (reads the same forwards and backwards). Ignore spaces and capitalization.
- **Example**:

```
Input: "Racecar"
Output: True

Input: "Hello"
Output: False
```

3. Find the Longest Word

- **Problem**: Write a function that finds the longest word in a given sentence and returns it. If there are multiple words of the same length, return the first one.
- **Example**:

```
Input: "The quick brown fox jumps over the lazy dog"
Output: "jumps"
```

4. Remove Duplicates

- **Problem**: Write a function that takes a string and returns a new string with duplicate characters removed. Keep only the first occurrence of each character.
- **Example**:

```
Input: "programming"
Output: "progamin"
```

5. Count Words in a Sentence

- **Problem**: Write a function that takes a sentence as input and returns the count of each word. Assume that words are separated by spaces.
- **Example**:

```
Input: "to be or not to be"
Output: {"to": 2, "be": 2, "or": 1, "not": 1}
```

6: Extract the File Name from a URL

- **Problem**: Write a function that extracts and returns the file name from a given URL string. For example, given the URL `https://bongodev.com/images/pet_image.png`, the function should return `"pet_image.png"`.
- **Objective**: Practice working with string methods to manipulate and retrieve specific portions of a URL.

Example

```
Input: "https://bongodev.com/images/pet_image.png"
Output: "pet_image.png"
```

List Data Operations:

1. **Creating a List**
   - **Code**:

```
my_list = [1, 2, 3, 4, 5]
print(my_list)  # Output: [1, 2, 3, 4, 5]
```

   - **Explanation**: Lists can be created using square brackets `[]` and can contain items of different data types.

2. **Accessing Elements**
   - **Code**:

```
my_list = [1, 2, 3, 4, 5]
print(my_list[0])  # Output: 1
print(my_list[-1])  # Output: 5
```

   - **Explanation**: Elements can be accessed using their index, where indexing starts at 0. Negative indexing starts from the end of the list.

3. **Appending Elements**
   - **Code**:

```
my_list = [1, 2, 3]
my_list.append(4)
print(my_list)  # Output: [1, 2, 3, 4]
```

4. **Inserting Elements**
   - **Code**:

```
my_list = [1, 2, 3]
my_list.insert(1, 1.5)
print(my_list)  # Output: [1, 1.5, 2, 3]
```

   - **Explanation**: The `.insert()` method allows you to add an element at a specified index.

5. **Removing Elements**
   - **Code**:

```
my_list = [1, 2, 3, 4]
my_list.remove(3)
print(my_list)  # Output: [1, 2, 4]
```

   - **Explanation**: The `.remove()` method removes the first occurrence of a specified value from the list.
6. **Popping Elements**
   - **Code**:

```
python

my_list = [1, 2, 3, 4]
popped_element = my_list.pop()
print(popped_element)  # Output: 4
print(my_list)         # Output: [1, 2, 3]
```

   - **Explanation**: The `.pop()` method removes and returns the last item in the list. You can specify an index to remove an element from a specific position.
7. **Sorting a List**
   - **Code**:

```
my_list = [3, 1, 4, 2]
my_list.sort()
print(my_list)  # Output: [1, 2, 3, 4]
```

   - **Explanation**: The `.sort()` method sorts the list in ascending order.

8. **Reversing a List**
    - **Code**:

```
my_list = [1, 2, 3, 4]
my_list.reverse()
print(my_list)  # Output: [4, 3, 2, 1]
```

    - **Explanation**: The `.reverse()` method reverses the elements of the list in place.
9. **Finding the Length of a List**
    - **Code**:

```
my_list = [1, 2, 3, 4]
print(len(my_list))  # Output: 4
```

    - **Explanation**: The `len()` function returns the number of elements in the list.
10. **Counting Occurrences**
    - **Code**:

```
my_list = [1, 2, 2, 3, 2]
print(my_list.count(2))  # Output: 3
```

    - **Explanation**: The `.count()` method counts the occurrences of a specified value in the list.
11. **Finding the Index of an Element**
    - **Code**:

```
my_list = [1, 2, 3, 4]
print(my_list.index(3))  # Output: 2
```

    - **Explanation**: The `.index()` method returns the index of the first occurrence of a specified value.
12. **Extending a List**
    - **Code**:

```
my_list = [1, 2]
my_list.extend([3, 4])
print(my_list)  # Output: [1, 2, 3, 4]
```

    - **Explanation**: The `.extend()` method adds elements from another list (or any iterable) to the end of the current list.

13. To merge two lists, you can use the `+` operator, which combines two lists into a single list. Here's an example:

```
list1 = [1, 2]
list2 = [3, 4]
merged_list = list1 + list2
print(merged_list)
```

Output:

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

14. List Comprehension

List comprehension is a concise way to create lists using a single line of code. It can include a `for` loop with an optional condition.

Example:

```
squared_numbers = [x**2 for x in range(1, 6)]
print(squared_numbers)
# Output: [1, 4, 9, 16, 25]
```

List Comprehension with Condition

Example:

```
even_numbers = [x for x in range(10) if x % 2 == 0]
print(even_numbers)
# Output: [0, 2, 4, 6, 8]
```

Solve these problems:

1. Sum of All Elements

Problem Statement: Write a function that takes a list of numbers and returns the sum of all elements in the list.

Code:

```
def sum_of_elements(numbers):
    return sum(numbers)

# Example Usage
print(sum_of_elements([1, 2, 3, 4, 5]))  # Output: 15
```

---

2. Maximum Element

Problem Statement: Write a function that finds the maximum element in a given list of numbers.

Code:

```
def find_max(numbers):
    return max(numbers)

# Example Usage
print(find_max([1, 2, 3, 4, 5]))  # Output: 5
```

3. Count Odd and Even Numbers

Problem Statement: Write a function that counts how many odd and even numbers are in a given list.

Code:

```
def count_odd_even(numbers):
    odd_count = sum(1 for x in numbers if x % 2 != 0)
    even_count = len(numbers) - odd_count
    return odd_count, even_count

# Example Usage
print(count_odd_even([1, 2, 3, 4, 5]))  # Output: (3, 2)
```

---

4. Remove Duplicates

Problem Statement: Write a function that takes a list and returns a new list with duplicate elements removed.

Code:

```
def remove_duplicates(numbers):
    return list(set(numbers))

# Example Usage
print(remove_duplicates([1, 2, 2, 3, 4, 4]))  # Output: [1, 2, 3, 4]
```

5. Find the Second Largest Number

Problem Statement: Write a function that returns the second largest number from a list of numbers.

Code:

```python
def second_largest(numbers):
    unique_numbers = list(set(numbers))
    unique_numbers.sort()
    return unique_numbers[-2] if len(unique_numbers) >= 2 else None

# Example Usage
print(second_largest([1, 2, 3, 4, 5]))  # Output: 4
```

6. Reverse a List

Problem Statement: Write a function that reverses a given list without using the built-in reverse method.

Code:

```python
def reverse_list(lst):
    reversed_lst = []
    for item in lst:
        reversed_lst.insert(0, item)
    return reversed_lst

# Example Usage
print(reverse_list([1, 2, 3, 4, 5]))  # Output: [5, 4, 3, 2, 1]
```

7. Check for Subset

Problem Statement: Write a function that checks if one list is a subset of another list.

Code:

```
def is_subset(list1, list2):
    return set(list1).issubset(set(list2))

# Example Usage
print(is_subset([1, 2], [1, 2, 3, 4]))  # Output: True
```

8. Flatten a Nested List

Problem Statement: Write a function that takes a nested list (a list of lists) and flattens it into a single list.

Code:

```
def flatten_list(nested_list):
    return [item for sublist in nested_list for item in sublist]

# Example Usage
print(flatten_list([[1, 2], [3, 4], [5]]))  # Output: [1, 2, 3, 4, 5]
```

9. Merge Two Lists

Problem Statement: Write a function that merges two lists into a single list, alternating elements from each list.

Code:

```
def merge_lists(list1, list2):
    merged_list = []
    for i in range(max(len(list1), len(list2))):
        if i < len(list1):
            merged_list.append(list1[i])
        if i < len(list2):
            merged_list.append(list2[i])
    return merged_list

# Example Usage
print(merge_lists([1, 2, 3], ['a', 'b', 'c']))  # Output: [1, 'a', 2, 'b', 3, 'c']
```

Loop:

A loop is a programming construct that repeats a block of code multiple times until a specific condition is met or a sequence is exhausted. Loops are fundamental to programming because they allow you to automate repetitive tasks, process data, and build efficient solutions to complex problems.

Why Use Loops?

1. **Automate Repetitive Tasks:** Loops save time by automatically repeating actions that would otherwise require many lines of code.
    - Example: Printing numbers from 1 to 100 manually would take 100 lines, but a loop can achieve this in just a few lines.
2. **Process Collections of Data:** Loops allow you to process each item in a list, tuple, dictionary, or set systematically, which is essential in data processing.
    - Example: Summing all elements in a list of numbers or checking each value in a data set.
3. **Simplify Complex Logic:** Many algorithms, like sorting and searching, rely on loops to go through data efficiently and produce results.
4. **Dynamic Code Execution:** Loops enable dynamic programming where the number of repetitions or processing steps depends on user input or real-time data.

Types of Loops

1. **For Loop**: Runs a fixed number of times, typically used when iterating over a collection or range of values.

```
for i in range(5):
    print(i)  # Output: 0, 1, 2, 3, 4
```

2. **While Loop**: Runs until a specified condition becomes false, often used when the number of repetitions isn't known beforehand.

```
count = 0
while count < 3:
    print("Hello")  # Output: Hello (3 times)
    count += 1
```

3. **Nested Loops**: Loops inside loops, used for multi-dimensional data or complex logic.

```
for i in range(3):
    for j in range(2):
        print(f"i={i}, j={j}")
```

Examples of Loop Use-Cases

1. **Printing Elements in a List**

```
fruits = ["apple", "banana", "cherry"]
for fruit in fruits:
    print(fruit)
```

2. **Calculating the Sum of Numbers**

```
numbers = [1, 2, 3, 4, 5]
total = 0
for num in numbers:
    total += num
print(total)  # Output: 15
```

3. **Prompting Until Valid Input**

```
user_input = ""
while user_input != "yes":
    user_input = input("Type 'yes' to continue: ")
```

4. **While Loop to Find Maximum Element in List**

```
numbers = [3, 5, 7, 2, 8]
max_number = numbers[0]
index = 1
while index < len(numbers):
    if numbers[index] > max_number:
        max_number = numbers[index]
    index += 1
print(max_number)
# Output: 8
```

Dictionary Data Operations:

Dictionaries in Python are collections that store data in key-value pairs, allowing for efficient data retrieval using keys. Here are some key dictionary operations with examples:

1. Creating a Dictionary

```
# Creating a dictionary
student = {"name": "Alice", "age": 20, "grade": "A"}
print(student)
# Output: {'name': 'Alice', 'age': 20, 'grade': 'A'}
```

2. Accessing Values

To retrieve a value, use its key.

```
# Accessing value using the key
print(student["name"])  # Output: Alice

# Using `get()` to access value safely (avoids KeyError)
print(student.get("age"))  # Output: 20
print(student.get("school", "Not Found"))  # Output: Not Found
```

3. Adding or Updating a Key-Value Pair

```
# Adding a new key-value pair
student["school"] = "Greenwood High"
print(student)
# Output: {'name': 'Alice', 'age': 20, 'grade': 'A', 'school': 'Greenwood High'}

# Updating an existing key
student["grade"] = "A+"
print(student)
# Output: {'name': 'Alice', 'age': 20, 'grade': 'A+', 'school': 'Greenwood High'}
```

4. Removing Elements

You can remove items using `pop()`, `popitem()`, or `del`.

```
# Using `pop()` to remove a key-value pair by key
age = student.pop("age")
print(age)         # Output: 20
print(student)     # Output: {'name': 'Alice', 'grade': 'A+', 'school': 'Greenwood

# Using `popitem()` to remove the last key-value pair
last_item = student.popitem()
print(last_item)   # Output: ('school', 'Greenwood High')
print(student)     # Output: {'name': 'Alice', 'grade': 'A+'}

# Using `del` to remove a key-value pair
del student["grade"]
print(student)     # Output: {'name': 'Alice'}
```

5. Checking if a Key Exists

```
print("name" in student)     # Output: True
print("age" in student)      # Output: False
```

6. Looping Through a Dictionary

You can loop through keys, values, or key-value pairs.

```
student = {"name": "Alice", "age": 20, "grade": "A"}

# Looping through keys
for key in student:
    print(key)
# Output: name age grade

# Looping through values
for value in student.values():
    print(value)
# Output: Alice 20 A

# Looping through key-value pairs
for key, value in student.items():
    print(key, ":", value)
# Output: name : Alice
#         age : 20
#         grade : A
```

7. Merging Two Dictionaries

You can merge two dictionaries using the `update()` method or the `{**dict1, **dict2}` syntax.

```
student_info = {"name": "Bob", "age": 18}
student_grades = {"grade": "B+", "school": "Greenwood High"}

# Using `update()`
student_info.update(student_grades)
print(student_info)
# Output: {'name': 'Bob', 'age': 18, 'grade': 'B+', 'school': 'Greenwood High'}

# Using dictionary unpacking (Python 3.5+)
merged_student = {**student_info, **student_grades}
print(merged_student)
# Output: {'name': 'Bob', 'age': 18, 'grade': 'B+', 'school': 'Greenwood High'}
```

8. Clearing All Items

```
student.clear()
print(student)  # Output: {}
```

9. Dictionary Comprehension

A dictionary comprehension allows you to create dictionaries in a single line.

```
squares = {x: x*x for x in range(1, 6)}
print(squares)
# Output: {1: 1, 2: 4, 3: 9, 4: 16, 5: 25}
```

Summary of Operations

Operation	Code Example	Description
Access value	`student["name"]`	Retrieve value by key
Add/Update	`student["grade"] = "A+"`	Add or update key-value pair
Remove item	`student.pop("age")`	Remove by key
Check existence	`"name" in student`	Check if a key exists
Loop through items	`for key, value in student.items()`	Iterate over key-value pairs
Merge dictionaries	`student_info.update(student_grades)`	Merge two dictionaries
Clear dictionary	`student.clear()`	Remove all items
Dictionary comprehension	`{x: x*x for x in range(1, 6)}`	Create a dictionary with comprehension

Tuple Data Operations:

Tuples are immutable sequences in Python, meaning they cannot be modified after creation. They are useful for grouping data that should not change and are often used in situations where you want to ensure data integrity.

1. Storing Related Data Together

Tuples allow you to group related pieces of data, often representing a single object or entity, without worrying about accidental modification.

```
# Storing information about a person (name, age, city)
person = ("Alice", 30, "New York")
print(person)
# Output: ('Alice', 30, 'New York')

# Accessing tuple elements
print(person[0])  # Output: Alice
print(person[1])  # Output: 30
print(person[2])  # Output: New York
```

2. Using Tuples as Dictionary Keys

Tuples, being immutable, can be used as keys in dictionaries, unlike lists. This is particularly useful when you need a composite key.

```
# Creating a dictionary with tuple keys
location_data = {
    ("New York", "USA"): 8.4,
    ("London", "UK"): 9.3,
    ("Tokyo", "Japan"): 14.0
}

# Accessing data using tuple keys
print(location_data[("New York", "USA")])  # Output: 8.4
print(location_data[("Tokyo", "Japan")])   # Output: 14.0
```

3. Tuple Unpacking

Tuple unpacking allows you to assign values from a tuple directly to variables. This is particularly useful for readable and concise code.

```
# Assigning values from tuple to variables
dimensions = (1920, 1080)
width, height = dimensions
print("Width:", width)   # Output: Width: 1920
print("Height:", height) # Output: Height: 1080
```

Function In Python:

In Python, a function is a reusable block of code designed to perform a specific task. Functions help to break down complex problems into smaller, manageable parts, making the code more readable, reusable, and organized.

1. Defining a Simple Function

A function is defined using the `def` keyword, followed by the function name and parentheses `()`.

```
# Define a function
def greet():
    print("Hello, welcome to Python!")

# Call the function
greet()
# Output: Hello, welcome to Python!
```

2. Function with Parameters

Parameters allow functions to accept inputs. These values are specified when the function is called and can be used within the function.

```
# Function with one parameter
def greet(name):
    print(f"Hello, {name}! Welcome to Python!")

# Calling the function with an argument
greet("Alice")
# Output: Hello, Alice! Welcome to Python!
```

3. Function with Multiple Parameters

You can define functions with multiple parameters, allowing you to pass multiple arguments.

```
# Function with two parameters
def add_numbers(a, b):
    result = a + b
    print("The sum is:", result)

# Calling the function with two arguments
add_numbers(5, 10)
# Output: The sum is: 15
```

4. Function with Return Value

The `return` statement allows a function to send a result back to the caller. This makes the function more flexible and allows further manipulation of its output.

```
# Function with return value
def multiply(x, y):
    return x * y

# Calling the function and storing the result
result = multiply(4, 3)
print("The product is:", result)
# Output: The product is: 12
```

5. Default Parameter Value

You can assign default values to function parameters. If an argument is not provided, the default value will be used.

```python
# Function with a default parameter
def greet(name="Guest"):
    print(f"Hello, {name}! Welcome to Python!")

# Calling the function without providing an argument
greet()  # Output: Hello, Guest! Welcome to Python!

# Calling the function with an argument
greet("Bob")  # Output: Hello, Bob! Welcome to Python!
```

6. Using `*args` for Variable Number of Arguments

The `*args` syntax allows a function to accept any number of positional arguments, making it flexible for cases where you don't know the exact number of inputs.

```python
# Function with variable number of arguments
def add_all(*numbers):
    total = sum(numbers)
    print("The sum is:", total)

# Calling the function with multiple arguments
add_all(1, 2, 3)       # Output: The sum is: 6
add_all(4, 5, 6, 7, 8) # Output: The sum is: 30
```

7. Using `**kwargs` for Keyword Arguments

The `**kwargs` syntax allows you to pass a variable number of keyword arguments, which are received as a dictionary.

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

# Calling the function with multiple keyword arguments
print_info(name="Alice", age=30, city="New York")
# Output:
# name: Alice
# age: 30
# city: New York
```

8. Lambda Functions (Anonymous Functions)

Lambda functions are short, anonymous functions defined with the `lambda` keyword. They are useful for simple, one-line functions.

```
# Lambda function for adding two numbers
add = lambda x, y: x + y
print(add(3, 5))  # Output: 8

# Lambda function in a list comprehension
squares = [(lambda x: x**2)(i) for i in range(5)]
print(squares)  # Output: [0, 1, 4, 9, 16]
```

9. Function with Docstring

A docstring is a descriptive text within triple quotes, added as the first line of a function, which describes its purpose.

```
def greet(name):
    """This function greets the person whose name is passed as an argument."""
    print(f"Hello, {name}!")

# Accessing the function's docstring
print(greet.__doc__)
# Output: This function greets the person whose name is passed as an argument.
```

10. Function with Recursion

A recursive function is a function that calls itself. It's commonly used for problems that can be broken down into sub-problems.

```
# Recursive function to calculate factorial
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n - 1)

# Calling the recursive function
print(factorial(5))  # Output: 120
```

Modules In Python:

Modules in Python are simply files containing Python code, such as functions, classes, and variables, that can be reused in other programs. By creating modules, you can organize code logically, prevent duplication, and share code across multiple scripts.

1. Creating a Module

To create a module, just save a Python file with functions, classes, or variables that you want to reuse. For example, let's create a module called `math_operations.py` with some mathematical functions.

`math_operations.py` (Module File)

```python
# This is the module with various math operations

def add(a, b):
    """Return the sum of two numbers."""
    return a + b

def subtract(a, b):
    """Return the difference between two numbers."""
    return a - b

def multiply(a, b):
    """Return the product of two numbers."""
    return a * b

def divide(a, b):
    """Return the division of two numbers, handling division by zero."""
    if b == 0:
        return "Cannot divide by zero"
    return a / b
```

2. Using a Module in Another File

Once you have created your module, you can import it into another Python script using the `import` statement. Here's how to use `math_operations` in a new file.

`main.py` (Script that Uses the Module)

```
# Importing the math_operations module
import math_operations

# Using functions from the math_operations module
result1 = math_operations.add(10, 5)
result2 = math_operations.subtract(10, 5)
result3 = math_operations.multiply(10, 5)
result4 = math_operations.divide(10, 5)
result5 = math_operations.divide(10, 0)  # Testing division by zero

print("Addition:", result1)          # Output: Addition: 15
print("Subtraction:", result2)       # Output: Subtraction: 5
print("Multiplication:", result3)    # Output: Multiplication: 50
print("Division:", result4)          # Output: Division: 2.0
print("Division by zero:", result5)  # Output: Cannot divide by zero
```

3. Using `from...import` to Import Specific Functions

You can import specific functions or variables from a module instead of the entire module. This makes your code more efficient and easier to read.

`main.py` with Specific Imports

```
# Importing specific functions from math_operations
from math_operations import add, divide

# Using the imported functions directly
print("Addition:", add(8, 2))       # Output: Addition: 10
print("Division:", divide(8, 2))    # Output: Division: 4.0
```

4. Renaming Imports with `as`

If the module name is long, you can give it a shorter alias using `as`.

```
# Importing with an alias
import math_operations as math_ops

# Using functions from the module with the alias
print("Addition:", math_ops.add(3, 7))      # Output: Addition: 10
print("Multiplication:", math_ops.multiply(3, 7))  # Output: Multiplication: 21
```

5. Built-in Python Modules

Python comes with many built-in modules, such as `math`, `random`, and `datetime`. Here's how to use some of them.

Example with `math` and `random` Modules

```
import math
import random

# Using math module
print("Square root of 16:", math.sqrt(16))       # Output: Square root of 16: 4.0
print("Value of pi:", math.pi)                   # Output: Value of pi: 3.14159265

# Using random module
print("Random number between 1 and 10:", random.randint(1, 10))
```

6. Creating and Using a Package

A package is a collection of modules organized in a directory. Each package must contain a special `__init__.py` file (which can be empty) to indicate that the directory is a package. Here's how to create and use a package:

Directory Structure

```
my_package/
    __init__.py
    math_operations.py
    string_operations.py
```

`string_operations.py` (Another Module in the Package)

```python
# This module contains string operations

def to_uppercase(s):
    """Convert a string to uppercase."""
    return s.upper()

def to_lowercase(s):
    """Convert a string to lowercase."""
    return s.lower()
```

Using the Package in `main.py`

```python
# Importing modules from the package
from my_package import math_operations
from my_package import string_operations

# Using functions from math_operations
print("Addition:", math_operations.add(2, 3))  # Output: Addition: 5

# Using functions from string_operations
print("Uppercase:", string_operations.to_uppercase("hello"))  # Output: HELLO
print("Lowercase:", string_operations.to_lowercase("HELLO"))  # Output: hello
```

Builtin DateTime Module:

The datetime module in Python provides classes for manipulating dates and times in a simple and efficient way. Here's a look at some common operations using datetime.

1. Current Date and Time

To get the current date and time, use `datetime.datetime.now()`.

```
from datetime import datetime

# Get the current date and time
current_datetime = datetime.now()
print("Current Date and Time:", current_datetime)  # e.g., 2024-11-04 10:21:35.1234
```

2. Formatting Dates and Times

The `strftime` method is used to format `datetime` objects into readable strings.

```
# Formatting current date and time in different ways
formatted_date = current_datetime.strftime("%Y-%m-%d")         # Year-Month-Day
formatted_time = current_datetime.strftime("%H:%M:%S")         # Hour:Minute:Second
custom_format = current_datetime.strftime("%A, %d %B %Y")      # e.g., Monday, 04 N

print("Formatted Date:", formatted_date)       # Output: 2024-11-04
print("Formatted Time:", formatted_time)       # Output: 10:21:35
print("Custom Format:", custom_format)         # Output: Monday, 04 November 2024
```

3. Creating Specific Dates and Times

You can create a `datetime` object for a specific date and time by using `datetime(year, month, day, hour, minute, second)`.

```
# Creating a specific date and time
birthday = datetime(1995, 7, 14, 9, 30)  # 14th July 1995, 9:30 AM
print("Birthday:", birthday)  # Output: 1995-07-14 09:30:00
```

4. Calculating Time Differences

Use `timedelta` to calculate differences between dates and times.

```
from datetime import timedelta

# Calculate the difference between two dates
today = datetime.now()
yesterday = today - timedelta(days=1)

print("Today:", today)
print("Yesterday:", yesterday)
print("Difference:", today - yesterday)  # Output: 1 day, 0:00:00
```

5. Working with `date` and `time` Classes

The `datetime` module also has `date` and `time` classes if you only need one of those.

```
from datetime import date, time

# Date-only object
today_date = date.today()
print("Today's Date:", today_date)  # Output: 2024-11-04

# Time-only object
specific_time = time(10, 15, 30)  # 10:15:30 AM
print("Specific Time:", specific_time)  # Output: 10:15:30
```

Error Handling:

Error handling in Python allows you to manage exceptions that occur during code execution. This is essential for making your code more robust and handling unexpected conditions gracefully.

In Python, error handling is done using `try`, `except`, `else`, and `finally` blocks.

Basic Syntax of Error Handling

```
try:
    # Code that may raise an exception
except SomeException:
    # Code that runs if an exception occurs
else:
    # Code that runs if no exception occurs
finally:
    # Code that runs no matter what (optional)
```

Example 1: Handling Division by Zero

Here's a simple example of handling a common error, division by zero.

```
try:
    numerator = 10
    denominator = 0
    result = numerator / denominator
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
else:
    print("Division successful, result is:", result)
finally:
    print("Execution complete.")
```

Output:

```
Error: Cannot divide by zero!
Execution complete.
```

Example 2: Handling Multiple Exceptions

You can handle different types of exceptions using multiple `except` blocks.

```
try:
    value = int(input("Enter a number: "))
    result = 100 / value
except ValueError:
    print("Error: Invalid input! Please enter a valid integer.")
except ZeroDivisionError:
    print("Error: Cannot divide by zero!")
else:
    print("Division successful, result is:", result)
finally:
    print("Execution complete.")
```

In this example, the `finally` block ensures that the file is closed, even if an exception occurs during file reading.

Example 3: Raising Exceptions

You can use the `raise` keyword to trigger an exception manually. This is useful for validating inputs.

```
def check_age(age):
    if age < 0:
        raise ValueError("Age cannot be negative!")
    print("Age is valid.")

try:
    check_age(-5)
except ValueError as e:
    print("Error:", e)
```

Output:

```
Error: Age cannot be negative!
```

Debugging:

Debugging is the process of finding and resolving errors or bugs in your code. In Python, you can use the `pdb` module (Python Debugger) to inspect code execution step-by-step. This helps you understand how your code runs and identify where things might be going wrong.

Here's how you can debug Python code and an example using `pdb.set_trace()` to pause execution and examine your program's state.

Example Code to Debug

Let's say you have a piece of code that calculates the average of a list of numbers. However, it throws an error when the list is empty.

```
def calculate_average(numbers):
    total = sum(numbers)
    count = len(numbers)
    average = total / count
    return average

# Test case
print("Average:", calculate_average([10, 20, 30]))  # Expected output: Average: 20.
print("Average:", calculate_average([]))            # Should raise an error due to
```

When running this code with an empty list, you'll encounter a `ZeroDivisionError` because dividing by zero is not allowed.

Debugging with `pdb`

To debug this, you can insert `import pdb; pdb.set_trace()` right before the line you want to examine. The `pdb.set_trace()` line will act as a breakpoint, stopping the code execution at that point so you can inspect variables and step through the code.

Steps to Debug

1. **Import `pdb` and insert `pdb.set_trace()`**: Place it before the problematic line or wherever you want to start examining the code.
2. **Run the code**: When the code reaches `pdb.set_trace()`, it will pause and give you a prompt to start debugging.
3. **Use debugging commands** to inspect variables, step through lines, and understand the flow.

Here's how it would look with `pdb.set_trace()`:

```
import pdb

def calculate_average(numbers):
    total = sum(numbers)
    count = len(numbers)
    pdb.set_trace()  # Set a breakpoint
    average = total / count  # This line may raise an error if count is zero
    return average

# Test case
print("Average:", calculate_average([10, 20, 30]))  # Expected output: Average: 20.
print("Average:", calculate_average([]))            # Should raise an error due to
```

Using `pdb` Commands

When you run this code, it will pause at `pdb.set_trace()` and enter interactive debugging mode. You can use the following commands to debug:

- `n` **(next)**: Execute the next line of code.
- `c` **(continue)**: Continue execution until the next breakpoint.
- `p` **(print)**: Print the value of a variable, e.g., `p count`.
- `q` **(quit)**: Exit the debugger and stop the program.

Cheers!